Framework til netværksprotokol analyse og I^{2}C analysator

af Kristian Kjær, s021727

Polyteknisk Eksamensprojekt
IMM-M.Eng-2007-53

Institut for Informatik og Matematisk Modellering
Danmarks Tekniske Universitet
Kongens Lyngby 2007

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

In many situations it is valuable to be able to diagnose the traffic on computer networks. These networks range from small range networks in embedded systems to large scale networks such as the Internet.

This report examines the possibilities to create a general purpose framework, which can be used to create a dedicated protocol analyzer. Thus the framework is completely independent of the network infrastructure. The framework provides access to sniffing from and interaction with a network. The purpose of the framework is to make the implementation of a network specific diagnostic tool easy.

A prototype of the framework has been produced in the C# language with .NET as the target platform. The report describes the design and implementation of this prototype.

The framework applies NetPDL to create a general and understandable description of the defined protocols such that patterns in the data stream on the network can be identified. This NetPDL description can be used to define filters, too. It is possible to keep the framework independent of a specific network by using complicated data structures.

The interaction with the network has been implemented by integrating the Ruby language to create scripts for the interaction. The framework offers a communication bridge to the scripts, where a dedicated communication protocol is used. This bridge makes Ruby able to control parts of the framework.

User surveys have been conducted to find the needed functionality of the framework. The report includes user tests of the prototype, too. These tests are used to examine whether or not the findings in the survey have been fulfilled.

The report is in Danish.

Resume

I mange situationer er der brug for at diagnosticere trafikken på et computernetværk. Disse computernetværk kan være meget forskelligartede og omfatter alt fra små netværk mellem indlejrede komponenter til store netværk som Internettet.

I denne rapport undersøges muligheden for et generelt framework, der kan danne basis for et netværksspecifikt værktøj, en protokolanalysator. Frameworket er altså helt uafhængig af et specifikt netværk. Frameworket giver mulighed for at sniffe på og interagere med netværket. Formålet med et sådant framework er at lette implementeringen af et diagnosticeringsværktøj til et specifikt netværk.

En prototype af frameworket er fremstillet i C# til .NET platformen og rapporten beskriver design og implementering af denne prototype.

Frameworket benytter NetPDL til at skabe en generelt forståelig beskrivelse af de definerede protokoller, så mønstre i datastrømmen fra netværket kan identificeres. Denne beskrivelse kan også benyttes til en specifikation af filtre. Det har vist sig at være muligt at undgå protokolspecifikke elementer i frameworket ved benyttelse af komplicerede datastrukturer for netværkspakkerne.

Interaktion med netværket er mulig vha. Ruby scripts. Frameworket skaber en kommunikationsbro til Ruby scripts, hvor der kommunikeres vha. en dedikeret protokol, så et Ruby script kan styre dele af frameworket.

Frameworkets funktionalitet er specificeret på basis af brugerundersøgelser, og rapporten inkluderer brugertests, der vurderer opfyldelsen af de fundne brugerkrav.

Forord

Denne rapport er et eksamensprojekt til civilingeniøruddannelsen på Danmark Tekniske Universitet. Den er lavet i samarbejde med Institut for Informatik og Matematisk Modellering og FOSS A/S. Projektet er udført af undertegnede med Bjarne Poulsen fra IMM, DTU og Lars Jeppesen, FOSS som vejledere.

For det største udbytte af rapporten anbefales det at have et basalt forhåndskendskab til følgende teknologier: C# sproget, .NET platformen og Ruby sproget. Desuden er et kendskab til basale datastrukturer samt computernetværk herunder almindelige protokolopbygninger en fordel. Den almindelige protokolstak, Ethernet/IP/TCP/HTTP, benyttes til eksempler på den fremsatte teori.

Projektet er gennemført i perioden 1/9 2006 til 15/6 2007.

Kristian Kjær

Kongens Lyngby 2007

Anerkendelser

Jeg vil gerne takke FOSS og TSC teamet for at tilbyde dette projekt og for godt samarbejde. Jeg takker alle, som har støttet mig gennem projekt perioden. Min tak går til

Vejledning
Bjarne Poulsen, IMM/DTU
Lars Jeppesen, TSC FOSS

Korrekturlæsning
Jacob Oettinger
Mogens Kjær

Installationstests
Mogens Kjær
Anders Kjær

Indholdsfortegnelse

1 Indledning **1**
1.1 Om FOSS . . . 1
1.2 Baggrund, motivation og vision . . . 2
1.2.1 Traditionel opbygning af apparater . . . 2
1.2.2 Ny opbygning af apparater . . . 2
1.2.3 Om I^2C . . . 3
1.2.4 Motivation og ønsker . . . 4
1.2.5 Projektets vision . . . 5
1.3 Foranalyse . . . 5
1.3.1 Identifikation af udfordringer . . . 5
1.3.2 Benyttet teknologi . . . 6
1.3.3 Metodevalg . . . 6
1.3.4 Foreløbige designbetragtninger . . . 7
1.3.5 State of the art . . . 7
1.4 Om rapporten . . . 8
1.4.1 Om engelske udtryk . . . 9
1.4.2 Rapportens omfang . . . 9
1.4.3 Medfølgende CD . . . 9

2 Indledende analyse **11**
2.1 Brugerkrav . . . 11
2.1.1 Identifikation af brugere . . . 12
2.1.2 Metoder . . . 12
2.1.3 Fremgangsmåde i formelle interviews . . . 14
2.1.4 Udførte interviews . . . 15
2.2 Kravspecifikation . . . 15
2.2.1 Frameworket . . . 16
2.2.2 Protokolanalysator . . . 17
2.2.3 Plug-ins . . . 18
2.3 Format af protokolspecifikke beskeder . . . 19
2.3.1 Krav til formatet . . . 19
2.3.2 Overordnet løsning . . . 20
2.3.3 Mulige eksterne formater . . . 22
2.4 Om NetPDL . . . 25
2.4.1 Muligheder NetPDL . . . 25
2.4.2 Eksempler . . . 26

2.4.3 Begrænsninger i NetPDL 28
2.4.4 Opdelte pakker 29
2.4.5 Eksisterende NetPDL værktøjer 31
2.5 Muligheder for interaktion med netværket 31
2.6 Konklusion på indledende analyse 32

3 Design af systemet 35
3.1 Overordnet design strategi 35
3.2 Brug af modulopbygget design 36
3.2.1 Kommunikation mellem moduler 37
3.3 Design af frameworkets arkitektur 38
3.3.1 Datakildelaget 38
3.3.2 Domænelaget 38
3.3.3 Præsentationslaget 40
3.3.4 Kommunikation i arkitekturen 42
3.4 Dynamisk flow af netværkspakker 43
3.4.1 Pakkens datastrukturer 44
3.4.2 Pakkedatabasens struktur 45
3.4.3 Struktur af filter 46
3.5 Integration af script sprog 46
3.5.1 Generering af hjælpebibliotek 47
3.5.2 Kommunikation mellem framework og Ruby scripts 49
3.6 Interface til protokolanalysator 51
3.7 Interface til plugin 52
3.7.1 Plug-ins i C# 53
3.8 Konklusion på design af systemet 54

4 Implementering og teknisk test 55
4.1 Implementeringsmetode og strategi for teknisk test 56
4.2 Generel opbygning af framework 57
4.2.1 Implementering af kommunikation mellem moduler 57
4.3 Protokoldatabase 59
4.3.1 Filtrering 60
4.4 Pakkehåndtering og pakkedatabase 63
4.4.1 Store datamængder i pakkedatabasen 63
4.4.2 Store datamængder i pakkevisningen 64
4.4.3 Brug af NetBee 65
4.5 Integration af Ruby 65
4.5.1 Ruby hjælpebibliotek 66
4.5.2 Eksekvering af scripts 66
4.5.3 Kommunikation mellem framework og Ruby scripts 67
4.6 Brug af frameworket 67
4.7 Plug-ins 68
4.7.1 Integration af plug-ins 69
4.8 Konklusion på implementering og teknisk test 69

5 Case: I^{2}C protokolanalysator 71
5.1 Realisering af I^{2}C protokolanalysator 71

6 Case: CAN protokolanalysator 73

6.1 Motivation for en CAN protokolanalysator 73
6.2 Design af CAN protokolanalysator 74
6.2.1 Format af protokollerne 74
6.2.2 FOSS specifikke protokoller 75
6.2.3 Brug af CAN API 77
6.3 Implementering af CAN protokolanalysator 78
6.3.1 Implementering af håndtering af lange pakker 79
6.4 Brugertests 80

7 Eksempler på plug-ins **83**
7.1 Farver i pakkevisningen 83
7.2 Dump af pakkedata 84

8 Konklusion **87**
8.1 Overordnede betragtninger 87
8.2 Konklusioner om frameworket 88
8.3 Konklusioner om protokolanalysatorer 88
8.4 Bruger konklusioner 89
8.5 Perspektivering 89
8.6 Virksomhedskonklusion på studenterprojekt 90
8.7 Endelig konklusion 90

A Undersøgelser af brugerkrav **91**
A.1 Interview med Mogens Velsing 91
A.1.1 Om Mogens Velsing 91
A.1.2 Resultater af interviewet 92
A.2 Uformelle interviews med ESWP udviklere 93
A.2.1 Resultater 93

Appendices **91**

B Foretagede brugertests **95**
B.1 Resultater 95

C Eksempler på NetPDL protokol specifikationer **97**
C.1 Ethernet-IP-TCP-HTTP stakken 97
C.1.1 Ethernet 97
C.1.2 IP 98
C.1.3 TCP 103
C.1.4 HTTP 106
C.2 I^2C protokollen 108
C.3 FOSS' CAN protokol 111

D NetPDL problemer **117**
D.1 Problemer ved definition af felt 117
D.2 Problemer med visualiserings 118

E Oversigt over klasser i systemet **121**
E.1 Domain 121
E.2 Presentation 122
E.3 PaInterface 123

E.4 NetPDLwrapper 123
E.5 Utils 124

Litteratur **125**

Kapitel 1

Indledning

Kapitlet giver en introduktion til det problemområde, der præsenteres i denne rapport. Der gives baggrundsinformation, der er relevant for valget af projektet. FOSS, som projektet udføres i samarbejde med, er en del af baggrunden for projektet. Kapitlet indledes derfor med en meget kort beskrivelse af virksomheden. Denne baggrundsinformation bruges til at definere, hvad motivationen for projektet er, og der gives en egentlig vision for projektet. Dernæst gives en overordnet indledende analyse af problemområdet. I denne redegøres for, hvilke udfordringer projektet stiller, samt hvilke teknologier, der forventes at kunne benyttes. For disse teknologier ses der på, hvad „State of the art“ er. Kapitlet sluttes af med en oversigt over rapportens opbygning.

Indholdsfortegnelse

1.1 Om FOSS 1
1.2 Baggrund, motivation og vision 2
1.3 Foranalyse 5
1.4 Om rapporten 8

1.1 Om FOSS

FOSS blev startet i 1956 af Niels Foss og har i dag over 1000 medarbejdere på verdensplan (tal fra 2005). Firmaet laver elektriske analyseapparater til fødevareindustrien. I starten blev der lavet apparater til analyse af korn. De kunne bl.a. måle kornets fugtighed, som er en væsentlig faktor i bestemmelsen af kornets kvalitet. I dag laves der apparater til en lang række andre formål, fx er der apparater til måling af alkoholindholdet i øl, indholdet af fedt i tørt og vådt dyrefoder, målinger af farmaceutiske piller, heriblandt konsistens, tykkelse m.m., måling af renheden af sukker og mange andre formål.

Måleudstyret benytter forskellige teknologier til målingerne, fx infrarød spektroskopi, der benyttes til måling af kemiske egenskaber af fx mælk og vin, røntgen til at måle fedtindholdet i både frisk og frossent kød og automatisk billedanalyse til inspektion af korn.

Teknologien implementeres i apparater, der sælges til FOSS' kunder over hele verden. Apparaterne skjuler den egentlige teknologi for brugeren (kunden), så brugeren kun ser en brugergrænseflade (kontrolenhed) til produktet. Apparatet benyttes af brugeren gennem denne kontrolenhed [13].

1.2 Baggrund, motivation og vision

FOSS' måde at implementere teknologi i deres apparater er i en udviklingsfase, hvor den traditionelle måde at opbygge apparaterne på erstattes af en ny og bedre opbygning. Baggrunden for projektet er denne udviklingsfase. I dette afsnit gives en redegørelse for hhv. den traditionelle opbygning og den nye opbygning. Denne redegørelse danner baggrund for en beskrivelse af motivationen for projektet, der benyttes til at specificere projektets vision.

1.2.1 Traditionel opbygning af apparater

Hidtil er apparaterne blevet lavet meget individuelt, dvs. at en kravspecifikation har været grundlaget for apparatets design og fremstilling. Enkelte dele (hardware og software) har man evt. kunnet tage fra tidligere fremstillede apparater og tilpasse dem til nye apparater.

Disse apparater har traditionelt været implementeret i ét individuelt system med én micro controller. Senere apparater har været mere komplekse og er derfor blevet opbygget med flere selvstændige moduler. Når disse moduler skulle kommunikere med hinanden, er det sket primært vha. af to protokoller: I^2C (afsnit 1.2.3) og ARCNET[1]. Modulerne skulle dog stadig, som beskrevet ovenfor, tilpasses for at kunne benyttes i nye apparater.

1.2.2 Ny opbygning af apparater

Den traditionelle opbygning lider af nogle overordnede problemer:

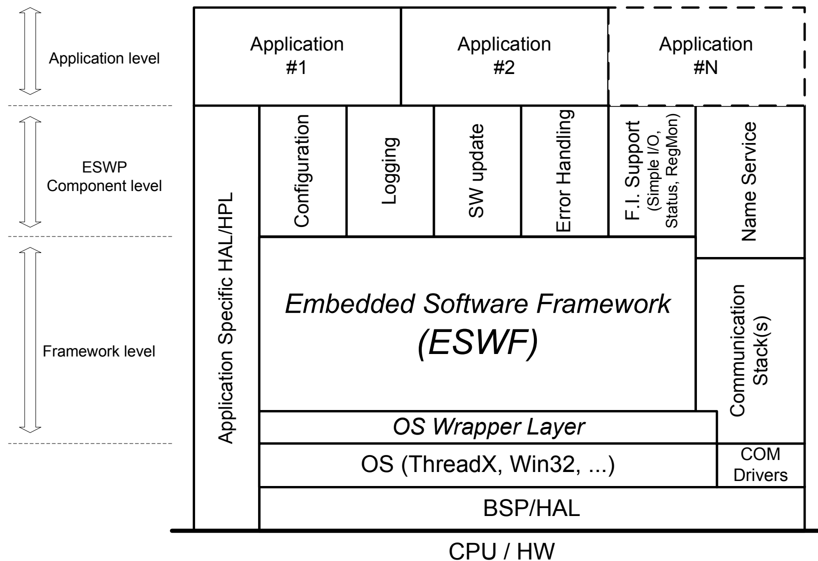
- Der er forholdsvis meget arbejde i udviklingen af et nyt apparat.
- Da apparaternes opbygning varierer, bliver den support, som FOSS skal yde sine kunder, besværliggjort. Der er ikke ens fejl i forskellige apparater, så det kræver ekspert viden inden for hver enkel type apparat at reparere, vedligeholde og diagnosticere fejl i apparaterne.
- Det er besværligt at opdatere og rette software fejl i apparaterne, da mekanismen for dette er meget forskellig i de enkelte apparater.

I den nye opbygning har man imødekommet nogle af disse problemer. Apparaterne opbygges af moduler som ved den tidligere metode, men i den nye opbygning benytter modulerne hver især den samme software platform, ESWP[2] –

[1] for yderligere informationer om ARCNET, se fx `http://www.arcnet.com/` – tilgængelig 1/6-2007.

[2] Embedded SoftWare Platform.

se figur 1.1[3]. Via denne platform styres modulerne af et software applikationslag („Application Level“ på figur 1.1).



Figur 1.1: Opbygning af modul.

Denne nye opbygning reducerer ovenstående problemer, da modul opbygningen gør, at moduler (hardware og software) i langt højere grad kan genbruges, ligesom ensartetheden i apparaterne gør support, reparation og opgradering/opdatering betydelig lettere.

I den nye opbygning kommunikerer modulerne med en central enhed i apparatet via en I^{2}C bus. Denne kommunikation er standardiseret vha. en protokol (Hextet[4]) og beskederne, der sendes vha. Hextet protokollen, er i et standard format (T123 beskeder) eller rå applikationsspecifik data. Hextet såvel som T123 beskeder er begge FOSS' opfindelser. Via disse protokoller bliver kommunikationen i det nye system ensartet, så der er mulighed for at udvikle generelle værktøjer til diagnosticering af kommunikationen. Dette projekt omhandler udviklingen af et sådant værktøj.

Et værktøj til diagnosticering af kommunikationen på I^{2}C netværket er specielt brugbart for FOSS ved udvikling af nye moduler, da værktøjet kan benyttes af udvikleren til test af modulets kommunikation. I det følgende gives en beskrivelse af I^{2}C protokollen, da denne danner grundlag for kommunikationen i FOSS apparaterne.

1.2.3 Om I^{2}C

I^{2}C bussen er en simpel netværksbus, der blev opfundet af Phillips til simpel kommunikation mellem komponenter på en printplade. Kommunikationen er ma-

[3]Figuren er tegnet af Jacob Bodholdt, FOSS.

[4]Principielt benyttes både Octet og Hextet, men i denne rapport benævnes de Hextet under ét, da de ligger i samme lag i protokolstakken, dvs. de benytter begge I^{2}C. Der er heller ikke stor forskel, da forskellen ligger i dataformatet – ikke i selve kommunikationsmekanismen.

ster/slave baseret – al kommunikation startes fra master, der kan læse fra eller skrive til en slave. Bussen kan benyttes på netværk med højst 112 komponenter. I^2C har en simpel kommunikations specifikation. Kommunikationen starter med en identifikation af destinationen af beskeden, der efterfølges af en indikation af, om det er en læse eller skrive besked. Til slut sendes den egentlige data, kun afbrudt af acknowledge signaler fra slaven. For yderligere oplysninger om kommunikationen henvises til [15].

Hos FOSS benyttes protokollen ikke til kommunikation mellem komponenter på en printplade, men i stedet mellem selvstændige moduler. Kommunikationen foregår derfor over større afstande.

Interface til I^2C netværket

For at kunne diagnosticere kommunikationen på I^2C bussen vha. et PC værktøj, har FOSS en adapter til rådighed, der kan lytte passivt til (sniffe) dataene på netværket og stille dem til rådighed for PC softwaren via USB. Man kan få adgang til adapterens funktioner via et dll-interface, der kan benyttes i et Windows-program. FOSS har desuden en anden adapter til rådighed, der udover at lave en simpel sniffing af data også kan benyttes som injektor, hvor data sendes ud på bussen.

Disse adaptere kan benyttes, når der skal kommunikeres med og sniffes på I^2C netværket.

1.2.4 Motivation og ønsker

I afsnit 1.2.2 blev det beskrevet, hvilke ideer der ligger bag den nye måde at opbygge apparater på. Det fremgik, at der er behov for at teste kommunikationen på netværket. Derfor har FOSS et ønske om at få et værktøj, der kan udføre denne opgave. Motivationen for projektet er at udvikle et sådant værktøj. Værktøjet skal benyttes internt i FOSS. FOSS har følgende ønsker/krav til værktøjet:

1. Det skal have den funktionalitet, der både er nødvendig og tilstrækkelig for brugerne. En del af opgaven er derfor først at få klarlagt, hvem brugerne præcist er og derefter finde ud af, hvilken funktionalitet brugerne har brug for i værktøjet.

2. Værktøjet skal dels kunne observere kommunikationen på I^2C netværket passivt og dels være en del af kommunikationen.

3. Værktøjet skal udvikles med Windows XP som mål platform.

4. Det skal være så simpelt som muligt at ændre værktøjet, så den analyserer en anden protokol, så en udskiftning af bus eller protokol i FOSS' moduler ikke forælder værktøjet.

Det sidste punkt skyldes, at FOSS har planer om i fremtiden at udskifte kommunikationsbussen (og protokolstakken) i apparaterne, da I^2C bussen i forandringsfasen har vist svaghedstegn. Dette faktum giver motivation for, at designet deles i et protokoluafhængigt framework og en protokolspecifik overbygning. Ved udskiftning af protokollen i systemet er det kun nødvendigt at skifte den protokolspecifikke overbygning, mens det underliggende framework forbliver uændret.

1.2.5 Projektets vision

På baggrund af ovenstående redegørelse for projektets baggrund og motivation i FOSS' ønske om et værktøj, kan der gives en overordnet vision. Visionen kan ses som projektets hovedmål.

Projektets vision er at undersøge muligheden for at lave et framework, der kan benyttes ved udvikling af en protokolanalysator til computernetværk. Det skal undersøges, hvordan et sådant framework kan realiseres fleksibelt, så det ikke begrænser protokolanalysatoren. Designet skal være let at udvide og vedligeholde. Målet med projektet er at skabe en prototype af frameworket, der kan benyttes af en konkret protokolanalysator. Som eksempel på den praktiske brug af frameworket skal der udvikles en protokolanalysator til en FOSS specifik protokolstak, der kommunikerer vha. en I^2C bus. Udover at teste prototypen af frameworket teknisk, kan protokolanalysatoren bruges til at brugerteste prototypen.

1.3 Foranalyse

I dette afsnit gives en foreløbig analyse af problemstillingerne i projektet. De største udfordringer i projektet identificeres og metoder samt teknologi til løsning af problemstillingerne introduceres.

1.3.1 Identifikation af udfordringer

Projektet stiller følgende udfordringer, der skal løses:

1. **Fleksibelt design** Et fleksibelt framework kræver en nøje analyse af designet. Der skal udarbejdes interfaces, der kan benyttes af protokolanalysatoren. Fleksibiliteten af frameworket afhænger af fleksibiliteten af interfacene. Dog skal interfacene yde den fornødne abstraktion, så frameworket hjælper udviklingen af protokolanalysatoren væsentligt. Men frameworket skal også designes fleksibelt mht. fremtidig ny intern funktionalitet, da der i fremtiden kan opstå uforudsete behov. Disse aspekter giver følgende udfordringer:

    a) Software modul integration og plug-in arkitektur.

    b) Identifikation af relevante design patterns.

    c) Generisk mønster genkendelse i datastrøm – gerne realtime.

2. **Identifikation af krav** Kravene til frameworket skal identificeres i samarbejde med de potentielle brugere af det færdige produkt. Disse krav vil utvivlsomt stille tekniske udfordringer, der ikke kan afklares her.

3. **Definition af protokolstak** Der skal defineres et fleksibelt format, som protokolstakke kan defineres i. Dette format benyttes af frameworket til at analysere den rå datastrøm og finde protokolspecifikke mønstre, jf. punkt 1c.

4. **Finde metode til styring af kommunikation** Når frameworket skal kommunikere på netværket, er der brug for en metode, hvorpå brugeren kan definere den ønskede kommunikation (fx vha. et simpelt script sprog).

1.3.2 Benyttet teknologi

I projektet benyttes følgende teknologi og værktøjer:

.NET framework Mål systemet er Windows XP, hvorfor .NET er et oplagt valg af platform. Her findes en lang række biblioteker med fundamental funktionalitet. Derved kan vægten i implementeringen lægges på projektets egentlige udfordringer, jf. ovenstående. En yderligere fordel ved valget er, at man kan forvente at næste generation af Windows, Windows Vista, understøtter .NET frameworket fuldstændigt, da begge produkter udvikles af Microsoft.

C# Da .NET frameworket benyttes, kan der benyttes en række programmeringssprog. Af samme grund er det mindre vigtigt, hvilket sprog der benyttes, da senere videreudvikling kan foretages i andre sprog. I projektet benyttes C#, da det blev udviklet i forbindelse med udviklingen af .NET frameworket, og det er derfor den mest direkte vej til frameworket [10].

NUnit De tekniske tests kan med fordel udføres vha. NUnit, der er et unit tests framework til .NET platformen.

C++/CLI En udvidelse af C++, som Microsoft har lavet for at gøre det muligt at skrive C++ til .NET platformen. Man kan benytte både managed og unmanaged kode, men resultatet er managed, så det kan bruges i andre programmer til .NET platformen. Sproget udmærker sig, når man ønsker at benytte unmanaged kode i et .NET projekt [5].

Ruby Et script sprog, der er udviklet med enkelhed som hovedmål. Sproget benyttes i forvejen hos FOSS, så kan dette sprog bruges i værktøjet, er det en fordel for brugerne.

1.3.3 Metodevalg

I dette afsnit beskrives kort, hvilke metoder der benyttes i projektet.

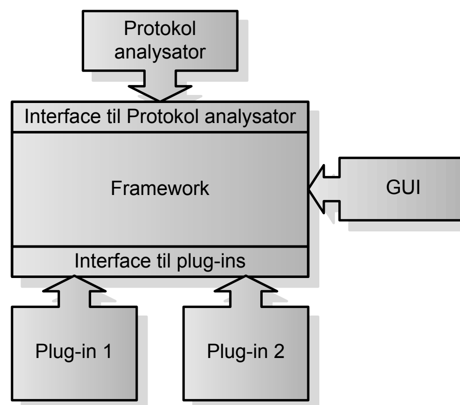
Bruger centreret design Brugerne af det endelige program involveres i analyse fasen, så relevant funktionalitet kommer med i systemet. Denne analyse fører frem til en kravspecifikation, som prototypen skal overholde. Den færdige prototype testes hos brugerne for at se, om den lever op til brugernes krav. Resultaterne fra disse brugertests benyttes til et design af næste prototype, der igen brugertestes. Udviklingen sker altså iterativt, hvor brugerne er med i centrale faser. Derved opnås et system, som overholder brugernes specifikationer. I dette projekt er målet (se afsnit 1.2.5) den første prototype og brugertests kan give ideer til fremtidige ændringer [11].

Test driven development (TDD) Det er fordelagtigt at kombinere ovenstående metode med udviklingsformen TDD. Udviklingen sker på basis af tests, der samlet set repræsenterer kravspecifikationen. Det antages, at systemet opfylder kravspecifikationen, når testene kører uden fejl. Testene kan med fordel genereres som unit tests, der automatiserer testene. Dermed kan de kan køres gentagne gange i udviklingsprocessen for at undersøge, om systemet (stadig) overholder kravspecifikationen [1]. Metoden er velegnet til systemets logik, men mindre velegnet til de grafiske og multitrådede dele.

Design patterns I designet benyttes relevante design patterns, hvor dette er muligt. Der findes design patterns til en række formål, hvoraf en del er relevante for projektet. Ved benyttelse af design patterns opnås en konsistent og fleksibel kode, der bygger på gennemarbejdede designs [3].

1.3.4 Foreløbige designbetragtninger

Figur 1.2 viser en ide til et overordnet design. Den kan ses som en udvidelse af en generel definition af et framework.



Figur 1.2: Ide til overordnet design.

Kernen i systemet er frameworket, der indeholder den fundamentale funktionalitet. En protokolanalysator benytter frameworket gennem et interface (fx ved at extende en abstrakt klasse). Protokolanalysatoren har mulighed for at sætte protokolspecifik information gennem dette interface. Ekstra funktionalitet til frameworket implementeres gennem plug-ins (biblioteker, der hentes dynamisk på kørselstidspunktet), fx vha. .NET's reflektion API. Disse plug-ins har mulighed for at interagere med frameworket gennem et interface, der kan ses som services, som frameworket tilbyder. Det er begrænsningerne i dette interface/disse services, der begrænser mulighederne for et plug-in.

1.3.5 State of the art

Ideen om en protokolanalysator er ikke ny. Men projektet indeholder stadig „uudforsket territorium". I dette afsnit beskrives, hvad der findes af teknologi på området i forvejen, og hvordan denne teknologi kan bruges i projektet. Men det fremgår ligeledes, at denne teknologi ikke kan *erstatte* projektet.

Ethereal er en protokolanalysator, der understøtter en lang række protokoller. Den kan sniffe data fra netværk, der understøttes af WinPcap[5], hvilket overordnet er netværk, der traditionelt benyttes som bærere af IP-protokollen.

FOSS Electric NetSniffer/Emulator er en protokolanalysator, der tidligere er udviklet på FOSS. Den kan benyttes på systemer, der benytter ARCNET protokollen (se afsnit 1.2.1).

[5]Forefindes på `http://www.winpcap.org` – tilgængelig 1/6-2007.

Begge ovenstående teknologier kan benyttes i projektet som inspirationskilde, når det skal besluttes, hvilken funktionalitet, der skal inkluderes i værktøjet.

De kan dog ikke erstatte værktøjet ligesom der heller ikke kan benyttes brudstykker fra dem, selvom kildekoden er tilgængelig for begge. Dette skyldes, at WinPcap ikke er open source, hvorfor man ikke kan benytte interfacet til I^2C netværket i Ethereal. I FOSS Electric NetSniffer/Emulator er understøttelsen af ARCNET protokollen indlejret i alle dele af systemet, så det vil være meget kompliceret at benytte det i dette projekt, når det skal fungere som et generel framework for en protokolanalysator.

Dette er de teknologier, der i natur minder mest om det værktøj, der skal udvikles i projektet. Men som det fremgik, er projektet relevant alligevel, da disse teknologier ikke kan opfylde projektets vision.

Når frameworket udvikles, bliver det tilstræbt at benytte eksisterende teknologi på markedet, hvis det er brugbart i en del af frameworket. Der „opfindes“ ikke en ny udgave af noget, der allerede har vist sig brugbart i praksis. Det bedste eksempel på dette er benyttelsen af eksisterende standarder eller teknologier, der kan indlejres direkte i projektet. Fx kan man benytte XML-standarden til at gemme data i stedet for at bruge et hjemmelavet binært format, som andre programmer ikke kan læse. Man kan ligeledes forestille sig, at hvis der skal benyttes et simpelt script sprog til interaktionen mellem værktøjet og netværket, kan der findes et eksisterende sprog, der opfylder de krav, der stilles til sproget.

De teknologier, der med fordel kan benyttes i projektet, vil blive identificeret, når en egentlig kravspecifikation er udarbejdet. I dette afsnit konstateres det blot, at eksisterende teknologi skal undersøges, hvor det kan indgå naturligt i projektet.

1.4 Om rapporten

Rapportens opbygning afspejler faserne i projektet. Der lægges ud med en foranalyse af problemstillingerne, og der foreslås brugbare løsninger i kapitel 2. Desuden bruges analysen til at give en egentlig kravspecifikation til den første prototype af frameworket. Derefter beskrives designet af systemet i kapitel 3. Dette design har til formål at kunne bruges til implementering af en prototype af frameworket. Men formålet med designet er også at opbygge en fleksibel prototype, da brugertests utvivlsomt vil kræve ændringer i designet. Derfor fungerer kapitlet også som en analyse af, hvordan et sådant fleksibelt design opnås. I kapitel 4 beskrives implementeringen af designet. Desuden beskrives det, hvordan denne implementering testes teknisk. Disse 3 kapitler (kapitel 2-4) beskriver således opbygningen af frameworket i alle detaljer. Frameworket benyttes i kapitel 5, hvor en beskrivelse af I^2C protokolanalysatoren gives. Denne del udgør sammen med frameworket et værktøj til analyse af FOSS' I^2C netværk. I kapitel 6 ses der på fleksibiliteten af frameworket, idet der opbygges en protokolanalysator til CAN bussen. Dette værktøj danner grundlag for brugertests af frameworket. Disse brugertests giver oplysninger om fremtidige ændringer af prototypen. Dette er relevant, da denne bus er relevant for fremtidens apparater hos FOSS. I kapitel 7 præsenteres nogle eksempler på plug-ins til frameworket, og til slut opsamles projektets resultater i kapitel 8.

Rapporten er vedlagt forskellige appendices. Disse benyttes gennem rapporten og vil derfor blive præsenteret, hvor det er relevant.

1.4.1 Om engelske udtryk

Rapporten er skrevet på dansk. Men projektets emneområde har mange engelske udtryk, der enten ikke findes en brugbar oversættelse til, eller hvor en oversættelse virker meningsforstyrrende, da man i emneområdet normalt kun benytter udtrykket på engelsk. I sådanne situationer fastholdes det engelske udtryk i rapporten. Dog kan de engelske ord ses benyttet med „dansk bøjning“. I dette kapitel er der allerede set eksempler på sådanne engelske ord:

Plug-in er et almindelig brugt udtryk om et stykke program, der tilføjer funktionalitet til et allerede eksisterende program. En oversættelse (fx opkobling) giver ikke mening.

Framework Med samme argumentation som ovenfor, giver en oversættelse ikke mening. Til gengæld kan danske bøjninger godt bruges, fx frameworket, men i flertal er frameworks bedre.

Observer Design Pattern er navnet på et Design Pattern, hvorfor navnet holdes på engelsk. Design Pattern lader sig i øvrigt heller ikke oversætte.

Generelt lægges der i rapporten vægt på, at indholdet er intuitivt forståeligt for læseren og ikke nødvendigvis følger alle regler hos Dansk Sprognævn.

1.4.2 Rapportens omfang

Tabel 1.1 viser lidt statistik over rapportens omfang. Dette er uafhængig af rapportens layout og kan derfor bruges til at give en bedre vurdering af omfanget, end antallet af sider giver.

	Indhold	Appendices	I alt
Tegn	208196	16224	224420
Ord	30780	1730	32510
Linier	1728	488	2216
Normal sider	*87*	*7*	*94*
Faktiske sider	90	34	124
Figurer	24	0	24

Tabel 1.1: Statistik over rapportens omfang. Figurer er ikke medregnet i antal normalsider.

1.4.3 Medfølgende CD

Der følger en CD med til rapporten. Indholdet af denne kan også findes elektronisk på `http://kangaroo.ozo.dk/PEP`. CD'en indeholder tre biblioteker: report, installationfiles og sourcecode. Report indeholder denne rapport i pdf og ps format. Installationfiles indeholder en kompileret setup fil til den fremstillede prototype. Desuden indeholder biblioteket programmer, der nødvendige for at køre prototypen. Sourcecode indeholder al produceret kildekode.

En vejledning til brugen af cd'en kan findes ved at åbne `index.html` fra roden af cd'en eller ved at gå ind på den elektroniske udgave af cd'en på den adresse, der er specificeret ovenfor.

Kapitel 2

Indledende analyse

I dette kapitel gives en indledende analyse af projektets hovedproblemstillinger. Hvert afsnit tager fat i én specifik problemstilling og foreslår en løsning til denne. Først undersøges brugerkrav, så der kan udarbejdes en kravspecifikation for projektet. Dernæst undersøges det, hvordan protokoller kan defineres i et generelt format, så frameworket kan bruge det til at genkende netværkspakker. Det valgte format beskrives i detaljer og problemer/ulemper identificeres og løses.

Når det er afklaret, hvordan pakkerne kan håndteres på en generel måde, skal det undersøges, hvordan en interaktion med netværket kan foretages og integreres i frameworket.

Kapitlet omhandler ikke relevante design patterns, da en beskrivelse af de enkelte patterns indgår mere naturligt i kapitlet, der omhandler designet af systemet (kapitel 3).

Indholdsfortegnelse

2.1 Brugerkrav **11**
2.2 Kravspecifikation **15**
2.3 Format af protokolspecifikke beskeder **19**
2.4 Om NetPDL **25**
2.5 Muligheder for interaktion med netværket **31**
2.6 Konklusion på indledende analyse **32**

2.1 Brugerkrav

En kravspecifikation består af to dele: Brugerkrav til slutproduktet samt tekniske krav til produktet. Nogle af de tekniske krav vil opstå som følge af brugerkravene.

I dette afsnit ses der på, hvilke metoder, der bruges til at identificere brugernes krav. Desuden ses der på, hvordan disse metoder bruges i praksis og hvordan, der kan drages konklusioner ud fra den brugte metode.

2.1.1 Identifikation af brugere

Det første trin i indsamlingen af brugerkrav er at finde ud af, hvem der er brugere af systemet. Udover at man finder ud af, hvem der skal indgå i en undersøgelse af brugerkravene, kan oplysninger om brugerne også bidrage med inputs til selve kravene, fx kan oplysninger om brugernes it-niveau fortælle noget om, hvor avanceret programmet kan være, uden at det forvirrer brugeren. Desuden kan forskellige bruger grupper have vidt forskellige krav til det samme program, så spredningen af bruger grupper kan fortælle noget om behovet for fleksibilitet i programmet [11, s. 171-172].

Generelt gælder, at brugerne af værktøjet er medarbejdere hos FOSS, enten i Hillerød, Danmark eller i Höganäs, Sverige. Derudover kan brugerne inddeles i to hoved grupper:

1. Udviklere af applikationer til ESWP. Kan bruge værktøjet, når applikationens kommunikation med netværket skal testes.

2. Udviklere af ESWP. Kan bruge værktøjet i udviklingen og videreudviklingen af ESWP til at teste platformens kommunikation.

Begge grupper består af en mindre gruppe mennesker (10-20 personer).

Denne identifikation giver følgende oplysninger, der kan bruges i mødet med brugerne samt ved fastsættelse af kravspecifikationen:

- Brugerne har et vist teknisk niveau, hvilket reducerer kravene til bruger venlighed, men øger kravene til mulighed for kompleksitet i funktioner.

- De forskellige bruger grupper har brug for informationer på forskellige niveauer i systemet (fx forskellige protokoller), hvilket stiller krav til muligheden for valg af abstraktionsniveau i systemet.

2.1.2 Metoder

Der findes en række metoder til at indsamle brugerkrav. Nedenfor ses de mest populære metoder og hvorfor eller hvorfor ikke, de er relevante i denne sammenhæng.

Spørgeskemaer Et større antal brugere udfylder et generelt spørgeskema. Metoden er billig og giver adgang til udetaljerede brugerkrav. Statistiske metoder kan benyttes til at analysere de kvantitative resultater. Denne metode er ikke relevant i denne sammenhæng, da antallet af brugere er for lille til at de statistiske resultater er signifikante. Desuden kan metoden kun benyttes i en meget indledende kravsanalyse, da der hurtigt bliver brug for mere detaljerede oplysninger.

Interviews Brugere interviews en ad gangen, hvor spørgsmålene er tilrettelagt, så brugeren får mulighed for at fremkomme selvstændigt med ideer, dvs. spørgsmålene må ikke være ledende og konklusionerne skal så vidt muligt komme fra brugeren. Spørgsmålene kan tilrettelægges gennem tre typer interviews [11, s. 392-396]:

- Struktureret interview, hvor spørgsmålene er defineret på forhånd og ikke afviges undervejs.

- Semi-struktureret interview, hvor der er forberedt spørgsmål på forhånd, der holder den røde tråd, men undervejs stilles uddybende spørgsmål, hvor det er relevant.
- Ustruktureret interview, hvor man mødes uden megen forudgående forberedelse og spørgsmålene kommer, som de falder ind. Der er snarere tale om en uformel samtale end et egentligt interview.

Metoden er relevant for projektet, da det er en forholdsvis billig måde at få detaljerede oplysninger. Der skal dog laves interviews med flere brugere, så alle brugergrupper er repræsenteret.

Fokus grupper Samling af brugere, der diskuterer kravene. Metoden eliminerer nogle af ulemperne ved interviews, da brugerne kan diskutere synspunkter og resultaterne bliver derfor mere kvalitative. Gruppen af brugere vælges som et repræsentativt udsnit af den samlede brugergruppe. Til gengæld er det en dyr løsning, da det både involvere mange personer og kan kræve en del tid. Af denne sidste grund er det ikke muligt at benytte metoden i projektet, da en fokusgruppe udgør så stor en del af den samlede brugergruppe, at tiden til det ikke kan afsættes. Metoden ville ellers være fordelagtig i projektet, da den sandsynligvis kunne føre til en komplet kravspecifikation.

Naturlige observationer Man ser på brugere, der bruger et lignende værktøj i forvejen og drager konklusioner ud fra det. Dette giver en uformel tilgang til analysen og brugeren er i sit naturlige miljø. Metoden er god, hvis man videreudvikler et eksisterende program. I dette projekt kunne man se på brugen af NetSniffer/Emulator i dagligdagen. Men da programmet benyttes i begrænset omfang og altså ikke i den generelle hverdag, er dette ikke muligt. Til gengæld kan man i undersøgelser tage udgangspunkt i brugen af dette program, da det effektiviserer søgningen betydeligt.

Dokumentation undersøgelser Man kigger i dokumentation til nuværende anvendte produkter og analyserer den eksisterende funktionalitet for at finde punkter, hvor der er mulighed for forbedringer. Man kigger også i dokumentation til produkter, som skal benyttes i forbindelse med det, man udvikler, så man kan se, hvad disse produkter stiller af krav til sit produkt. Metoden er billig, da den ikke involverer brugere. Men man mister derved også meget subjektiv information, hvorfor denne metode sjældent er tilstrækkelig. I projektet kan denne metode bruges til at få oplysninger om de teknologier, der skal benyttes af værktøjet, fx en specifikation af I^2C protokollen.

Metoderne kan udføres på forskellig vis. Nogle er pr. definition uformelle i opbygningen (naturlige observationer), mens andre er formelle (fokus grupper). Interviews kan være begge dele, idet et interview fx kan holdes i et mødelokale med en formel dagsorden (spørgsmål) eller uformelt over frokosten. Begge fremgangsmåder har fordele og ulemper, da formelle omgivelser kan lægge begrænsninger på brugeren, der fx kan være nervøs eller genert, mens uformelle omgivelser kan give for løse og ustrukturede resultater. Det er et aspekt, der skal overvejes, når en metodes omgivelser vælges [11, s. 214].

I ovenstående ses det, at interviews er en god fremgangsmåde i projektet, da det giver billig specifik information om kravene. Det er vigtigt, at der afholdes interviews med begge identificerede brugergrupper (afsnit 2.1.1). Jeg er tilknyttet

gruppe 2 hos FOSS, så interviews med denne gruppe udføres naturligt i et uformelt miljø med løbende spørgsmål, mens interviews med gruppe 1 udføres bedst i en formel sammenhæng, da der i forvejen skal indgås en formel aftale om interviewet.

Undersøgelse af eksisterende dokumentation kan også benyttes som fremgangsmåde, idet FOSS Electric NetSniffer/Emulator, kan benyttes ved identificering af krav. Det er to selvstændige programmer: NetSniffer benyttes til at få oplysninger om nettrafikken, mens Emulator benyttes til at skabe simpel kommunikation (se endvidere afsnit 1.3.5). Disse programmer benyttes derfor som udgangspunkt i interviewene, hvis den interviewede er bekendt med programmerne.

Desuden undersøges dokumentationen til ESWP platformen, da denne fortæller noget om den sammenhæng, som værktøjet skal bruges i.

2.1.3 Fremgangsmåde i formelle interviews

I et interview er det som nævnt ovenfor vigtigt at lade brugeren fremkomme med så mange selvstændige ideer og forslag som muligt. Dette gøres gennem klare, korte og ikke-ledende spørgsmål. Desuden følges følgende punkter i interviewet:

1. Introduktion. Deltagerne præsenteres. Emne for interview præsenteres og specielle hensyn diskuteres.

2. Opvarmningsspørgsmål. Dvs. neutrale spørgsmål, der kan give noget baggrundsinformation om den interviewede.

3. Hoved del. De egentlige spørgsmål præsenteres i stigende sværhedsgrad/detaljeringsgrad.

4. Afsluttende spørgsmål. Spørgsmålene koncentreres om at få klarlagt diffuse detaljer og andre småting.

5. Afslutning. Det klarlægges, at alt er forstået korrekt.

På denne måde ledes den interviewede gennem de ønskede spørgsmål uden at presses for hårdt [11, s. 390-391].

I de konkrete interviews er det som nævnt oplagt at tage udgangspunkt i NetSniffer/Emulator. Ud fra udsagn i interviewene kan „genbrugelige“ funktioner fra NetSniffer/Emulator identificeres. Desuden kan udsagnene benyttes til en gruppering af funktionerne i følgende kategorier:

Skal have (Must have) En funktion, der er helt central i en protokolanalysator. Det er derfor et utvetydigt krav (fra brugeren), at funktionen er eksisterende i det endelige program.

Bør have (Should have) En funktion, som er særdeles brugbar i programmet, men programmet kan dog benyttes uden. Funktionen giver et løft til programmet, så det skal tilstræbes, at funktionen eksisterer i det endelige program.

Kan have (Nice to have) En funktion, der er mindre vigtig for programmet, idet brugeren måske knap vil opdage, at funktionen mangler. Men funktionen kan fx eliminere et mindre irritationsmoment eller give et mindre løft til brugeroplevelsen. Funktionen skal implementeres, hvis der er ressourcer til det – ellers er det en mulig fremtidig udvidelse.

Udover at identificere genbrugelige funktioner i NetSniffer/Emulator, er det ligeså brugbart at få identificeret funktioner, der *ikke* er brugbare og derfor skal udelades. Det drejer sig fx om funktioner i NetSniffer/Emulator, der virkede brugbare i designøjeblikket, men som i brug har vist sig aldrig at blive brugt. Man kan se på denne gruppe af funktioner som en fjerde kategori i ovenstående beskrivelse („Har ikke“ – „Does not have“). Desuden kan det blandt de identificerede funktioner i NetSniffer/Emulator diskuteres, hvorvidt de er tilstrækkelige i deres nuværende form, og en given udvidelse af funktionen kan kategoriseres som ovenfor.

Endelig kan udsagnene fra interviewene give anledning til uddybende spørgsmål, der i sidste ende kan føre til funktionalitet, der ikke eksisterer i NetSniffer/Emulator. Sådan funktionalitet kan også identificeres ud fra mere eller mindre ledende spørgsmål, fx ud fra forudgående ideer til ny funktionalitet eller ud fra identificerede irritationsmomenter i NetSniffer/Emulator. Interviewene gennemføres derfor som semi-strukturede interviews.

2.1.4 Udførte interviews

I afsnit 2.1.1 blev to brugergrupper identificerede. Resultaterne af de formelt udførte interviews med denne gruppe er opsummeret i appendiks A.1. Uformelle interviews er udført med gruppe 2 og resultaterne er opsummeret i appendix A.2. Der er tale om opsummeringer, da det dels fokuserer på det relevante fra interviewene og dels, fordi interviewene ikke blev optaget (lyd/billede).

Disse resultater giver anledning til en egentlig kravspecifikation, der supplerer specifikationerne fra afsnit 1.2.4 og 1.2.5. En opfyldelse af kravspecifikationen dækker således de behov, FOSS har (motivationen for projektet) samt de krav, de individuelle brugere har til værktøjet.

De fundne krav er hovedsageligt funktionelle krav. Kravspecifikationen, der gives i næste afsnit og formaliserer de fundne krav, suppleres med ikke-funktionelle krav i projektet. Tilsammen udgør disse en komplet kravspecifikation for projektet.

2.2 Kravspecifikation

Ovenstående redegørelse af teorien bag brugerundersøgelser og de udførte interviews giver anledning til en egentlig kravsspecifikation, der er mere detaljeret end de ønsker, FOSS er fremkommet med, afsnit 1.2.4. Kravene følger i de kommende afsnit grupperet efter, om de skal være en del frameworket, protokolanalysatoren eller plug-ins til frameworket. Denne gruppering bestemmes ud fra prioriteringen af kravet, og om kravet er protokolspecifikt. Med prioritering refereres til de introducerede kategorier i afsnit 2.1.3 (must have/should have/nice to have/does not have). Afsnittene koncentrerer sig om de funktionelle krav til de enkelte dele, men medtager også relevante ikke-funktionelle dele – de resterende ikke-funktionelle krav vil fremgå i designet og analysen af de enkelte dele i systemet[1].

[1] [11, s. 205-209] og [7, s. 57] benytter samme definition på, hvordan funktionelle krav og ikke-funktionelle krav opdeles. I denne rapport benyttes denne definition ligeledes – dog kategoriseres ikke-funktionelle krav ikke yderligere.

2.2.1 Frameworket

I frameworket skal der overordnet være to funktioner: Sniffing og interaktion. Sniffingen kan startes og stoppes. Beskeder opdages gennem protokolanalysatoren, og præsenteres individuelt for brugeren i en liste. Denne liste vokser efterhånden, som beskederne opdages. I oversigten over beskederne vises oplysninger, som er fælles for alle typer af beskeder, se tabel 2.1. Et eller flere felter kan være tomme afhængig af beskeden og protokollen.

Feltnavn	Betydning
Id	Unikt id på en pakke
Time	Tidspunkt for opdagelse
Protocol	Besked type eller protokol

Tabel 2.1: Generelle oplysninger om beskeder.

Frameworket skal give protokolanalysatoren mulighed for at udbygge denne liste med protokolspecifikke oplysninger.

Ved at klikke på de enkelte beskeder kan man få detaljerede oplysninger om beskeden. Disse oplysninger er protokolspecifikke. Desuden vises beskedens rå data i hexadecimal notation.

Der kan tilføjes filtre til listen af beskeder, så man kun får vist beskeder, man helt specifikt er interesseret i. Det er frameworket, der understøtter den funktionalitet, dvs. muligheden for at oprette filtre og påføre dem listen af beskeder. Men filtrene genereres vha. data fra protokolanalysatoren, da filtrene skal kunne defineres vha. protokolspecifik information. Fleksibiliteten af filtrene giver mulighed for at lette læsevenligheden af en stor mængde beskeder, idet irrelevante beskeder kan ignoreres. I brugerundersøgelserne viste det sig, at al funktionalitet, der kan lette overskueligheden af beskederne er „nice to have“. Den beskrevne filtrering er en del af dette, men man kan forestille sig andre ideer, der gør dette aspekt endnu bedre. Dette er en mulighed til et fremtidigt plug-in, se afsnit 2.2.3.

Der lægges i øvrigt vægt på, at defineringen af filtrene er så simpel som mulig for brugeren.

En „optagelse“, dvs. en liste af beskeder, der er modtaget fra netværket, skal kunne gemmes i en fil, så den senere kan hentes frem i programmet. Formatet af denne fil kan være specifik for værktøjet, men i så fald skal der være mulighed for at eksportere optagelsen til et velkendt format, fx csv eller xml. Filen kan også gemmes i et sådant format direkte.

Filhåndteringen i programmet skal være intuitiv for brugeren, fx kan fremgangsmåder fra andre kendte programmer benyttes (fx måden, der benyttes i Microsoft Office). Dette krav viste sig i brugerundersøgelsen på baggrund af den uhensigtsmæssige håndtering i Emulator. Det er ikke et krav, at der kan være flere filer åbne på én gang, men det kan være rart at have denne mulighed for at sammenligne filer.

Interaktionen med netværket skal foregå vha. et egentlig computersprog. Men det er vigtigt, at det valgte sprog giver mulighed for, at det er muligt at lave fra meget simpel interaktion til forholdsvis kompleks interaktion. Dette gælder, da interaktionen i 80-90%[2] af tilfældene bruges med få kommandoer, mens de resterende 10-20%[2] er mere komplekse, fx hvor felter i et svar skal bruges i den videre

[2]Estimeret ud fra interviewene med brugerne.

interaktion. Med andre ord: Der er brug for begge muligheder. Dette kan fx opnås vha. et eksisterende (script) sprog, og Ruby er foreslået som mulighed, da dette i forvejen benyttes hos FOSS. For ikke tekniske brugere kan programmet tilbyde en let adgang til templates, der udfører simple og ofte benyttede procedurer.

Det skal være muligt at få adgang til svar til sendte beskeder, så disse oplysninger kan bruges i det efterfølgende program på kørselstidspunktet, dvs. vha. variabler e.l. Når en besked skal sendes på netværket, kan dataene i beskeden komme fra en fil eller fra en konstant i programmet.

Programmet kan indeholde en editor til at redigere og køre programmer i. Men vælges et allerede eksisterende sprog, kan programmet nøjes med mulighed for afvikling af programmer. Denne sidste funktion kan dog ikke undværes, da funktionen er særdeles anvendelig i sammenhæng med sniffingen, fx kan sniffingen starte og stoppe sammen med programmet.

Filhåndtering er også i interaktionsdelen af frameworket nyttig. Programmer skal kunne hentes og gemmes. Formatet afhænger af det valgte sprog, men umiddelbart kan det være et almindeligt tekstformat, hvis der er tale om et fortolket sprog.

Et vigtigt ikke-funktionelt krav til frameworket (og til værktøjet som helhed) er, at det skal være brugervenligt i sin opbygning, så et minimum af support er nødvendig, dvs. indlæringskurven skal være stejl[3]. Dette aspekt kan verificeres vha. brugertests på prototyper af værktøjet.

Der stilles desuden nogle krav til frameworket for at nedenstående krav til protokolanalysatoren og plugins kan opfyldes. Dette er krav til de interfaces, der er mellem frameworket og hhv. protokolanalysator og plugins (se figur 1.2 s. 7). Disse vil fremgå af de følgende afsnit.

2.2.2 Protokolanalysator

Protokolanalysatoren skal sørge for alt det, der mangler i frameworket, for at man har en funktionel protokolanalysator. I praksis betyder det, at protokolanalysatoren skal sørge for alt, hvad der er specifikt for den benyttede protokol. Den skal definere netværkets kommunikation for frameworket, dvs. protokolstakken og hvordan de enkelte beskeder i de enkelte protokollag ser ud. Derudover skal protokolanalysatoren kommunikere med det pågældende netværk og stille opdaget data til rådighed for frameworket. Frameworket kan så sammenligne dataene med den definerede protokol for at identificere beskederne.

Protokolanalysatoren skal ikke kun definere mønstrene i dataene. Den skal også give en fortolkning af de enkelte dele i en besked, så frameworket kan præsentere detaljerede oplysninger om beskeden.

Det er et ikke-funktionelt krav, at specifikationen skal være simpel at udarbejde og ændre, fx for at udvide med ny understøttet funktionalitet i en ny version af protokollen.

Dette er den store opgave for protokolanalysatoren. Derudover er der mindre opgaver som at definere yderligere kolonner i pakkelisten og at beskrive, hvilke data fra pakken, der skal stå i disse kolonner. Når designet af frameworket udfærdiges vil der utvivlsomt komme flere krav til protokolanalysatoren, men for at overholde kravspecifikationen skal disse opgaver holdes til et minimum.

[3] Med „stejl indlæringskurve“ menes her en kurve i et (tid, indlæring) koordinatsystem med stor hældningsværdi.

Dette er de dele, der er essentielle for at have en protokolanalysator. Ovenstående er altså det minimum af arbejde, der skal til for at bruge frameworket til at lave en protokolanalysator. Men det betyder ikke, at det ikke skal være muligt at lave et bedre værktøj for brugeren. Man kan fx have mulighed for at definere filtre til frameworket, der er specifikke for protokollen. Dette sparer brugeren for at skulle lave dem.

I^{2}C protokolanalysator

I projektet skal brugen af frameworket vises ved at lave en I^{2}C protokolanalysator. Denne skal overholde ovenstående krav. Det betyder, at stakken skal defineres, dvs. protokolinformation om hhv. I^{2}C, Hextet og T123 skal defineres, da det er disse protokoller, der benyttes hos FOSS. De enkelte dele af disse protokoller skal desuden defineres, så de kan vises i de detaljerede oplysninger i frameworket. Der skal ligeledes defineres filtre, der er relevante for FOSS' produkter, fx ofte brugte værdier i T123 beskeder e.l.

2.2.3 Plug-ins

Ved opfyldelse af ovenstående har man en funktionel protokolanalysator til en given protokol. Men det betyder ikke, at alle de fundne krav er opfyldt. De resterende krav har dog vist sig at have lavere prioritet hos brugeren, hvorfor de ikke implementeres i selve frameworket. Men der skal være mulighed for at udvide frameworket på et senere tidspunkt vha. plugins. For at opfylde de resterende af brugerens krav kunne følgende plug-ins overvejes:

- Mulighed for automatisk sammenligning af filer (optagelser).
- Understøttelse af udklipsholderen.
- Mulighed for datadump til fil af en markeret del af den rå data (så det fx kan analyseres nærmere i andre programmer).
- Eksport til nye formater (andre filtyper).
- Syntaksmarkering i en editor i interaktionen.
- Hjælp til læseligheden i beskedoversigten fx med farve(r) og tekst.

Listen kunne fortsættes, idet et sådant værktøj kan udvides på et utal af måder.

Der stilles ingen krav om, at der skal implementeres specifikke plugins, idet den krævede funktionalitet allerede er en del af frameworket. Men muligheden for udvidelse vha. plugins stiller krav til frameworket. Disse krav vil der nu blive set nærmere på.

De enkelte plugins får adgang til frameworket via services, frameworket stiller til rådighed. Det er fleksibiliteten i disse services, der begrænser mulighederne for plugins. Derfor handler dette afsnit om kravene til disse services. Frameworket skal give adgang til:

- Listen af beskeder.

- Det interface, der også er tilgængeligt for brugeren gennem brugerinterfacet. Dette giver et plugin samme muligheder som en bruger. Et plugin, der gør brug af dette interface kan opfattes som en „indspillet makro“[4].
- Værktøjslinier, så der kan tilføjes knapper og lignende.
- Markeret tekst.
- Program i interaktion editoren.
- Filtre.
- Oplysninger om hændelser i frameworket, som plug-ins kan reagere på.

Med disse muligheder er frameworket fleksibelt og plugins har rig mulighed for at forbedre frameworket.

2.3 Format af protokolspecifikke beskeder

I gennemgangen af kravene til værktøjet, dels ønskerne fra FOSS og dels de funktionelle krav fundet ved brugerundersøgelser, har det vist sig, at der skal opbygges et protokoluafhængigt framework. Alligevel skal det være i stand til at genkende protokolspecifikke beskeder, så frameworket må nødvendigvis tilknyttes en beskrivelse af de protokoller, der skal genkendes. Problemet er, hvordan en sådan beskrivelse kan gives mest hensigtsmæssigt. Dette aspekt ses der nærmere på i dette afsnit.

2.3.1 Krav til formatet

Inden der foreslås en løsning til problemet, er det vigtigt at se på, hvilke krav, der er til formatet, så den rigtige løsning vælges. Der er to grupper af krav, der skal overvejes:

1. Frameworkets krav til formatet.
2. Protokolanalysatorens krav til formatet, dvs. krav fra brugeren af frameworket.

Frameworkets krav kommer af, at formatet skal være brugbart for frameworket, dvs. formatet skal kunne benyttes på datastrømmen, så protokolmønstre genkendes. I denne forbindelse er der følgende overordnede krav:

- Effektivitet. Brugeren skal have oplevelsen af, at han ser protokolbeskederne i real-time.
- Pålidelighed. Hvis der er en besked på netværket, der opfylder en specificeret protokol, skal frameworket kunne bruge protokolspecifikationen, så beskeden opdages. Dette betyder bl.a., at protokolspecifikationen skal være entydig.
- Kompabilitet. Frameworkets værdi øges signifikant, hvis der benyttes et generelt protokolformat, idet frameworket kan benytte protokolbeskrivelser lavet til andre applikationer.

[4] Betegnelsen stammer fra Microsoft Office, hvor en makro kan indspilles ved at klikke rundt i brugerinterfacet.

I afsnit 2.2 blev det fundet, at implementeringen af en protokolanalysator skal være en simpel opgave. Desuden skal specifikationen af en protokol være simpel at ændre. Det stiller nogle krav til den måde, hvorpå formatet defineres af protokolanalysatoren:

- Intuition. En protokol skal kunne defineres på en for brugeren intuitiv måde. En bruger med et indgående kendskab til protokollen skal hurtigt kunne oversætte den teoretiske protokoldefinition til en praktisk specifikation, der er forståelig for frameworket.
- Fleksibilitet. Specifikation skal kunne ændres dynamisk, dvs. små ændringer i protokollen (fx i en ny version) skal hurtigt kunne realiseres i værktøjet.
- Anvendelighed. Om muligt skal alle protokoller kunne defineres i det valgte format.

Disse identificerede krav til formatet, hvori en besked defineres, fører til, at konkrete løsningsforslag kan fremføres.

2.3.2 Overordnet løsning

Overordnet set kan problemet løses ved at give en specifikation af protokollerne vha. en af to metoder:

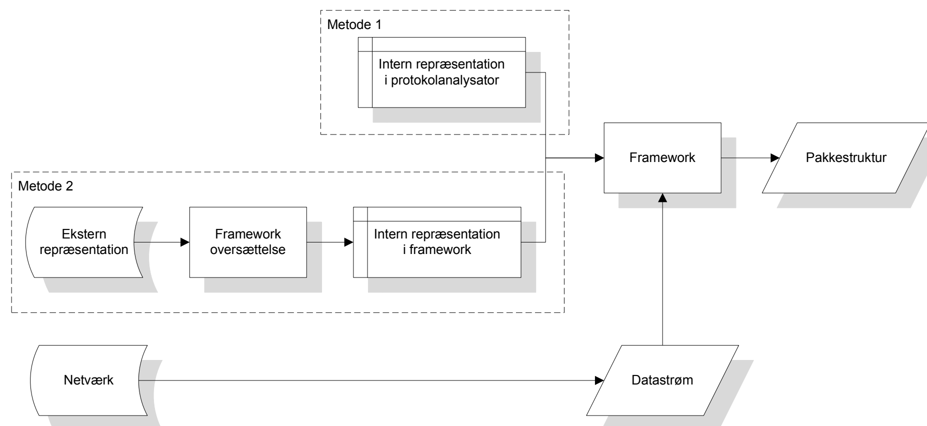
1. En indlejring af specifikationen i protokolanalysatorens kode. Frameworket analyserer pakkerne på netværket gennem interfaces til denne kode.
2. Placering af specifikationen foreligger i en selvstændig ekstern datafil, som frameworket oversætter til en intern repræsentation. Denne interne repræsentation kan sammenlignes med ovenstående indlejrede kode – bortset fra, at den ligger i frameworket i stedet for i protokolanalysatoren. Den interne kode benyttes ved tolkning af pakker på netværket, så de enkelte protokoller i pakken genkendes.

Metoderne illustreres i et rutediagram på figur 2.1. Figuren viser desuden, hvordan frameworket benytter formatet på den rå datastrøm, som fås fra protokolanalysatoren. Figuren viser, der er et ekstra skridt i processen i frameworket, når metode 2 benyttes.

Nedenfor identificeres henholdsvis fordele og ulemper ved de to metoder i forhold til de stillede krav. På baggrund af disse identifikationer vælges en af de to foreslåede metoder.

Metode 1 er den benyttede metode i de mest udbredte protokolanalysatorer i dag (fx Ethereal – se afsnit 1.3.5). Fordelene ved denne metode er, at protokolanalysatoren får en direkte kontrol over tolkningen af datastrømmen. Og der er ikke brug for en intern oversættelse fra en ekstern specifikation til en intern specifikation, så arbejdet med at konstruere frameworket reduceres. Metoden kan desuden anvendes på alle tænkelige protokoller, da al datahåndtering styres internt. Og det må ligeledes formodes, at denne metode opfylder frameworkets krav til formatet mht. effektivitet, da en direkte implementering i programmeringssproget normalt er det mest effektive.

Ulempen ved metoden er, at selv små ændringer i protokollen kræver en rekompilering af den samlede kode. Dog kunne en protokolanalysator lave sin egen



Figur 2.1: Rutediagram over overordnede metoder til protokolformatet.

eksterne repræsentation, der kunne ændres dynamisk (uden rekompilering), som læses internt i protokolanalysatoren og oversættes til det interface, frameworket forventer. Men det flytter bare det ekstra led ud i protokolanalysatoren, se figur 2.1. Og denne ekstra funktionalitet kræver ekstra implementering i protokolanalysatoren, hvilket strider mod de stillede krav til denne, se afsnit 2.2.2. Det kan diskuteres, hvorvidt metoden opfylder kravet om, at specifikationen kan defineres intuitivt. Det er meget afhængigt af den specifikke bruger, om det er intuitivt at specificere en protokol vha. et programmeringssprog (C#), men generelt kan det antages, at en bruger foretrækker et for mennesker letlæseligt format, hvorfor dette er en ulempe for metoden.

Fordelene ved metode 2 er, at den kompilerede kode ikke afhænger af specifikationen, så denne kan ændres uden at det kræver en rekompilering. Desuden opfylder metoden brugerkravene fuldstændigt, hvis der vælges et intuitivt og fleksibelt format af den eksterne datafil, der udgør specifikationen. Så dette skal være med i overvejelserne i forbindelse med valget af dette format. Fleksibiliteten i valget er sikret, idet en ekstern datafil let kan udvides/ændres/omskrives. Frameworkets krav kan også opfyldes vha. denne metode, idet pålideligheden og effektiviteten kan opnås i implementeringen af frameworket. Dog skal formatet af den eksterne datafil vælges, så dette er muligt. Desuden kan det sidste krav fra frameworket om kompabilitet opnås ved at vælge et generelt kendt format.

Ulempen ved metode 2 er, at der kræves mere arbejde ved implementering af frameworket.

Metodernes karakteristika i forhold til kravene er summeret i tabel 2.2.

Spørgsmålstegnene betyder, at det ikke umiddelbart kan afgøres, om metoden opfylder et givent krav. Dette skyldes, at metode 1 afhænger af protokolanalysatorens implementering, mens metode 2 afhænger af, hvordan det eksterne filformat vælges. Men det betyder, at metode 2 opfylder kravene, hvis der kan findes et passende eksternt format til specifikationen, mens man aldrig kan være sikker på, at metode 1 opfylder kravene, da frameworket ikke har kontrol over, hvordan protokolanalysatoren implementerer en protokolspecifikation. Hvis kravene skal være opfyldt, uanset hvordan protokolanalysatoren er implementeret, må metode 2 vælges. Denne metode opfylder også kravene bedst, idet de sidste manglende krav

Krav \ Metode	Intern (1)	Ekstern (2)
Effektivitet	Høj	Mellem
Pålidelighed	?	?
Kompabilitet	Lav	?
Intuition	Lav	?
Fleksibilitet	Lav	Høj
Anvendelighed	Høj	?

Tabel 2.2: Karakteristika ved identificerede overordnede løsninger til protokolformat problemet.

forsøges opfyldt vha. det valgte eksterne format (spørgsmålstegnene). Det skal derfor undersøges, om der findes et format, der kan udfylde de manglende felter i tabel 2.2 med „Høj". Et sådant format identificeres i det følgende.

2.3.3 Mulige eksterne formater

For at opfylde kravene til formatet fra afsnit 2.3.1 vælger vi metode 2 præsenteret ovenfor, jf. diskussionen. Men dette metodevalg sikrer ikke, at alle kravene opfyldes. I valget af det eksterne format står vi tilbage med følgende krav:

- Pålidelighed. Sikres ved at formatet er entydigt defineret, dvs. at én datastrøm ikke kan tolkes på flere måder vha. det definerede format.
- Kompabilitet. Sikres ved at vælge en kendt teknologi til specifikationen, hvilket betyder, at der ikke egenhændigt udvikles et format.
- Intuition. Et individuelt aspekt, men det kan sikres ved at vælge en udbredt teknologi, så det kan formodes, at brugeren er bekendt med teknologien, og den derfor er intuitiv at bruge.
- Anvendelighed. Sikres ved at undersøge fleksibiliteten af det valgte format. Det kan være svært at bevise, at et format kan understøtte alle fremtidige spidsfindige protokoller, men det kan vha. konkrete nøje udvalgte eksempler sandsynliggøres, at formatet er fleksibelt.

I de følgende afsnit beskrives forskellige forslag til en løsning. Forslagene bygger på, at man kan opfatte en specifik protokol som et sprog [6], dvs. en (muligvis uendelig) mængde af tekststrenge, der tilsammen udgør sproget. Et sprog kan defineres vha. forskellige metoder. Når man anser en protokol for at være et sprog, benyttes en metode til at specificere formatet af en protokol, dvs. ud fra formatet kan det afgøres, om en given tekststreng er en lovlig del af sproget (protokollen).

Udvidede regulære udtryk

En mulig løsning er at benytte regulære udtryk, der er en udbredt teknologi. Man opfatter datastrømmen fra netværket som en tekststreng og påfører tekststrengen specifikationen (det regulære udtryk). Et match betyder, at datastrømmen entydigt tilhører den pågældende protokol og derfor skal tolkes som denne protokol, [6, s. 83-120].

Fordelen ved regulære udtryk kan løse problemet med at genkende et mønster i datastrømmen, og de såkaldte „capturing groups“ kan benyttes til at identificere specifikke dele i datastrømmen, fx specifikke felter i protokollen, så frameworket kan vise indholdet af det pågældende felt. Opgaven til brugeren af frameworket bliver vha. regulære udtryk at definere protokollen, som han ønsker at benytte frameworket på samt at definere betydningen af de enkelte capturing groups. Derved har frameworket tilstrækkelig information til at identificere protokollen.

Der er en ulempe ved regulære udtryk, idet de benyttes på tekststrenge, dvs. til genkendelse af et tegnmønster i tekststrengen. Derfor er det nødvendigt at udvide den benyttede syntaks, så genkendelse af bit og bytes understøttes. Dette kunne gøres ved at definere en „kontrol sekvens“[5], som brugeren kunne benytte, når en byte eller bit var forventet. Kontrol sekvensen skal give mulighed for dels at definere, at der nu forventes en arbitrær byte/bit og dels, at der forventes en byte/bit med en specifik værdi (eller værdi interval).

En anden ulempe ved regulære udtryk er, at ikke alle sprog kan defineres vha. regulære udtryk. Et eksempel på et sprog, der ikke kan beskrives vha. disse, er sproget der defineres ved „tekststrenge der består af n nuller efterfulgt af n 1-taller“ eller matematisk 0^n1^n, så fx 0011 og 00001111 tilhører sproget [6, 128-129]. Dette betyder, at der kan være protokoller, der ikke kan defineres vha. regulære udtryk. Kravet om anvendelighed er altså ikke opfyldt. Desuden er regulære udtryk ikke fleksible mht. afhængigheder på tværs af tekststrengen, som i det tilfælde hvor en del af protokollen afgør, hvordan en senere del skal opfattes. I sådanne tilfælde vil kompleksiteten af det regulære udtryk være propertionalt med antallet af permutationer af sådanne betingelser. Men ikke kun i dette tilfælde bliver regulære udtryk komplekse. Generelt bliver de hurtigt lange og uoverskuelige. Derfor er kravet om intuition ikke opfyldt, selvom det er en udbredt teknologi[6].

Til gengæld er formatet både pålideligt og kompatibelt (hvis der findes andre specifikationer af protokoller som regulære udtryk og der ses bort fra de nødvendige udvidelser til syntaksen).

Med denne løsning kan vi altså ikke få alle vores krav fuldstændig opfyldt, omend det er en mulig løsning.

Context-free grammars

Context-free grammars (CFG) udvider den mængde af sprog, der kan defineres vha. regulære udtryk. Et sprog defineres vha. „Terminals“ og „Non-Terminals“. Non-Terminals indeholder information om deres definition i form af en sekvens af Terminals, evt. blandet med Non-Terminals. Informationen kan altså være rekursiv, idet definitionen kan indeholde Non-Terminalen selv. I så fald skal der være flere mulige definitioner af Non-Terminalen, hvoraf mindst én ikke er rekursiv. Terminals er mindste enheder som fx konkrete tegn. Et sprog er defineret som én Non-Terminal (start symbolet), som kan konstrueres ud fra Terminals og Non-Terminals. Om en tekststreng tilhører et sprog beskrevet vha. en CFG, kan afgøres ved at undersøge, om tekststrengen kan konstrueres ud fra start symbolet[6, s.169-214].

Det føromtalte sprog, der ikke kan beskrives vha. regulære udtryk, kan vha. en

[5] En reserveret sekvens af tegn, der fortolkes specielt.

[6] Dette skyldes, at regulære udtryk normalt benyttes i meget mindre målestok, som fx i tekstbehandlingsprogrammer til søgning.

CFG beskrives ved:

$$S \rightarrow \mathbf{0}S\mathbf{1}|\epsilon$$

Her er S den (eneste) Non-Terminal, der definerer sproget. | separerer forskellige mulige definitioner af S og ϵ specificerer en tom streng (det betyder, at n i ovenstående definition af sproget kan være 0). Sproget består af 3 Terminals: 0, 1 og ϵ. Eksemplet viser, at det er muligt at definere sprog med CFGs, som ikke kan defineres med regulære udtryk. Det kan i øvrigt bevises, at den mængde sprog, der kan beskrives vha. regulære udtryk, er en ægte delmængde af den mængde sprog, der kan beskrives vha. CFGs [6, s. 247-251].

Fordelen ved CFGs er, at en tekststreng (ligesom ved regulære udtryk) på en simpel måde kan parses for at undersøge, om det følger syntaksen i det af CFG'en generede sprog. En CFG kunne benyttes til at specificere en protokol, når protokollen ses som et sprog. Der findes endda værktøjer, der ud fra en CFG kan generere en parser til det af CFG'en generede sprog, fx YACC[7].

Generelt besidder en CFG specifikation af protokollen samme fordele som regulære udtryk. Dog har CFG yderligere fordele. Anvendeligheden er større, idet der kan udtrykkes flere sprog vha. en CFG end ved regulære udtryk. Kompabiliteten er større, da teknologien er kendt og udbredt, og hjemmelavede udvidelser er ikke nødvendige (bit og bytes kunne defineres som Non-Terminals). Desuden findes der allerede værktøjer, der kan bruges. Intuitionen hos brugeren er bedre, da en CFG ikke bliver kompleks i samme grad som regulære udtryk. Men definitionerne af de enkelte Non-Terminals kan dog godt være forholdsvis komplekse.

En CFG er altså en mulig løsning som opfylder hovedparten af kravene.

NetPDL

Et eksempel på en CFG er XML, der i dag er en særdeles udbredt teknologi. Et oplagt spørgsmål er derfor, om man kan benytte XML til at definere protokollerne? Der findes en teknologi, der benytter denne fremgangsmåde. Teknologien hedder NetPDL. Idet XML er en CFG, besidder NetPDL fordelene fra CFG (og dermed også fordelene ved regulære udtryk). Vi opnår fx samme pålidelighed som ved en CFG.

I designet af NetPDL har hovedmålet været enkelthed forstået på den måde, at det skal være enkelt og intuitivt for brugeren at beskrive en protokol i sproget [12]. Netop nogle af de krav, der stilles til valget af dataformatet.

NetPDL giver således vha. XML sproget en standard måde at definere en protokol på, hvilket sikrer kompabiliteten. Kompabilitet er altid afhængig af, om andre applikationer også benytter standarden[8]. Men NetPDL er udviklet specifikt til dette formål og er p.t. eneste forslag på markedet til en standardisering af specifikation af et protokolformat.

Tilbage er kun spørgsmålet om anvendelighed, hvilket afhænger af fleksibiliteten af mulighederne i NetPDL. NetPDL er designet med målet om at kunne specificere alle protokoller [12, s. 689]. Man må derfor forvente en stor grad af fleksibilitet i syntaksen, men det kræver en nærmere analyse af NetPDL.

[7]Yet Another Compiler-Compiler: se `http://dinosaur.compilertools.net/` – tilgængelig 1/6-2007.

[8]Som før nævnt benytter eksisterende protokolværktøjer fx generelt en intern repræsentation, hvorfor disse specifikationer ikke kan benyttes af dette framework og vice versa.

Valg af format

Ovenstående afsnit har gennemgået tre muligheder til valget af formatet. Det blev gennemgået, hvilke fordele og ulemper, der er ved de enkelte formater. På baggrund af denne gennemgang, kan der opstilles en tabel, der viser i hvilken grad formaterne opfylder de stillede krav, se tabel 2.3.

Løsning / Krav	Regulære udtryk	CFG	NetPDL
Pålidelighed	Høj	Høj	Høj
Kompabilitet	Lav	Mellem	Høj
Intuition	Mellem	Høj	Høj
Anvendelighed	Lav	Mellem	Høj[9]

Tabel 2.3: Kravopfyldelse af identificerede løsninger til formatvalg.

Tabellen viser, at NetPDL er det oplagte valg til specifikationen af protokoller. Valget bygger på de tekniske og brugermæssige krav til formatet samt gennemgangen af tre mulige løsninger. NetPDL har iflg. tabel 2.2 og 2.3 vist sig at opfylde de stillede krav bedst. Idet dette valg er truffet, vil næste afsnit koncentrerer sig om NetPDL som teknologi. Afsnittet vil dels give en introduktion til teknologien og dels retfærdiggøre, at NetPDL har en høj anvendelighed som postuleret i tabel 2.3.

2.4 Om NetPDL

NetPDL sproget er en specifikation af konkrete XML tags og fortolkningen af disse. Vha. disse tags kan man definere en protokol, idet man forventer, at det specificerede bliver fortolket i henhold til NetPDL specifikationen. Selve protokolspecifikationen er således blot et XML dokument, mens det er op til en konkret NetPDL fortolker at fortolke protokolspecifikationen i henhold til NetPDL specifikationen. Det er NetPDL fortolkerens ansvar at tage mod en rå datastrøm fra netværket og derefter give oplysninger om, hvilke protokoller, strømmen består af ud fra protokolspecifikationen.

Oplysninger om den specifikke syntaks for en NetPDL protokolspecifikation og fortolkningen af de enkelte dele kan ses i [12] og [14]. En detaljeret gennemgang her er derfor unødvendig. Men det er vigtigt at undersøge, hvilke muligheder, der generelt er i NetPDL og ligeså vigtigt, hvilke begrænsninger NetPDL sætter for projektet. Hvis begrænsningerne giver problemer for projektet, skal løsninger til problemerne findes.

2.4.1 Muligheder NetPDL

NetPDL benyttes overordnet set til at definere felter i en protokol samt definere payloaden, dvs. de data, som protokollen bærer. Det er disse data, der evt. kan udgøre en anden protokol (fx kan payloaden i en IP pakke være en TCP pakke). NetPDL giver mulighed for at definere denne indkapsling af protokoller.

Generelt specificeres felter i den rækkefølge, de optræder vha. `<field>` tagget. Dog giver NetPDL forskellige muligheder for at definere ikke-sekventielle felter,

[9] I afsnit 2.4 retfærdiggøres denne påstand

fx hvis et felt kun optræder som en betingelse af værdien af et andet felt, hvis et felt optræder flere gange, hvis felter optræder i tilfældig rækkefølge e.l. Dette defineres vha. kendte løkke og betingelseskonstruktioner som fx `while` og `if` erklæringer i XML udformning. Derudover kan mere specielle felter defineres vha. forskellige XML konstruktioner. Fx kan man specificere felter, der fylder et antal bit, er tekststrenge eller hele linjers tekst. Der benyttes desuden gennemgående regulære udtryk til at defininere enkeltdele og til tekstsammenligning. For en forklaring af regulære udtryk henvises fx til afsnit 2.3.3. Se eksempler på NetPDL konstruktioner i nedenstående afsnit.

Indkapslingen af protokollen defineres vha. `<encapsulation>` tagget, hvor de fundne felter i en protokol kan benyttes til at afgøre, hvilken protokol, der indkapsles. Fx benytter definitionen af TCP det fundne portnummer til at identificere den indkapslede protokol. Hvis den finder port 80 (i enten afsender eller modtager), forventes det, at specifikationen til HTTP protokollen kan benyttes til at identificere TCP pakkens payload.

Samlet set virker specifikationen fleksibel nok til at kunne håndtere mange spidsfindigheder i diverse protokoller. Men det er først i brug, at begrænsninger viser sig. For at illustrere fleksibiliteten, gennemgås i næste afsnit nogle små eksempler på vidt forskellige protokoller, NetPDL kan håndtere.

2.4.2 Eksempler

I dette afsnit ses der nærmere på nogle protokoller, som defineres i NetPDL. Alle de givne definitioner ses i appendiks C. Ved at vælge protokoller, der har meget forskelligartet struktur, kan vi *sandsynliggøre*, at NetPDL specifikationen er fleksibel.

Et godt udgangspunkt er at undersøge, om den almindelige protokolstak bestående af `Ethernet`, `IP`, `TCP` og `HTTP` kan specificeres vha. NetPDL. Her vil enkelte interessante dele fremhæves for at vise fleksibiliteten i NetPDL. Den samlede definition kan ses i appendiks C.1.

Alle protokoller har følgende grundlæggende opbygning:

```
<?xml version="1.0" encoding="utf-8" ?>
<netpdl name="NETPDL_NAME" version="1.0" date="DATE">
  <protocol name="PROTOCOL_NAME" longname="LONG_PROTOCOL_NAME">
    <format>
      <!-- Definition of protocol fields -->
    </format>
    <encapsulation>
      <!-- Possible interpretations of payload -->
    </encapsulation>
  </protocol>
</netpdl>
```

`NETPDL_NAME` er et navn til den samlede NetPDL definition, `DATE` er datoen, hvor definition er lavet, `PROTOCOL_NAME` er navnet på den protokol, man vil definere og `LONG_PROTOCOL_NAME` er et navn på protokollen, som er mere menneskeligt forståeligt.

Inden for `<format>` tagget defineres protokollens felter vha. `<field>` tagget. Fx kan feltet, der indeholder en afsenders MAC adresse i ethernet protokollen, defineres således:

```
<field type="fixed" name="src" longname="MAC Source" size="6"/>
```

Typen af feltet defineres med `type` attributten, der antager værdi afhængigt af det pågældende felt. Fx betyder `bit` at feltet skal trækkes ud som et antal bit af nogle bytes.

Hvis en protokol har felter, der gentages, kan det defineres vha. `<LOOP>` tagget. Fx kan en HTTP pakke indeholde et arbitrært antal header linier. Dette angives ved:

```
<loop type="size" expr="$packetlength - $currentoffset">
  <if expr="buf2int($packet[$currentoffset : 2]) == 0x0D0A">
    <if-true>
      <field type="line" name="endheader" longname="End Of Header"/>
      <loopctrl type="break"/>
    </if-true>
  </if>

  <switch expr="extractstring($packet[$currentoffset: 0], '[^:]*', 1,
      0)">
    <case value="'User-Agent'">
      <field type="line" name="useragent" longname="User Agent"/>
    </case>
    <!-- other possible header fields are defined here -->
  </switch>
</loop>
```

Løkken fortsætter, indtil der ikke er flere data (angivet ved `<expr>` attributten) eller den sidste header nås (angivet ved `<endheader>` feltet og `<loopctrl>` tagget, der afbryder løkken). Eksemplet viser også eksempler på betingelser angivet vha. `<IF>` og `<SWITCH>` taggene. Disse ser på strengen før et kolon (der findes vha. et regulært udtryk) og parser feltet afhængig af denne streng. `<IF>` tagget bruges til at undersøge, om header linjen er tom, dvs. der ikke er flere headere.

Indenfor `<encapsulation>` tagget defineres mulige tolkninger af payloaden. Også her kan man benytte betingelser, så den rigtige protokol bruges til tolkning af payloaden. For TCP protokollen er der to felter, `sport` og `dport`, hvor hhv. afsenderens og modtagerens porte er specificeret. Disse benyttes til at finde næste protokol, fx ved:

```
<if expr="buf2int(sport) == 80">
  <if-true>
    <nextproto-candidate proto="#http"/>
  </if-true>
</if>

<if expr="buf2int(dport) == 80">
  <if-true>
    <nextproto-candidate proto="#http"/>
  </if-true>
</if>
```

Hvis et af felterne indeholder værdien 80 (`buf2int` konverterer feltets værdi til et heltal) benyttes HTTP protokollen til at parse payloaden. Dette defineres vha. `<nextproto-candidate>` tagget (som her) eller `<nextproto>` tagget. `<nextproto-candidate>` giver definitionen af HTTP protokollen mulighed for at afvise, at det er næste protokol, idet der jo kan være andre protokoller, der benytter port 80. Dem må TCP protokollen så identificere i stedet. I øvrigt vil en komplet definition af TCP protokollen definere andre protokoller til forskellige porte – dette eksempel viser blot mekanismen.

Det samlede eksempel (appendiks C.1) viser, at det er muligt at lave definitionen af hele protokolstakken. Eksemplet viser mange standard dele af NetPDL

inkl. løkker og betingelser, når felter gentages eller afhænger af andre felter.

Et andet interessant og højest relevant eksempel er at undersøge, om protokollen til I^{2}C kan specificeres i NetPDL. Dette er jo en nødvendighed for at lave I^{2}C protokolanalysatoren, der er projektets mål. Definitionen ses i appendiks C.2. Protokollen har den spidsfindighed, at adressen kan være enten 7 eller 10 bit. Er den 10 bit, er den ydermere delt i to, idet den bit, der afgør om der læses eller skrives, deler adressen i hhv. 2 og 8 bit. Dette kan man tage højde for i specifikationen vha. en hjælpevariable, der initialiseres til enten 7 eller 10. Dette afgøres ved at kigge på den første byte i pakken, der kan afgøre, hvilken adressering, der er tale om. Udsnittet herunder viser, hvordan denne adressering håndteres:

```
<execute-code>
  <init>
    <variable type="number" name="$addresslength" validity="
        thispacket"/>
  </init>
  <before>
    <if expr="buf2int($packet[0:1]) ge 0xF0">
      <if-true>
        <assign-variable name="$addresslength" value="10"/>
      </if-true>
      <if-false>
        <assign-variable name="$addresslength" value="7"/>
      </if-false>
    </if>
  </before>
</execute-code>
```

Udsnittet er placeret i `<protocol>` tagget. `<variable>` tagget bruges til at definere en variable og `<assign-variable>` benyttes til at give variablen en værdi. Eksemplet viser endnu en spidsfindighed, som NetPDL kan håndtere.

Samlet viser eksemplerne, at der kan defineres mange forskellige protokolopbygninger i NetPDL. Dette betyder, at muligheden for at lave selve *specifikationen* af protokollerne kan lade sig gøre vha. NetPDL. I næste afsnit ses der på, hvilke begrænsninger, der er i NetPDL. Men disse begrænsninger har ikke noget med protokolspecifikationen at gøre, idet ovenstående gennemgang ikke har afsløret begrænsninger.

2.4.3 Begrænsninger i NetPDL

Selvom ovenstående har vist stor fleksibilitet mht. definition af protokolformatet, er det vigtigt at huske, at NetPDL blot definerer formatet af en protokol. Der kan ikke defineres noget om forhold mellem pakker, og specielt tidslige aspekter kan ikke håndteres med NetPDL alene. Fx kan det ikke afgøres, om en pakke giver mening i den nuværende kontekst, eller om der opdages et HTTP svar uden at der har været en forespørgsel. NetPDL formatet kan blot bruges til at afgøre, at der var en svarpakke på netværket. Formatet kan heller ikke benyttes til at sætte flere pakker sammen for protokoller, der understøtter afsendelse af data i flere pakker (fx TCP).

Hvis sådanne aspekter er krævet, må dette derfor håndteres separat fra genkendelsen af beskederne. I kravanalysen afdækkedes et behov for, at protokolanalysatoren skal være i stand til at opdage alle beskeder på alle niveauer i protokolstakken. Hextet protokollen specificerer en måde, hvorpå data kan sendes vha. flere beskeder. Dette er nødvendigt, da Hextet har en maksimal størrelse af payloaden,

så større datamængder må sendes over flere pakker. I en sekvens af pakker, der udgør én større pakke, kan det vha. NetPDL opdages, at den første pakke er en Hextet besked med en T123 besked, der indeholder data. Men de næste pakker vil opfattes som Hextet beskeder, hvor det ikke er muligt at forstå payloaden, da det er brudstykker af de samlede data. Opdagelsen af den samlede pakke skal derfor defineres et sted.

Dette er et problem, der kræver en nærmere analyse for at finde en løsning. Dette ses der derfor nærmere på i næste afsnit.

2.4.4 Opdelte pakker

NetPDL er ikke i stand til at opdage, når en samlet pakke er opdelt i et antal mindre pakker. I dette afsnit ses der nærmere på, hvordan dette problem kan løses.

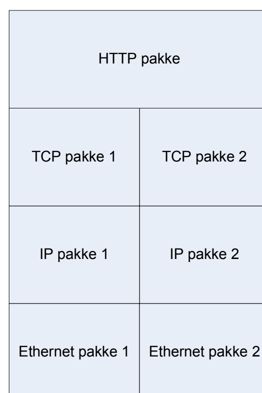
En simpel måde at løse problemet på er, at give protokolanalysatoren mulighed for at kommunikere sådanne specielle pakker med frameworket. Protokolanalysatoren kan registrere en bestemt type beskeder i frameworket. Når frameworket opdager denne type beskeder, får protokolanalysatoren mulighed for at analysere denne internt. På denne måde kan protokolanalysatoren eksempelvis analysere Hextet beskeder og se på, om der er tale om en længere besked. Ud fra disse oplysninger kan den samlede besked sættes sammen, og frameworket kan få besked om, at der er en ny besked (en T123 besked), hvor den samlede længde er tilgængelig.

En anden måde at løse dette problem på er at udvide NetPDL specifikationen, så der bliver mulighed for at tage højde for dette tidslige aspekt. Dette kan lade sig gøre, idet en del af NetPDL specifikationen er, at den skal kunne udvides med brugerdefinitioner. NetPDL fortolkere skal være i stand til at ignorere sådanne udvidelser, hvis de ikke kan forstås. Fordelen ved denne løsning er, at frameworket selv kan opdage sammensatte beskeder. Dette reducerer den nødvendige funktionalitet i protokolanalysatoren, hvilket er et af de stillede krav. Ulempen ved løsningen er, at det må formodes, at det er en meget kompliceret opgave, at definere et generelt sprog indenfor NetPDL's rammer, som løser problemet. Det generelle problem er, at man ved parsing i NetPDL ikke har kendskab til den kontekst, som pakken indgår i. Man kan godt angive, at et givent felt i en protokol fungerer som tæller for pakkens nummer i sekvensen (som tilfældet er ved TCP protokollen), men hvad gør man så, hvis et sådant felt ikke eksisterer? Hvis fx en protokol definerer, at den samlede besked altid består af 4 pakker – så er der jo ingen oplysninger i de enkelte pakker om, hvilket nummer i rækken de er. Det afhænger af konteksten.

Så selvom en generel løsning er at foretrække, vil det være svært at argumentere for, at dette rent faktisk er opnået. Derfor kan den første foreslåede løsning være ligeså god – og noget mindre kompliceret. Man kan sige, at i denne del af protokolanalysen vælges metode 1 i afsnit 2.3.2, da fordelene ved en ekstern løsning her ikke opvejer kompleksiteten af en sådan løsning.

Ved valg af den simple løsning viser nærmere analyse, at der skal gøres yderligere overvejelser, da det kan være problematisk for protokolanalysatoren at sætte mindre pakker sammen til én stor. Lad os tage et eksempel for at ridse situationen op. En HTTP pakke sendes over to TCP pakker, som sendes vha. IP og ethernet, se figur 2.2.

Spørgsmålet er, hvordan denne pakke skal håndteres. Hvis NetPDL selv tager sig af det, vil to TCP pakker med uforståelige data blive genkendt. I dette til-



Figur 2.2: Eksempel på en opdelt pakke.

fælde må protokolanalysatoren registrere hos frameworket, at den er interesseret i TCP beskeder. Protokolanalysatoren opdager, at TCP pakke 1 af 2 og pakke 2 af 2 er ankommet og vil så gerne vise brugeren, at der er en HTTP besked. Men hvordan kan dette gøres, så det samtidigt er intuitivt for brugeren? Den kan ikke blot tage de enkelte byte arrays i de to pakker og sætte dem i forlængelse af hinanden og derefter lade NetPDL fortolkeren analysere den samlede pakke, for fortolkeren forventer jo fx ikke ethernet headere to steder. Det er heller ikke en holdbar løsning blot at tage TCP payloaden fra den første besked og sætte bag på den anden, da TCP headeren i den samlede besked ikke er intuitiv for brugeren (sekvensnummeret vil fx ikke give mening). Tilbage står vi med den mulighed, at frameworket får lov at vise de to TCP pakker hver for sig, og at protokolanalysatoren bygger HTTP pakken og får denne vist. I skærmbilledet skal denne pakke være markeret, så brugeren kan se, at det er en sammensat pakke. Dette er en intuitiv løsning for brugeren (men dette skal selvfølgelig testes i brugertests i en prototype). Vi må derfor se på, hvordan protokolanalysatoren kan sende HTTP beskeden til frameworket, så de enkelte HTTP felter i pakken bliver identificeret rigtigt.

Det er oplagt, at vi skal bruge NetPDL fortolkeren, da den jo i forvejen er i stand til at parse en HTTP besked. Men vores pakke er nu bytes, der kun udgør HTTP headerne og payload. Desværre forventer fortolkeren, at der er en ethernet, IP og TCP header foran HTTP headeren. Vi må altså have NetPDL fortolkeren til at springe direkte til en parsing af HTTP protokollen. Fra frameworket kan vi sætte en variabel, som kan bruges i NetPDL protokolspecifikationen til at springe det rigtige sted hen. I `<startproto>` elementet i NetPDL specifikationen kan denne variable bruges til at finde den rigtige protokol. Variablen bruges altså i stedet for `linklayer` variablen, der ellers identificerer den protokol, som først skal bruges til parsing.

For at løse problemet med at håndtere sammensatte pakker lader vi altså protokolanalysatoren håndtere sammensætningen, og frameworket lader NetPDL fortolkeren analysere den sammensatte pakke, idet der forudsættes understøttelse af disse sammensatte pakker i NetPDL protokolspecifikationen, dvs. det forudsættes, at NetPDL protokolspecifikationen håndterer den variabel, som frameworket sætter.

2.4.5 Eksisterende NetPDL værktøjer

En sidegevinst ved at vælge NetPDL er, at dem, der står bag standarden[10], også har udviklet en pakke, der indeholder nyttige værktøjer for udviklere, der vil benytte NetPDL standarden[11]. For projektet indeholder pakken to vigtige værktøjer. Det ene, `NetBee`, er en komplet NetPDL fortolker, der kan læse en NetPDL protokolspecifikation og benytte denne til at parse en datastrøm. Resultatet af en parsing er en `C` struktur, der indeholder oplysninger om alle fundne protokol headere i den parsede pakke. Det andet værktøj, `NetPDLProtoDB`, kan læse en NetPDL protokolspecifikation og oversætte XML strukturen til en `C` struktur, der i et program er en mere direkte adgang til protokolspecifikationen. Det første værktøj er nyttigt i projektet, når pakker skal parses ud fra en given protokolspecifikation, og det andet værktøj er nyttigt, når frameworket skal have oplysninger om de protokoller, det „forstår“. Det kan fx bruges, når brugeren skal definere filtre.

Eneste ulempe ved værktøjerne er, at de er i meget tidlige versioner uden megen support. Det kan altså ikke afvises, at der kan være fejl i dem. Håbet er dog, at de kan benyttes tilfredsstillende alligevel, så det ikke bliver nødvendigt at reimplementere dem.

Værktøjerne er tilgængelige som en „unmanaged dll“, dvs. de kan ikke direkte anvendes i programmer til .NET platformen. For at benytte værktøjerne kan der laves et „wrapper“ bibliotek, der stiller værktøjernes services til rådighed som en „managed dll“, der kan benyttes direkte i programmer til .NET platformen. Et sådant wrapper bibliotek kan med fordel laves i C++/CLI som beskrevet i afsnit 1.3.2.

2.5 Muligheder for interaktion med netværket

Det er nu afklaret, at NetPDL kan benyttes til at specificere genkendte protokoller i frameworket. Men det er ikke afklaret, hvordan interaktion med netværket kan foretages. Dette aspekt behandles i dette afsnit.

Der er overordnet tre krav, der skal opfyldes:

1. Det skal være simpelt at udføre simple operationer, dvs. det skal ikke være nødvendigt at skrive mange linjers kode, hvis man blot skal udføre én simpel operation.

2. Det skal være muligt at udføre mere komplekse operationer, hvis dette ønskes. Dette giver selvfølgelig mere kompleks kode.

3. Den benyttede teknologi til interaktionen skal være eksisterende, dvs. der skal fx ikke opfindes et nyt scriptsprog til formålet.

Kravene blev fundet i brugerundersøgelsen. I forbindelse med krav 3 blev Ruby foreslået som et muligt scriptsprog. Da Ruby er et script sprog kan operationer udføres med bare én linjes kode. Findes der ikke operationer til formålet, kan disse defineres. Derfor opfylder Ruby krav 1. Ruby indeholder et rigt bibliotek af funktioner, der kan benyttes til mere komplekse operationer. Er dette ikke nok, kan der laves yderligere biblioteker til brug i programmer – enten i Ruby sproget

[10] Netgroup ved Politecnico di Torino, Italien.

[11] Pakken kan downloades på `http://www.nbee.org/Download/nbDevPack.zip` – tilgængelig 1/6-2007.

selv eller også i C. Derfor opfylder sproget også krav 2. Det ses altså, at Ruby opfylder brugernes krav til interaktionen med netværket. Tilbage er der nu at se på, om Ruby også opfylder de tekniske krav til interaktionen. Det skal være muligt at integrere sproget i frameworket, så scripts kan udføres direkte i programmet, der jo er skrevet til .NET platformen.

Ruby kan benyttes på to måder: Eksekvering af et script eller eksekvering af blot én linjes kode. Vi har her brug for eksekvering af et helt script. Dette gøres fx fra kommando linjen, hvor Ruby fortolkeren (`Ruby.exe`) kaldes og får scriptet som input (enten en fil som argument eller fra standard input). Herefter eksekveres scriptet til ende, eller indtil der opstår en fejl. Denne procedure kan udføres fra et .NET program vha. `System.Dianostics.Process` klassen.

Til sidst resterer at undersøge, hvordan et Ruby script kan interagere med .NET programmet, hvis scriptet fx ønsker at sende eller modtage en pakke til/fra netværket via de interfaces til netværket, som protokolanalysatoren definerer. Vi ønsker en interproces kommunikation mellem de to processer. Kommunikationen skal være asynkron, så et Ruby script fx kan sende en besked og derefter fortsætte andet arbejde, og den skal være persistent, så det fx er sikkert, at frameworket modtager en besked fra et Ruby script om at sende en besked til netværket. Der findes forskellige metoder til at opnå en sådan kommunikation.

Et simpelt eksempel på en mulig kommunikationsform er at udveksle data via en datafil, som det ene program skriver, og det andet program læser. Løsningen er ikke optimal, da der fx kan være problemer med at „opdage“ en sådan kommunikationsfil (fx vha. polling), men det er en metode som både Ruby og .NET understøtter. Kommunikationen er altså mulig, men ikke optimal. Andre og bedre metoder understøttes dog også, så sådanne metoder skal undersøges, når kommunikationen designes i kapitel 3. For nu er det tilstrækkeligt at konkludere, at Ruby opfylder de tekniske krav til interaktion med netværket.

2.6 Konklusion på indledende analyse

Den indledende analyse har givet en kravspecifikation til projektet. Denne er fundet vha. interviews med brugere af det endelige produkt. Kravspecifikationen danner grundlag for designet af den første prototype til frameworket med tilhørende protokolanalysator. Prototypen skal brugertestes, så uoverensstemmelser med kravspecifikationen findes. Desuden kan brugertests afsløre nye krav til frameworket og/eller protokolanalysatoren.

Det blev analyseret, hvilke muligheder der er for generelle formater til definering af protokoller, som frameworket kan benytte til at parse netværkspakker. Det blev fundet, at kravspecifikationen bedst blev overholdt ved at finde et eksternt format, dvs. protokolspecifikationen lægges i en ekstern datafil, som frameworket læser. Tre formater blev overvejet til dette format: Udvidede regulære udtryk, context-free grammars og NetPDL.

NetPDL giver de største fordele. Vi får opfyldt de krav, vi har til formatet, idet det er en standard måde, hvorpå man kan definere en protokol på en intuitiv måde. Formatet er entydigt og fleksibelt, idet der er understøttelse til at definere spidsfindigheder i en protokol. Hvis det skulle ske, at en protokol ikke kan defineres vha. NetPDL, kan man i NetPDL angive et plug-in, som kan udføre opgaven.

En anden fordel ved NetPDL er de værktøjer, der allerede eksisterer, der derfor kan gøres brug af i projektet. Disse reducerer implementeringen af pakkehåndterin-

gen i frameworket betydeligt. Dog skal der laves en .NET wrapper til værktøjerne, så de kan benyttes på denne platform.

NetPDL har den ulempe, at sammenhænge mellem pakker ikke kan håndteres – specielt kan NetPDL ikke håndtere pakker, der er delt i mindre pakker. Dette problem løses ved at indbygge funktionalitet i protokolanalysatoren, så den håndterer de opdelte pakker.

Samlet viste analysen af NetPDL, at det er en brugbar løsning til problemet.

Til interaktion med netværket er Ruby blevet undersøgt som mulighed. Det blev fastslået, at sproget dels opfylder brugerkravene til interaktion med netværket og dels opfylder de tekniske krav, der for at integrationen af sproget i værktøjet er mulig. Da Ruby desuden kan læres hurtigt og er simpelt at bruge, er det særdeles velegnet til formålet.

Efter arbejdet med disse problemstillinger kan systemet designes i henhold til kravspecifikationen, og frameworket designes med integration af en NetPDL fortolker samt Ruby sproget. Dette emne behandles i det følgende kapitel.

Kapitel 3

Design af systemet

Kapitlet omhandler systemets design, hvor detaljerne i skitsen i afsnit 1.3.4 (figur 1.2, s. 7) klarlægges. Det beskrives, hvordan frameworkets arkitektur designes internt, dvs. hvordan det opbygges uden at se på, hvordan det interagerer med eksterne moduler (protokolanalysator og plug-ins). I det efterfølgende afsnit beskrives det, hvordan dette interne design håndterer beskeder fra netværket og interagere med netværket, da det er centrale elementer i designet af frameworket. Slutteligt betragtes kommunikationen med andre dele af systemet, protokolanalysatoren og plug-ins, jf. figur 1.2.

Indholdsfortegnelse

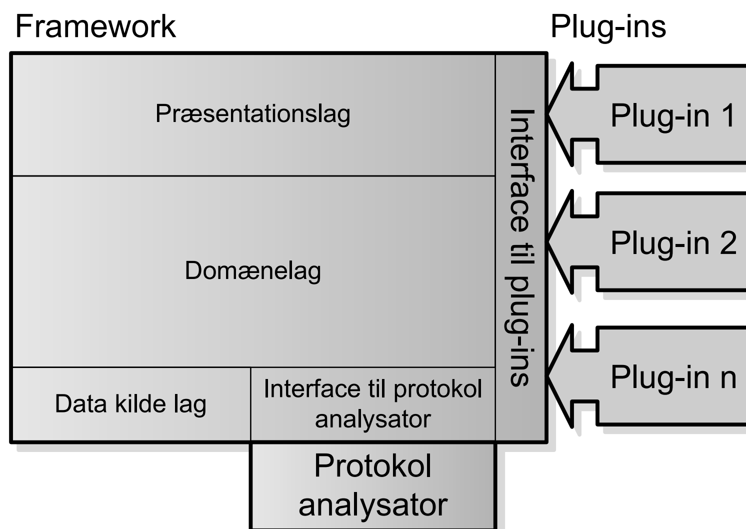
3.1 Overordnet design strategi **35**
3.2 Brug af modulopbygget design **36**
3.3 Design af frameworkets arkitektur **38**
3.4 Dynamisk flow af netværkspakker **43**
3.5 Integration af script sprog **46**
3.6 Interface til protokolanalysator **51**
3.7 Interface til plugin **52**
3.8 Konklusion på design af systemet **54**

3.1 Overordnet design strategi

Figur 3.1 viser et overordnet design af systemet. Figuren kan ses som en videreudvikling af figur 1.2 s. 7. Figuren (3.1) afspejler, at målet med projektet er et generisk framework, der benytter en protokolspecifik protokolanalysator. Dette indebærer, at designet er modulært, hvilket figuren også viser. Protokolanalysatoren er det nederste lag i arkitekturen, idet den definerer nogle protokolspecifikke services, som frameworket benytter sig af. Frameworket stiller services til rådighed

dels for plug-ins, der kan tilføje funktionalitet til frameworket, dels til brugeren gennem en brugergrænseflade. Frameworket benytter sig ikke af services i plug-ins, der heller ikke (nødvendigvis) har kommunikation imellem sig.

En sådan modulær opbygning indkapsler data, så utilsigtet adgang undgås i god objektorienteret stil. Fx bliver frameworket et samlet bibliotek, der indeholder services relevante for plug-ins (via plug-in interface) og bruger (via præsentationslaget), der derfor kun ser disse services, men ikke de irrelevante indkapslede data [9].



Figur 3.1: Overordnet design af systemet.

Dette overordnede design er ikke detaljeret nok til at starte en implementering af systemet. Designet beskriver fx intet om, hvordan frameworket skal designes internt (udover at den interne lagopdeling, som der vendes tilbage til, er medtaget). Men et „Big design up front“[1] er heller ikke løsningen, da målet er at skabe en prototype, der efterfølgende skal brugertestes. Brugertestene vil give ændringer til designet, hvorfor det er fordelagtigt ikke på forhånd at have alt for fastlagte strukturer. Til gengæld er det en fordel at have fleksible strukturer, der ikke er så detaljerede, men som til gengæld er modtagelige overfor ændringer („Design for change“ [9]). Designet skal altså være tyndt (udetaljeret) og fleksibelt.

3.2 Brug af modulopbygget design

Et tyndt og fleksibelt design kan opnås ved at opdele systemet i selvstændige moduler, der hver især har et ansvarsområde og tilbyder services til andre moduler. Denne modulopbygning er allerede brugt i det overordnede design, idet frameworket, protokolanalysatoren og de enkelte plug-ins kan opfattes som moduler. De kan inddeles yderligere i moduler, der igen har et ansvarsområde indenfor det modul, hvori de er placeret. Et eksempel er det modul i frameworket, der tager sig af

[1] Betegnelse for den omdiskuterede traditionelle softwareudvikling, hvor systemet designes detaljeret og fuldstændig inden implementeringen startes [4, s. 14].

Boks 1: Mediator

Mediator bruges for at samle kommunikation mellem objekter ét sted. Objekterne kender kun til mediator objektet, så når et objekt ønsker at kommunikere med et andet objekt, sker det gennem mediator objektet. Mediator objektet kender til gengæld alle de objekter, der ønsker at kommunikere. Mediatoren har kun ringe funktionalitet i sig selv, men fungerer i stedet som en formidler af kommunikationen. Fordelen ved en mediator er, at man undgår et kompliceret net af referencer mellem objekter. En anden fordel er, at ændringer i objekter kun kræver ændringer i mediator objektet, hvorved slippage problemet minimeres [3, s. 273ff].

pakkeparsing. Modulet tilbyder en service, der tager imod en strøm af bytes og returnerer en konkret pakke med en håndterbar struktur. Dette gør modulet vha. forskellige NetPDL services (i andre moduler).

Denne opbygning er fleksibel, idet ændringer kan foretages i de enkelte moduler uden det behøver påvirke de øvrige moduler. Desuden kan hvert modul udvikles individuelt og til sidst sættes sammen med andre. Derfor er Test-Driven Development (se afsnit 1.3.3) en oplagt metode til at udvikle de enkelte moduler, idet der på forhånd kan laves unittests for modulet (red fase), som er færdigimplementeret, når testene forløber uden fejl (green fase og refactoring fase) [1]. Hvis der efter brugertests opstår nye krav til modulet, kan der blot tilføjes tests til unittesten af modulet – og den tre fasede proces gentages (red-green-refactoring). Den praktiske del af emnet behandles i afsnit 4.1.

3.2.1 Kommunikation mellem moduler

I systemet med moduler vil der være brug for kommunikation mellem moduler. Når brugeren fx udfører en handling kan det sætte en kædereaktion i gang, hvor mange forskellige moduler skal involveres, idet de har brug for at bruge services fra andre moduler. Hvis alle moduler indeholder referencer til en række andre moduler, kan der hurtigt opstå et kompliceret net af referencer på kryds og tværs. Resultatet er kode, der er uoverskuelig og svær at udvide og vedligeholde, idet det er svært at overskue ændringer. Problemet „the slippage problem“ er beskrevet i [9]. Vi kan benytte et `Mediator` pattern, der samler al kommunikation i ét modul, se boks 1.

I de følgende afsnit, der beskriver designet af systemets enkeltdele, vil modul opbygningen fremgå. Afsnittene vil koncentrere sig om at beskrive, hvilket ansvarsområde, de enkelte moduler har, hvilke services, de tilbyder, og hvilke services de benytter fra andre moduler. Den mere detaljerede opbygning af de enkelte moduler beskrives i kapitel 4, hvor resultatet af TDD processen præsenteres.

3.3 Design af frameworkets arkitektur

Frameworkets overordnede ansvar er at være i stand til at opfylde de krav, der blev fundet i afsnit 2.2. Dvs. der skal være en håndtering af en indkommen datastrøm af netværkspakker og en interaktion med netværket. Disse dele skal ske ud fra en NetPDL definition af en række protokoller. Derudover skal der være en brugergrænseflade, som tager sig af interaktionen med brugeren. Vi kan opbygge disse dele i tre lag som beskrevet i [2, s. 17ff]: Datakilde-, domæne- og præsentationslaget, se figur 3.1. Datakilden inkluderer eksterne dataressourcer samt ekstern kode i biblioteker/pakker. Domænelaget indeholder den logik, der udfører applikationens hovedopgaver. I præsentationslaget er interaktionen med brugeren defineret i form af grafiske elementer samt input fra mus og tastatur. Under frameworket befinder protokolanalysatoren sig. Frameworket benytter et interface til at tilgå protokolanalysatorens services. Dette interface befinder sig under domænelaget, da det er dette lag, der skal tilgå protokolanalysatoren.

I de følgende afsnit gennemgås, hvilke moduler de enkelte lag består af, og hvilket ansvarsområde, de har. Desuden vil det blive beskrevet, hvilke services, de tilbyder andre moduler i deres eget lag, og hvilke services, de tilbyder moduler i andre lag. Derefter ses der på, hvordan kommunikationen mellem moduler i samme lag samt kommunikationen mellem forskellige lag håndteres, dvs. hvordan en mediator realiseres, se boks 1, s. 37.

3.3.1 Datakildelaget

For denne applikation er datakilden de netværkspakker, der skal parses vha. NetPDL protokoldatabasen. Men disse pakker kommer fra protokolanalysator delen, da interfacet til det virkelige netværk må specificeres dér. Til gengæld er NetPDL databasen og fortolkeren samt Ruby fortolkeren en del af datakilden, da disse foreligger som eksterne ressourcer. Andre eksterne biblioteker er på samme måde en del af datakildelaget, fx de biblioteker, som NetPDL fortolkeren gør brug af.

En anden del af datakildelaget er permanent lagring af data. Det drejer sig om at gemme pakke optagelser, scripts og definerede filtre. Der er ikke brug for hurtig dynamisk adgang til dataene på det permanente lager, så en indekseret database er unødvendig, men i stedet kan dataene blot hentes ind i hukommelsen fra eksterne filer. Disse filer er også en del af datakildelaget.

Det ses, at datakildelaget er relativt tyndt. Det er blot plads på harddisken, der opbevarer dels datafiler, dels eksterne biblioteker, der er nødvendige for domænelaget. Det interessante er derimod *interaktionen* med datakildelaget, som sker fra domænelaget.

3.3.2 Domænelaget

Domænelagets to hovedopgaver er at parse netværkspakker vha. en given NetPDL database samt at kunne interagere med netværket. Disse opgaver fordeles på en række moduler, der hver især har ansvar for en mindre del af opgaverne. Nedenfor beskrives de enkelte modulers ansvar. Der vendes tilbage til de mere komplicerede moduler og de moduler, der kræver yderligere beskrivelse i de senere afsnit. Nedenfor refereres der til de relevante afsnit.

Pakkedatabase Indeholder alle opdagede pakker og kan udføre operationer på disse, fx filtrering. Dette modul skal ved opdagelse af en pakke sørge for

at gemme den i interne datastrukturer. Modulet indeholder mulighed for at definere et filter vha. `Filter` modulet, der efterfølgende kan tilføjes pakkerne. De definerede filtre gemmes i en filterdatabase (en datastruktur), der kan serialiseres til fx en fil på disken. Modulet er derfor ansvarlig for interaktionen med datafilen med filtre i datakildelaget. Nærmere beskrivelse af den interne datastruktur beskrives i afsnit 3.4.2.

Pakke En samling datastrukturer til at gemme identificerede mønstre i en bytestrøm på netværket (en opdaget „netværkspakke"). En pakke har forskellig udseende undervejs i systemet, idet den starter som en række bytes, men ender som en fast struktur, hvor de forskellige dele af pakken er fastsat. De opdagede protokoller og de enkelte felter i protokollerne er defineret. For hver af disse definitioner er der en reference til protokol/felt definitionen i protokoldatabasen samt konkrete oplysninger fra den konkrete pakke såsom feltets værdi, feltets placering i de rå data osv. Modulet indeholder services til at stille disse oplysninger til rådighed. Datastrukturerne for en pakke beskrives i afsnit 3.4.1.

Pakkeparser Skaber en adgang til NetPDL fortolkeren i datakildelaget. Dette modul er i stand til at omsætte en række bytes til en pakke fra pakkemodulet, der kan arbejdes videre på i systemet (vises, gemmes i database osv.)

Protokoldatabase Indeholder oplysninger om de protokoller, som frameworket er i stand til at forstå, dvs. parse fra en række bytes. Oplysningerne hentes fra NetPDL definitionen, der er knyttet til frameworket af protokolanalysatoren. Modulet indeholder services, så disse oplysninger kan hentes af andre interesserede moduler.

Filter En samling datastrukturer, der kan gemme et defineret filter. Filtret er fleksibelt nok til, at der kan filtreres på specifikke dele af en pakke. Disse dele blev identificeret i kravspecifikationen. Overordnet er strukturen et arbitrær dybt træ bestående af individuelle filtererklæringer, der specificerer én specifik del af filtret. Når filtret skal evaluere en pakke (eller en samling pakker), gennemløbes dette træ af erklæringer, der hver især får ansvar for evalueringen af netop dén del af filtret. Filtererklæringerne har desuden ansvar for en række andre opgaver, fx serialisering til `xml`. Denne træstruktur beskrives i detaljer i afsnit 3.4.3.

Filterdatabase Indeholder de definerede filtre og skaber adgang til disse.

Ruby script eksekverer Kan eksekvere Ruby scripts, som gives til modulet som input. Modulet benytter den eksterne Ruby fortolker og styrer eksekveringen, dvs. giver oplysninger om output fra scriptet, samt hvornår det startes og termineres.

Ruby interface Skaber overgangen mellem Ruby scriptet og frameworket. Dette modul modtager beskeder fra et script, udfører den ønskede operation og sender svar tilbage til scriptet. Detaljerne i denne overgang mellem Ruby og C# beskrives i afsnit 3.5.2.

Ruby script generator Genererer et Ruby script som stiller relevante funktioner til rådighed for eksekverende Ruby scripts i frameworket. Ruby scriptet genereres ud fra protokoldatabasen, men kun når det er nødvendigt (hvis

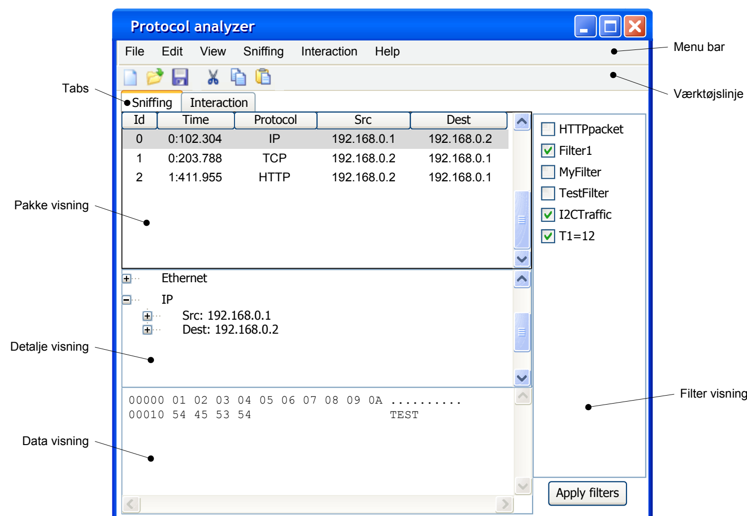
databasen er ændret eller frameworket eksekveres for første gang). Genereringen af dette hjælpebibliotek beskrives i detaljer i afsnit 3.5.1.

Ruby syntaks læser Læser Ruby scripts og giver oplysninger om syntaksen. Oplysningerne kan bruges til at markere syntaksen i Ruby kode.

IO handler Skaber overgangen til datakildelagets filer, hvor optagelser er lagret. Modulet kan gemme en pakkedatabase i en fil i forskellige formater og hente dem ind igen. Desuden håndterer modulet lagring af script filer og definerede filtre i et `xml` format.

3.3.3 Præsentationslaget

I præsentationslaget er alt visuelt defineret. Hvert modul tager sig af hver deres område af det visuelle felt. Det største grafiske område er dækket af et område, der kan have forskelligt indhold. Dette styres vha. to faneblade. Der er et modul til at definere udseendet af indholdet for hver af de to faneblade: Sniffing modulet til udseendet ved sniffing og interaktions modulet til udseendet ved interaktion med netværket. Figur 3.2 og 3.3 viser skitser af brugergrænsefladerne.



Figur 3.2: Skitse af sniffing visningen.

Sniffing modulet består af følgende moduler, idet udseendet af dette er delt yderligere op, se figur 3.2:

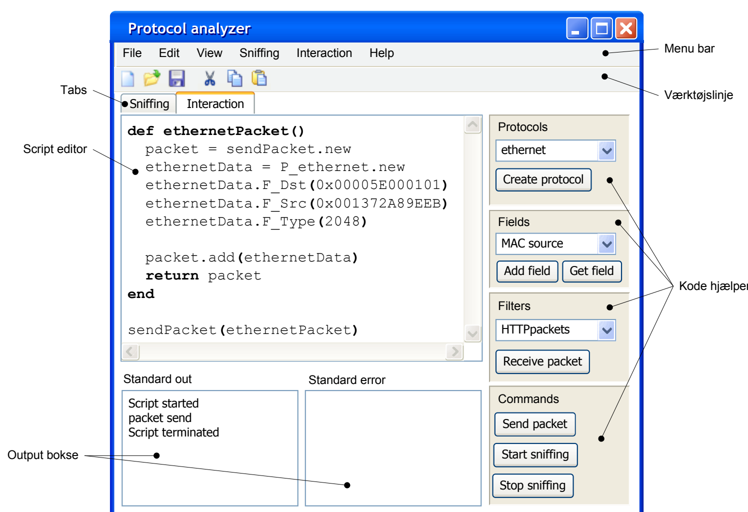
Pakkevisning Modulet indeholder en liste over de pakker, der er opdaget på netværket. Det er dette moduls ansvar at vise en pakke, når den opdages. Det skal desuden kommunikere ud, når der klikkes på en pakke i listen, idet det betyder, at brugeren vil se detaljer om pakken.

Detaljevisning Dette modul er i stand til at vise detaljer om en pakke. Når der vælges en pakke i pakkevisningen, viser dette modul detaljer om pakken. Der opbygges et træ med pakkens felter. Navnet på feltet samt feltets værdi

vises. Disse informationer er en del af pakkestrukturen og kan derfor hentes derfra. Det er dette moduls ansvar at kommunikere ud, når der klikkes på en af pakkens detaljer, da dette betyder, at brugeren vil se, hvor det pågældende felt er lokaliseret i pakken.

Datavisning Modulet er i stand til at vise en pakkes indhold i byteform, dvs. de rå bytes, pakken er opbygget af. Når en pakke vælges i pakkevisningen, viser modulet denne pakkes rå dataindhold. De enkelte bytes vises dels med hexadecimal notation og dels med ASCII notation. Når brugeren vælger et felt i detaljevisningen, er modulet ligeledes i stand til at markere, hvor i de rå bytes, det pågældende felt er taget fra, idet disse oplysninger stilles til rådighed af pakkemodulet.

Filtervisning Modulet kan vise tilgængelige filtre, som kan sættes på eller fjernes fra listen af opdagede pakker. Når brugeren beslutter at gøre dette, kan modulet sørge for at kommunikere denne beslutning ud, så pakkevisningen kan vise de korrekte pakker.



Figur 3.3: Skitse af interaktion visningen.

Interaktionsmodulet er delt op i følgende moduler, se figur 3.3:

Script editor En editor, hvor et Ruby script kan indtastes. Modulet sørger for at farve koden i henhold til Ruby syntaksen. Editoren modtager oplysninger om, at scriptet ønskes eksekveret, hvorefter modulet sender scriptet til Ruby executor modulet, som udfører eksekveringen.

Outputbokse Der er to bokse, der viser output fra et eksekverende script. De viser output til hhv. standard out og standard error fra scriptet.

Kodehjælper Her er der mulighed for at generere kode vha. musen. Der kan vælges et felt eller protokol, som man ønsker at tilføje en pakke, man er ved at bygge. Der kan vælges et felt, som man ønsker at hente fra en modtaget

pakke. Man kan lave en kommando, som modtager en pakke fra netværket, der opfylder et specificeret filter og sidst er der mulighed for at generere kode til kommandoerne „send pakke“, „start sniffing“ og „stop sniffing“. Modulet hjælper brugeren til at generere kode uden at skulle bekymre sig alt for meget om syntaksen.

Udover dette hovedområde består præsentationslaget af forskellige andre moduler:

Værktøjslinjer og menubar Det er nogle moduler, der hver især har ansvaret for på en for brugeren let tilgængelig måde at kunne tage imod ordrer fra brugeren i form af klik på knapper. Det er modulernes ansvar at kommunikere brugerens klik videre, så ordren kan udføres.

Interfacevælger Modulet viser brugeren et vindue, hvor tilgængelige netværksinterfaces kan vælges. Oplysninger om disse interfaces kommer fra protokolanalysatoren. Det er modulets opgave at vise listen over disse interfaces og kommunikere brugerens valg ud, så sniffing/interaktion med netværket sker fra det rigtige interface. Desuden giver modulet mulighed for, at brugeren kan sætte interfacespecifikke indstillinger, se desuden afsnit 3.6.

Filtereditor Modulet tilbyder brugeren et vindue, hvor filtre kan defineres. Disse filtre kan senere påføres listen af opdagede pakker eller benyttes i interaktionen med netværket. Det er modulets opgave at tilbyde brugeren fleksible muligheder for at definere et filter, så der er de muligheder, der er krævet i kravsspecifikationen. Bl.a. skal informationer fra protokollerne defineret i den til frameworket givne protokoldefinition kunne bruges, så man fx kan filtrere på værdier i bestemte felter. Det er modulets ansvar at oversætte brugerens valg til et filter, der derefter kan benyttes i filtervisningen.

IO handler Skaber en fælles indgang til domænelagets IO handler. De andre moduler i præsentationslaget kan bruge dette modul, når der forespørges lagring eller hentning af data fra filsystemet.

Med disse moduler stiller præsentationslaget domænelagets services til rådighed for brugeren.

3.3.4 Kommunikation i arkitekturen

Én mediator til at samle kommunikationen i hele arkitekturen er ikke nok. Det er mere hensigtsmæssigt, at samle hvert lags kommunikation i en mediator. Da datakildelaget og interfacet til protokolanalysatoren ikke kommunikerer internt, bliver der tale om 2 mediatorer: En i domænelaget og en i præsentationslaget. Fordelen ved denne opbygning er, at lagets services samtidig kan udstilles her, så modulerne i præsentationslaget kan udføre operationer i domænelaget gennem domænelagets mediator. Desuden kan mediatorerne benyttes af fremtidige plug-ins til at udføre deres ønskede operationer i frameworket.

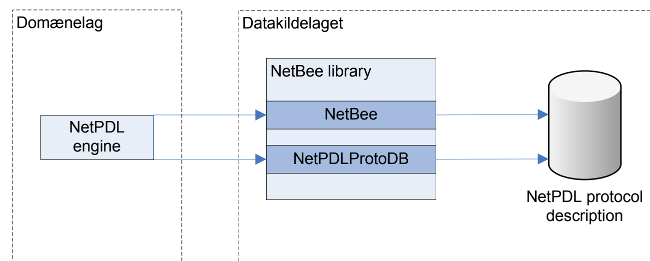
For at kommunikationen virkelig er samlet ét sted, kan der ikke være flere mediatorer der samler den samme kommunikation. Dette problem kan håndteres med et design pattern, `singleton`, som kombineres med mediator, se boks 2.

Når domænelaget skal kommunikere med datakildelaget kan dette håndteres på to måder. Skal datafiler indlæses kan dette gøres direkte fra de moduler, der ønsker indlæsningen. Disse moduler har så eneretten til de pågældende datafiler.

Boks 2: Singleton

Singleton er et design pattern, der benyttes, når man kun ønsker én objektinstans af en klasse. Singleton kan implementeres ved at sørge for, at klassen ikke kan benyttes til oprettelsen af objekter. Istedet benyttes en anden indgang i klassen, der altid returnerer den samme instans af klassen (og opretter den ved første tilgåelse). I C# gøres dette i praksis ved at lave alle constructors private og have en statisk offentlig indgang til en statisk instans af den pågældende klasse [3, s. 127ff].

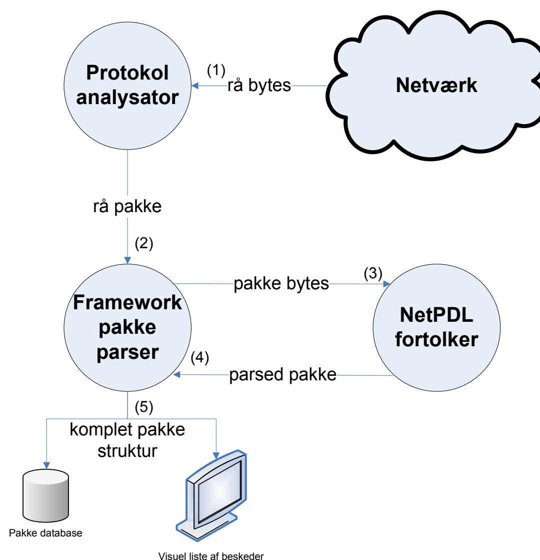
Når eksterne biblioteker skal benyttes, kan det ske gennem mediatoren for domænelaget for at skabe en ensartet adgang. Eksempelvis skaber domænelagets mediator adgang til NetPDL fortolkeren ved at benytte `NetPDLengine` modulet. Dette modul benytter `NetBee` biblioteket til at få oplysninger om genkendte protokoller (via. `NetPDLProtoDB`) hhv. parse pakker (via `NetBee`). `NetBee` biblioteket benytter NetPDL protokolspecifikationen til at tilbyde de services, som domænelaget er afhængige af. `NetPDLengine` tilbyder således en abstraktion af `NetBee` bibliotekets services til andre moduler i domænelaget, idet den fungerer som wrapper for `NetBee` biblioteket, se afsnit 2.4.5. Figur 3.4 viser dette eksempel på kommunikation mellem domænelaget og datakildelaget.



Figur 3.4: Kommunikation mellem domænelaget og datakildelaget.

3.4 Dynamisk flow af netværkspakker

I dette afsnit ses der nærmere på, hvordan netværkspakkerne håndteres i systemet. Målet er, at de kommer ind fra netværket som rå bytes og ender med at kunne sendes op på skærmen som en detaljeret struktur, hvor man kan hente alle oplysninger om pakken. Figur 3.5 viser dette dynamiske flow af en netværkspakke, idet den viser, hvilke af de tidligere omtalte moduler, der håndterer pakken undervejs i systemet. Pilene på figuren angiver pakkens vej fra modul til modul og hvilken tilstand, pakken er i.



Figur 3.5: Pakkens vej gennem systemet.

Som det ses af figuren, er det protokolanalysatorens opgave at modtage de rå bytes fra netværket, idet den ved, hvornår en pakke er modtaget. Denne rå pakke kun bestående af bytes sendes videre til frameworket. Det er op til protokolanalysatoren at levere en pakke, der opfylder et interface, som frameworket specificerer for at kunne trække de nødvendige informationer ud af den rå pakke. Udover de rå bytes kan frameworket derfor få oplysninger om tidspunktet, pakken var på netværket. I frameworket er det pakkeparsermodulet, der modtager pakken. Pakken påføres et for frameworket unikt id inden den sendes videre til NetPDL fortolkeren, der giver en struktureret pakke tilbage, hvorfra pakkens enkeltdele kan findes. Frameworkets pakkeparser ændrer strukturen til en veldefineret datastruktur, hvor relevante oplysninger hurtigt kan hentes.

Pakken når sin endelige destination i præsentationslagets pakkevisning og pakkedatabasen og har her en udformning, der kan bruges fornuftigt til visning og filtrering.

3.4.1 Pakkens datastrukturer

Der er flere steder i systemet brug for en hurtig adgang til pakkernes data. Hvis applikationen var protokolspecifik, kunne de enkelte felter stilles til rådighed direkte (fx via metodekald eller som variabler). Men pakken må have en generel struktur, da pakkemodulet er en del af det generelle framework. Oplysninger, som er fælles for alle pakker, kan tilgås direkte: Tidspunkt for opdagelse på netværket, id og den rå data. Til gengæld må de konkrete protokolinformationer være tilgængelige på en generel måde. Til det kan bruges en hashtabel, hvor protokollernes unikke navn mappes til den parsede protokol med alle dens informationer. Hver parsed protokol indeholder endnu en hashtabel, hvor oplysninger om de parsede felter for den pågældende protokol er opbevaret. Igen mappes felternes unikke navn til parsede felter med det pågældende navn, idet der kan være flere ens felter, der

derfor har samme navn fx i forbindelse med et felt, der gentages. Idet felter kan have en dyb hierarkisk opbygning består hvert parsed felt også af en hashtabel af indeholdte felter. Dermed opnås en rekursiv definition af parsede felter. Fra en pakke kan de enkelte felter tilgås i tiden *O(n)*, hvor n er feltets højdeposition i hierarkiet. I de fleste tilfælde er n lille $(1-4)$ og adgangen derfor hurtig.

Ovenstående gennemgang viser, at en pakke har en datastruktur, der giver hurtig adgang til alle felterne i pakken. Men pakken indeholder yderligere information til at skabe en hurtig adgang til informationerne. En pakke giver også en direkte adgang til protokolstakken. Pakken kan give navnet på den øverst opfattede protokol i stakken og hver parsed protokol har en reference til den næste protokol i stakken. Der er altså en lineær adgang til protokolstakken, dvs. i tiden *O(n)*, hvor n er antallet af protokoller i stakken.

3.4.2 Pakkedatabasens struktur

Pakkerne gemmes i pakkedatabasemodulet, der skal sørge for en hurtig adgang til de enkelte pakker ud fra fx filter oplysninger. Man kunne placere pakkerne i en lang liste og når pakker, der opfyldte et krav skulle findes, kunne man iterere over hele listen og undersøge hver enkelt pakke. Men det forventes, at listen af pakker kan bliver meget lang, hvorfor dette ikke er en god løsning, da en filtrering vil tage tiden *O(n)*, hvor n er antallet af pakker i databasen – altså en meget lang køretid, idet n er stor.

En anden tilgang er at udnytte, at man kender den måde, hvorpå pakkerne ønskes fundet. Det giver fx ikke mening at søge efter pakker, der indeholder feltet „feltnavn", idet „feltnavn" kun giver mening i sammenhæng med den protokol, hvori den er defineret. Derfor kan pakkerne først inddeles efter hvilke protokoller, de indeholder. Pakkedatabasen indeholder en relation, hvor protokolnavnet knyttes til en samling af pakker, der alle har det til fælles, at de indeholder en specifik protokol. Da en pakke er sammensat af en protokolstak, dvs. indeholdende flere protokoller, vil pakken optræde ligeså mange steder i relationen, som der er protokoller i stakken. Udover at protokolnavnet knyttes til en samling af pakker, knyttes protokolnavnet også til nye relationer, der knytter feltnavne til en samling pakker. Denne relation er på præcis samme måde som protokolrelationen, men indeholder i stedet de felter, som den pågældende protokol indeholder. Pakkerne bliver igen refereret, så alle pakkerne i en samling i feltrelationerne har det tilfælles, at de har dels en protokol dels et felt til fælles. Sådan fortsættes en rekursiv relation, så protokoltræerne afspejles. Man kan på denne måde hurtigt finde de pakker, der indeholder en given protokol og givne felter. Hvis man ønsker at finde pakker med en given protokol og med k felter, tager det altså tiden *O(k)*, hvilket er betydeligt bedre end de ovenfor *O(n)*, da n er meget større end k.

Ulempen ved denne metode er, at man vil have referencer til den samme pakke mange steder i databasen. Hvis man har p pakker, der hver indeholder gennemsnitlig m protokoller i stakken, der hver indeholder gennemsnitlig f felter hver (inklusive felter længere nede i hierarkiet), vil et estimat for det samlede antal af pakkereferencer, n, være:

$$n = p \cdot m \cdot f$$

Antallet af pakkereferencer i datastrukturen vil altså estimeret være $m \cdot f$ gange større end antallet af pakker. En pakke reference vil i et 32 bit system fylde 32 bit

i hukommelsen, så datastrukturen vil kræve $4 \cdot p \cdot m \cdot f$ bytes. I et typisk eksempel[2] vil $m = 3.5$ og $f = 10$. Her vil hver pakke altså kræve 140 bytes hukommelse i datastrukturen. I 100 mb hukommelse, som man må formode minimum kan afsættes til formålet, kan der således være ca. 750.000 pakker. Det ses, at med selv store datamængder, vil hukommelses forbruget ikke være urealistisk stort. Og fordelen ved datastrukturen i forhold til hastighed opvejer langt dette ressource forbrug.

3.4.3 Struktur af filter

I kravspecifikationen blev det fundet, at filtrering skal kunne foretages på vilkårlige felter i vilkårlige protokoller. På protokollerne skal der kunne filtreres på, om protokollen er til stede i pakken. Dette er også gældende for felterne, der desuden skal kunne filtrere på feltets værdi. En filtrering kan udtrykkes vha. en erklæring, der består af en protokol, evt. et eller flere felter, en operator og evt. en værdi. Felterne er til stede, hvis man ønsker at filtrere på nogle af de felter, som en given protokol indeholder. Ellers filtreres på protokollen. Værdien er til stede i erklæringen, hvis man ønsker at filtrere på værdien af et felt. Det betyder, at værdien kun kan være til stede, hvis der er defineret et felt i erklæringen.

På baggrund af ovenstående betragtninger ses det, at et filter består af et udtryk, hvor en eller flere erklæringer er sammensat med de logiske operatorer, `AND` og `OR`. Et filter kan således udtrykkes ved følgende BNF:

$$
\begin{aligned}
Filter &::= Expression \\
Expression &::= BinaryExpression | Statement \\
BinaryExpression &::= Expression\ \mathrm{AND}\ Expression \\
&\quad | Expression\ \mathrm{OR}\ Expression \\
Statement &::= \mathrm{ProtocolName}, [FieldList]\ , \mathrm{Operator}, [\mathrm{Value}] \\
FieldList &::= \mathrm{FieldName}, [FieldList] \\
Operator &::= \mathrm{PRESENT} \,|\, \mathrm{NOTPRESENT} \,|\, = \,|\, \neq \,|\, > \,|\, < \\
&\quad | \geq | \leq | \mathrm{CONTAINS} \,|\, \mathrm{MATCH}
\end{aligned}
$$

Det ses, at et filter kan opbygges som en træstruktur bestående af udtryk og erklæringer. Denne træstruktur kan være vilkårlig dyb, da ovenstående BNF notation har rekursive felter. Genereres et filter med ovenstående struktur, kan filtret evaluere en pakke vha. et design pattern, visitor (boks 3), idet filtret lader pakken besøge hvert udtryk i træet. Hvert udtryk i filtret evaluerer pakken i forhold til udtrykket og giver et boolsk resultat afhængig af sammenhængen mellem pakken og filtret.

3.5 Integration af script sprog

I afsnit 2.5 blev der argumenteret for, at Ruby er et velegnet script sprog til at interagere med netværket, da det opfyldte dels brugerkravene dels de tekniske krav til sproget. I dette afsnit ses der på, hvordan integrationen af Ruby i frameworket

[2]Taget fra en sniffing session, hvor trafikken har været typisk trafik på en hjemmecomputer, og hvor 830 pakker blev sniffet.

Boks 3: Visitor

Repræsenterer en operation, der skal udføres på en given objekt struktur. Alle elementer i strukturen bliver „besøgt" og operationen udføres alle steder. Dette kan give et resultat (som det er tilfældet her), som kan bruges øverst i hierarkiet. Dette design pattern er nyttigt, når man har en struktur, der deler en egenskab. En fremtidig ændring er ligeledes simpel, da man blot ændrer visitor operationen [3, s. 331ff].

kan designes. Der ses på, hvordan de øvrige moduler i systemet kan benyttes til at hjælpe brugeren, og hvordan kommunikationen mellem Ruby sproget og frameworket kan realiseres på en effektiv måde. I afsnit 4.5 ses der på de mere tekniske detaljer.

Generelt er der brug for at kunne udføre følgende operationer fra et Ruby script i frameworket:

- Sende en pakke
- Modtage en pakke
- Starte sniffingen
- Stoppe sniffingen

Afsendelsen af en pakke er i form af en række bytes, som protokolanalysatoren har ansvar for at sende ud på netværket. Modtagelsen af en pakke kan være, at man ønsker den næste pakke på netværket eller at man ønsker den næste pakke, der overholder et defineret filter. Strukturen af en pakke (en pakke til afsendelse eller modtagelse) er som udgangspunkt blot rå bytes. Men det er muligt at lave en bedre struktur ud fra oplysninger i de øvrige moduler i frameworket. Et sådant hjælpebibliotek behandles i det følgende afsnit. I afsnit 3.5.2 ses der på, hvordan operationerne overhovedet kan udføres.

3.5.1 Generering af hjælpebibliotek

Fra protokoldatabasemodulet kendes de protokoller, som er defineret i frameworket (i NetPDL). Der kan desuden skaffes oplysninger om de mulige felter i protokollerne og disses datatyper. Denne viden kan udnyttes til at skabe et bibliotek af Ruby funktioner, som kan benyttes, når Ruby scripts defineres i frameworket.

Når en pakke skal sendes til netværket, skal den i sidste ende være rå bytes. Men Ruby biblioteket kan indeholde funktioner, der abstraherer fra dette. Hjælpebiblioteket kan indeholde klasser, der hver især repræsenterer en protokol. En sådan protokol kan tilføjes felter, og når den er færdigdefineret kan protokollens rå bytes tilføjes pakkens rå bytes. Felter kan tilføjes vha. funktioner, som benytter viden om feltets datatype til at generere rå bytes, der svarer til den ønskede værdi

for feltet. Hvis et felt fx er et heltal på 4 bytes, kan en hjælpefunktion tage et heltal som input og generere 4 bytes og tilføje disse til en protokols rå bytes. En pakke kan altså i et Ruby script opbygges med et funktionskald pr. felt, der skal tilføjes pakken (i den rigtige rækkefølge).

Det skal bemærkes, at biblioteket ikke kan håndtere manglende felter eller protokoller automatisk, da protokoldatabasen ikke kender felters placering eller protokollers længde, da disse kan være forskellige fra pakke til pakke. Det er således brugerens ansvar at konstruere en korrekt udformet pakke. Derfor kan biblioteket også give mulighed for at tilføje de rå bytes direkte uden at bruge de abstraherede funktioner. Dette er fx en fordel, hvis man har en pakke liggende i en datafil, som man henter ind fra sit Ruby script.

Nedenstående viser et simpelt eksempel på afsendelsen af en pakke. Her sendes en pakke kun bestående af ethernet protokollen. Hjælpebibliotekets funktioner benyttes.

Eksemepel 3.1: Afsendelse af en pakke til netværket

```
startSniffing
p = SendPacket.new
ethernetData = P_ethernet.new
ethernetData.F_Dst(0x010203040506)
ethernetData.F_Src(0x0708090a0b0c)
ethernetData.F_Type(0x0800)
p.add(ethernetData)
sendPacket(p)
stopSniffing
```

Hjælpebiblioteket kan også benyttes, når der modtages en pakke fra netværket. Biblioteket kan definere en klasse, som en modtaget pakke er en instans af. Klassen kan indeholde funktioner til at hente de relevante data ud med. I eksemplet ovenfor blev en ethernet pakke sendt til netværket. Denne pakke ville kunne modtages og benyttes i Ruby scriptet vha. hjælpebiblioteket med følgende eksempel:

Eksemepel 3.2: Modtagelse af en pakke fra netværket

```
startSniffing
p = receivePacket("ethernet")
stopSniffing
puts p["ethernet"]["dst"].value
puts p["ethernet"]["src"].value
puts p["ethernet"]["type"].value
```

Dette giver outputtet (hvis den modtagne pakke var magen til den afsendte pakke i første eksempel):

```
010203-040506
070809-0a0b0c
2048
```

Bemærk at de data, der er gemt i `value`, er den streng, der er defineret i NetPDL's visualiseringsdel, dvs. man får samme værdi, som man ser på skærmen (deraf fås fx bindestregen i Ethernet adresserne i eksemplet, da denne er defineret i den tilhørende NetPDL definition). Man kan overveje en tilføjelse, så man også kan få de rå bytes for feltet, fx vha. `rawValue`-variabel.

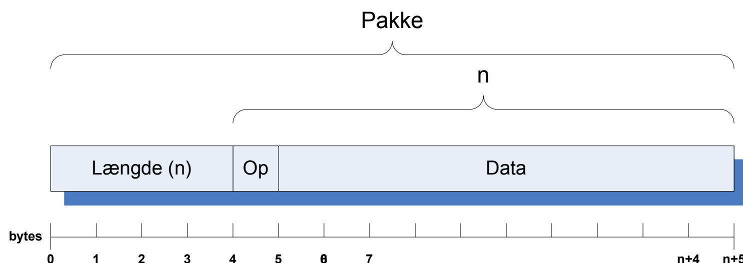
3.5.2 Kommunikation mellem framework og Ruby scripts

I afsnit 2.5 blev der givet et eksempel på en mulig kommunikation mellem framework og Ruby scripts, idet eksterne datafiler kan bruges til formålet. Denne metode lider dog af nogle problemer, idet en sådan fil skal opdages af de kommunikerende parter, og der kan være problemer med fx navngivning og placering af filerne. En bedre løsning er at oprette en intern forbindelse mellem parterne, hvor der kan sendes data frem og tilbage. Dataene kan sendes vha. TCP protokollen, da forbindelsen dermed bliver pålidelig og desuden understøttes denne protokol i både Ruby og .NET.

Frameworket kan ved start af et script lytte på en forbindelse fra scriptet. Når en sådan forbindelse oprettes beholdes den så længe scriptet kører, så begge parter kan sende beskeder. Men kun denne ene forbindelse accepteres i frameworket. Når scriptet terminerer, lukker frameworket forbindelsen igen.

Denne løsning opfylder kravene til kommunikationen, idet data kan sendes asynkront og persistent på TCP forbindelsen. Ulempen ved løsningen er, at alle data, der skal sendes, skal serialiseres inden. Dette er dog ikke et problem her, da dataene i forvejen skal serialiseres for at forberede dem til netværket.

For at parterne kan kommunikere sammen, må de være enige om den benyttede protokol til kommunikationen (ovenpå TCP). Figur 3.6 viser denne protokol. De første 4 bytes angiver længden af den efterfølgende data, dvs. af hele pakken minus de fire bytes til længdeangivelsen. Den 5. byte er en værdi, der beskriver de data, der er indholdet i resten af pakken. Hvis pakken er en forespørgsel, angiver den 5. byte den operation, der ønskes udført, se tabel 3.1. Er pakken et svar, indeholder den 5. byte en fejl kode, hvor 0 betyder, at operationen var succesfuld. I et svar, hvor fejlkoden ikke er 0, indeholder pakkens data en ASCII fejlbeskrivelse.



Figur 3.6: Protokol til kommunikation mellem framework og Ruby scripts.

Kode	Operation	Data
1	Start sniffing	Ingen
2	Stop sniffing	Ingen
3	Send pakke	Pakken som rå bytes
4	Modtage filter pakke	Filternavn som ASCII
5	Modtage enhver pakke	Ingen

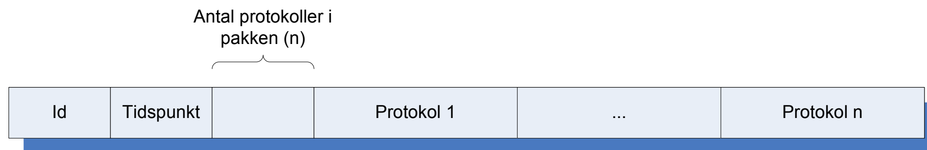
Tabel 3.1: Operationskoder i forespørgsler.

Der sendes altid et svar på en forespørgsel. Kun forespørgsler med operationerne 4 og 5 (modtage pakke) indeholder data, hvis svaret er uden fejlkode. De

indeholdte data er her den modtagede pakke. Formatet, som denne pakke sendes i, beskrives i det følgende afsnit.

Format af modtaget pakke

Frameworket må sende en del information med, når et Ruby script skal modtage en pakke. Ellers kan Ruby scriptet ikke tilgå pakkens data på en intuitiv måde. Relevant data for pakken sendes derfor med, fx protokol/feltnavne. Figur 3.7-3.9 viser formatet af hhv. en pakke, en protokol, et felt og en liste af felter.



Figur 3.7: Formatet af en pakke.

Figur 3.8: Formatet af en protokol.

Figur 3.9: Formatet af et felt.

Figur 3.10: Formatet af en feltliste.

En pakke består af en eller flere protokoller, der består af en eller flere felter. Et felt er en samling af feltlister. En feltliste er en samling felter i protokollen med

hver sin værdi, men med samme navn (da et unikt felt kan optræde flere gange i den samme protokol). Idet en feltliste består af felter (udover værdierne), der igen er defineret som feltlister, er opbygningen rekursiv, hvilket afspejler, at de indlejrede felter i en protokol kan have en arbitrær dybde.

Alle strenge (navne og værdier af felter) specificeres som nul terminerede strenge, så de kan have en vilkårlig længde. Værdien angives som den læselige værdi som defineret i NetPDL specifikationen. Dermed bliver der overensstemmelse mellem det, brugeren ser i sniffingen og det, der kan fås ud af Ruby datastrukturen.

Med denne dataopbygning kan Ruby hjælpebiblioteket opbygge en datastruktur af pakken, så de enkelte data kan hentes ud som eksempel 3.2 ovenfor viser.

3.6 Interface til protokolanalysator

På figur 3.1, s. 36, ses det, at en del af frameworket er at definere et interface til protokolanalysatoren, så denne har mulighed for at stille de protokolspecifikke dele til rådighed for frameworket. I dette afsnit ses der på, hvad dette interface må indeholde.

Interfacet skal være generelt opbygget forstået på den måde, at det indeholder dele, der er fælles for alle protokolanalysatorer, men hvor de enkelte dele skal specificeres for hver protokolanalysator. Konkret skal interfacet give frameworket mulighed for at få følgende dele fra protokolanalysatoren:

1. Interface til netværksenheder, der kan sniffe data fra netværket.
2. Interface til netværksenheder, der kan interagere med netværket.
3. Beliggenheden og navnet på NetPDL beskrivelses filen.

De to første punkter giver frameworket mulighed for den egentlige kommunikation med det for frameworket ukendte netværk. Der stilles nogle krav til disse interfaces for at frameworket kan udføre de nødvendige operationer. For et sniffinginterface (punkt 1) er det følgende operationer:

1. Oplysning om navn på interface, der kan bruges i præsentationen af interfacet for brugeren.
2. Sætte interfacespecifikke indstillinger.
3. Forbinde til interfacet.
4. Afbryde forbindelsen til interfacet.
5. Modtage underretning om en ny pakke på netværket.
6. Starte sniffing på netværket.
7. Stoppe sniffing på netværket.
8. Nulstille tiden, der påsættes pakkerne.

For et interaktionsinterface (punkt 2) er det følgende operationer:

1. Oplysning om navn på interface, der kan bruges i præsentationen af interfacet for brugeren.

2. Sætte interfacespecifikke indstillinger.
3. Forbinde til interfacet.
4. Afbryde forbindelsen til interfacet.
5. Sende en række bytes ud på netværket.

Nogle netværksenheder vil pr. automatik være koblet på netværket, men andre skal gøre noget aktivt, før de er klar til fx at starte sniffingen. Derfor skal frameworket give mulighed for at koble til/fra netværket. Nogle netværksenheder kan have indstillinger, der skal sættes, inden kommunikationen med netværket kan iværksættes. Fx skal nogle netværksenheder vide, hvilken bitrate (hastighed) netværket kører med, så sniffingen/interaktionen sker ved den rigtige hastighed. I frameworkets præsentationslag kan man via interfacevælgermodulet angive disse indstillinger. Dette sker gennem de ovenstående interfaces, idet frameworket ikke i sig selv kan vide, hvilke indstillinger, der skal være tilgængelige. Når en protokolanalysators sniffing eller interaktionsnetværksinterfaces får besked fra frameworket om, at brugeren har forespurgt indstillinger, kan protokolanalysatoren præsentere de mulige indstillinger på den ønskede form, fx vha. et „indstillingsvindue".

Ovenstående har beskrevet, hvordan frameworket får adgang til de protokolspecifikke dele i protokolanalysatoren. Men protokolanalysatoren har også brug for at kunne bruge frameworkets services. Dette kunne foretages vha. plug-ins til frameworket, men de services, som frameworket stiller til rådighed for plug-ins (se afsnit 3.7), er ikke de samme, som protokolanalysatoren har brug for. Derfor stiller frameworket nogle services specielt til rådighed for protokolanalysatoren. Nogle er dog også til rådighed for plug-ins. Disse services er:

1. Definerer ikke-standardkolonner i pakkevisningen.
2. Tilføje kobling mellem definerede protokoller (i NetPDL) og ikke-standardkolonnerne i pakkevisningen. Dette gør frameworket i stand til at udfylde kolonnerne med feltdata fra pakkens øverste protokol.
3. Abonnere på typer af pakker, som protokolanalysatoren er interesseret i. Dette er brugbart for protokolanalysatoren, når den skal sætte pakker sammen, der eksisterer over flere pakker, jf. afsnit 2.4.4.
4. Definerer nye filtre.
5. Underrette om nye pakker på netværket efter en sniffing er startet.

Med disse services er en protokolanalysator i stand til at opfylde kravene til denne. Det skal dog bemærkes, at der ikke er nogen krav fra frameworkets side om, at disse services skal benyttes. En meget simpel protokolanalysator kan undlade at bruge services og dermed holde udviklingsarbejdet på et minimum til opfyldelse af kravspecifikationen.

3.7 Interface til plugin

Når der skal tilføjes funktionalitet til frameworket i form af plug-ins, sker det ved, at frameworket stiller sine services til rådighed for plug-ins. I det foregående er

Boks 4: Adapter

Konverterer interfacet af en klasse til et andet interface, som forventes af andre klasser. Herefter kan en klasse bruges et sted, hvor den ellers er inkompatibel. I dette tilfælde kan man bruge en adapter til at gøre en unmanaged klasse managed, så klassen kan benyttes i .NET. Dette design pattern kan også anvendes på andre dele end klasser, idet fx en metode kan kaldes af en adapter metode, der „konverterer" resultatet til det ønskede [3, 139ff].

frameworkets moduler og deres services gennemgået. Denne modulopbygning har den fordel, at det gør muligheden for plug-ins lettere.

For hvert modul skal det blot afgøres, hvilke services, der skal være tilgængelige for plug-ins. Dette gøres mest praktisk på implementeringstidspunktet. I forbindelse med dette valg skal følgende overvejes:

- Har brugen af en service konsekvenser (bivirkninger) internt i frameworket, som plug-in implementatoren ikke har overblik over? I givet fald, bør denne service ikke være tilgængelig.
- Bidrager servicen overhovedet med brugbar funktionalitet for plug-in implementatoren?
- Er der sikkerhedsmæssige betænkeligheder ved at udstille en given service?

Principielt har services, der ændrer globale variabler bivirkninger. Dette behøver dog ikke at betyde, at de ikke kan udstilles til plug-ins. Det skal bare være intuitivt klart for plug-in implementatoren, at der er de pågældende bivirkninger. Services, der kun læser variabler, er som udgangspunkt helt sikre at udstille.

I det tilfælde det afgøres, at en service ikke bør udstilles af en de ovenstående grunde, kan man overveje, om det kan lade sig gøre at udstille blot noget af servicen. Man kan lave en adapter (design pattern), se boks 4 til servicen, så kun den ønskede del af servicen er udstillet.

3.7.1 Plug-ins i C#

I forbindelse med at C# benyttes som udviklingssprog, er det specifikke dele af en klasse (udover klassen selv), der skal overvejes i forbindelse med at udstille services til plug-ins. Hver enkelt del kan fra plug-in implementatorens synspunkt ses som en service. De relevante dele er:

- Events.
- Properties. „get" delen af en property kan som udgangspunkt altid udstilles (hvis det er relevant), idet det er et eksempel på en service, der kun læser variabler.

- Methods. Her skal det undersøges nærmere, om metoden har bivirkninger.

Rent praktisk udstilles ovenstående identificerede dele vha. deres *access-modifiers*, idet kun dele, der er *public* eller *protected* kan ses udenfor den assembly, hvori de er defineret.

3.8 Konklusion på design af systemet

Kapitlet har gennemgået designet af det samlede system. Overordnet er systemet designet fleksibelt vha. moduler med hver deres ansvarsområde og udbud af services. Disse ansvarsområder og services er gennemgået i kapitlet.

Til designet af frameworket er valgt en trelagsopdeling, hvor præsentation, domæne og datakilde skilles ad. Hvert af disse lag indeholder moduler til at udføre lagets opgave.

Specielle dele af designet er beskrevet mere detaljeret. Pakkernes vej gennem systemet fra de kommer ind fra netværket som rå bytes til de ender på skærmen er gennemgået og det er blevet præsenteret, hvordan pakken struktur kan benyttes i en pakkedatabasestruktur, der gør filtrering effektiv. Denne databasestruktur giver mulighed for filtrering med køretiden $O(k)$ hvor k er dybden af filtret defineret ved antallet af felter, der filtreres på.

Integrationen af Ruby i frameworket kan lade sig gøre ved dels at give brugeren hjælp i form af et Ruby hjælpebibliotek dels at dedikere en kommunikationskanal til interaktion mellem framework og Ruby script. Den protokol, der benyttes til kommunikationen, er præsenteret.

I kapitlet er et interface til protokolanalysator hhv. plug-ins designet, og det er beskrevet, hvilke aspekter, der er vigtige i implementeringen af disse, så frameworket holder sin interne robusthed.

Undervejs i kapitlet er relevante design patterns introduceret i de sammenhænge, hvor de kan gøre nytte og effektivisere designet.

Dermed er der skabt et overordnet design med en struktur for applikationen. Der er tale om en fast men ikke komplet struktur for applikationen, da der skal være mulighed for ændringer af designet senere. Men strukturen kan benyttes i næste kapitel, hvor designet implementeres.

Kapitel 4

Implementering og teknisk test

Dette kapitel beskriver implementeringen af designet i C# til .NET platformen. Kapitlet indledes med en beskrivelse af den benyttede implementeringsmetode. Dette hænger sammen med, hvordan den tekniske test gennemføres, der derfor også bekrives her.

En beskrivelse af den egentlige implementering tager udgangspunkt i en beskrivelse af den generelle opbygning af frameworket. Herefter beskrives de ikke-trivielle dele af frameworket. Dernæst beskrives, hvordan frameworket bruges, dvs. det interface, der er mellem framework og protokolanalysator. I det efterfølgende afsnit beskrives, hvordan fleksibiliteten i frameworket er opbygget, så det giver mulighed for tilføjelse af plug-ins.

Kapitlet vil koncentrere sig om den overordnede opbygning, dvs. klassernes individuelle ansvar samt forholdet mellem forskellige klasser. Desuden beskrives det, hvordan designet er implementeret. Der gives ikke en detaljeret beskrivelse af koden, da det ikke synes relevant for rapporten – i stedet henvises til koden og kommentarer deri. Dog vil der blive redegjort for mere interessante specifikke problemstillinger i implementeringen.

Indholdsfortegnelse

4.1 Implementeringsmetode og strategi for teknisk test **56**
4.2 Generel opbygning af framework **57**
4.3 Protokoldatabase **59**
4.4 Pakkehåndtering og pakkedatabase **63**
4.5 Integration af Ruby **65**
4.6 Brug af frameworket **67**
4.7 Plug-ins **68**
4.8 Konklusion på implementering og teknisk test **69**

4.1 Implementeringsmetode og strategi for teknisk test

I dette afsnit beskrives strategien for at verificere, at applikationen virker korrekt. Denne strategi indeholder nemlig metoden, der benyttes til implementering.

Som beskrevet i afsnit 1.3.3 og 3.2 er Test-Driven Development (TDD) en oplagt metode at benytte i implementeringen. Generelt er filosofien bag TDD at skrive tests til applikationen, før man implementer den. Implementeringen udføres iterativt ved at gentage tre faser [1] [4]:

Red fase Der skrives test, der afspejler kravspecifikationen for applikationen. Testene gør, at applikationen ikke kan kompilere, idet der ikke er skrevet kode til applikationen endnu.

Green fase Applikationen implementeres med det ene formål, at de skrevne tests skal køre succesfyldt.

Refaktoreringsfase Koden refaktoreres så fx duplikeret kode elimineres.

De tests, der skrives, kan med fordel skrives vha. et xUnit framework og da .NET benyttes i implementeringen, er NUnit[1] et godt værktøj at benytte. xUnit er en familie af værktøjer til forskellige programmeringssprog, der letter realiseringen af automatiserede tests.

Fordelen ved at benytte TDD er, at man ved, at man, når ens tests forløber fejlfrit, har opfyldt kravspecifikationen. Derudover kan man efter ændringer i koden tjekke, at man ikke med sine ændringer har ødelagt andre dele af koden, idet man blot eksekverer alle sine tests igen. Metoden er velegnet i implementeringen af frameworket, idet der skrives tests for hvert modul, der tester modulet teknisk. Desuden skrives der en gruppe tests, der tester kommunikationen mellem modulerne.

Metoden er god, men kvaliteten af applikationen afhænger selvfølgelig af kvaliteten af ens tests. Det er vigtigt, at testene tester alle de muligheder, hvorpå en bruger kunne finde på at bruge applikationen. Dette indebærer bl.a., at testene skal være en god afspejling af kravspecifikationen.

Men metoden kan ikke bruges i alle tilfælde. Fx er det svært at skrive automatiserede tests til brugergrænsefladen, idet fx et klik på knap kan være svært at simulere realistisk. Dette aspekt af projektet testes derfor manuelt. Det samme gælder for multitrådede dele af applikationen, da en automatiseret test kan give forskellige resultater fra eksekvering til eksekvering. Denne del må i stedet testes manuelt, men da resultater her også vil variere, er det vigtigt, at designet af disse dele analyseres nøje inden implementering. Kapitel 3 har behandlet de vigtigste elementer indenfor dette område.

Det ligger udenfor dette projekt at beskrive TDD metoden i flere detaljer samt at beskrive de tests, der er benyttet i implementeringen – formålet her er blot at dokumentere implementeringsstrategien og teststrategien. Da projektets mål ligeledes er en prototype af et framework, der i sagens natur ikke er et færdigt produkt, ligger det derfor også uden for opgaven at teste applikationen fuldstændigt. Til gengæld er det en vigtig del af projektet at undersøge, hvorvidt frameworket overholder brugerkravene – dette vil der derfor blive set nærmere på i kapitel 6, hvor en konkret protokolanalysator præsenteres, der derfor kan benyttes som grundlag

[1] Værktøjet kan findes på `http://nunit.com` – tilgængelig 6/6-2007.

for en brugertest. Det er ligeledes vigtigt at se på fremtidige muligheder for frameworket, hvilket derfor behandles, når projektets resultater præsenteres og vurderes i konklusionen (kapitel 8).

4.2 Generel opbygning af framework

I designet af frameworket fremgår det, hvordan moduler hver især kan håndtere en mindre del af opgaven. Dette princip skal nu implementeres i konkrete C# klasser. Opbygningen og inddelingen af klasser afspejler i høj grad det modulære designet. Nogle moduler kan implementeres blot vha. én klasse, mens andre benytter en række klasser.

Overordnet opbygges frameworket af følgende assemblies:

Domain Implementerer domænelaget og alle dets moduler.

Presentation Implementerer præsentationslaget og alle dets moduler.

PaInterface Implementerer interfacet til protokolanalysatoren.

NetPDLwrapper Implementerer `NetPDL engine` modulet, der skaber en managed adgang til NetPDL værktøjerne. Modulet er i en selvstændig assembly, da den er skrevet i C++/CLI.

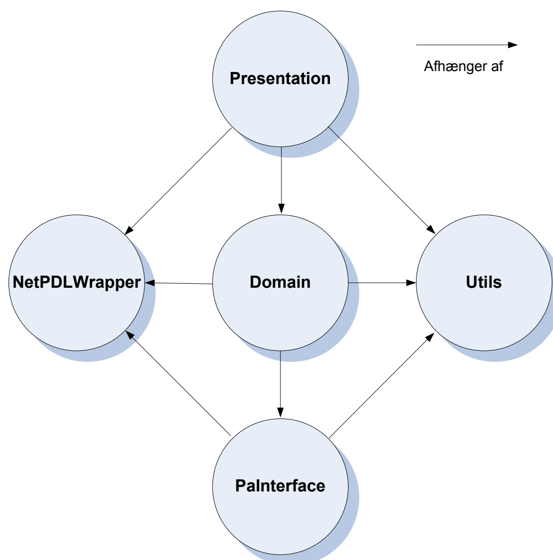
Utils Indeholder forskellige klasser med diverse små værktøjer, som flere af de andre moduler kan bruge. Det er bl.a. implementeringen af en hash mængde, forskellige værktøjer, der kan håndtere tid, en klasse til håndtering af meget store tal osv.

Figur 4.1 viser afhængighederne mellem de enkelte assemblies. Afhængighederne afspejler lagopdelingen, idet ingen afhænger af præsentationslaget (det øverste lag), mens alle afhænger af protokolanalysator interfaces (det nederste lag). Alle er desuden afhængige af de assemblies, der ligger udenfor lagopdelingen, da dette er generelle værktøjer, alle kan have glæde af, mens disse værktøjer ikke afhænger af assemblies i lagopdelingen, da de er generelle og ikke må afhænge af sammenhænge. Figuren viser, at implementeringen afspejler det ønskede i designet.

Modulerne beskrevet i designkapitlet er implementeret i de enkelte assemblies, hvor hvert modul består af en eller flere klasser. I appendiks E er givet en oversigt over alle de fremstillede klasser samt hvilket moduler, de er med til at implementere. De givne klassenavne vil blive brugt i den mere detaljerede fremstilling af implementeringen i kapitlets resterende afsnit.

4.2.1 Implementering af kommunikation mellem moduler

I afsnit 3.2.1 blev det beskrevet, hvordan en mediator, se boks 1 kombineret med en singleton, se boks 2, kan benyttes til at skabe kommunikation mellem moduler/klasser uden at få et indviklet og uoverskueligt netværk af referencer. Der er en mediator i domænelaget og en i præsentationslaget. For at gøre klasserne mindre og mere overskuelige, er de enkelte mediatorer delt i to. En klasse tager sig



Figur 4.1: Afhængigheder mellem assemblies.

af forespørgsler og en tager sig af at underrette[2] om hændelser. Dette giver fire klasser i alt, som er:

DomainRequestHandler Tager sig af forespørgsler til eller i domænelaget.

DomainController Tager sig af at underrette om hændelser i domænelaget.

PresentationRequestHandler Tager sig af forespørgsler til eller i præsentationslaget.

PresentationController Tager sig af at underrette om hændelser i præsentationslaget.

`DomainController` og `PresentationController` har tilknyttet en række events, som interesserede moduler/klasser kan abonnere på. Der er metoder i klasserne til at sætte underretningerne igang. Disse metoder kan bruges af de klasser, hvor hændelserne reelt opstår. Skulle løsningen være helt stringent, burde de relevante klasser selv have events, som hhv. `DomainController` og `PresentationController` abonnerer på. Når hændelserne så sker, vil de to klasser sende dem videre. Men dette giver et ekstra unødvendigt trin i kommunikationen med reduceret overskuelighed til følge.

Opbygningen sker i tråd med et design pattern, Observer, idet de abonnerende klasser fungerer som observertører og `DomainController` og `PresentationController` fungerer som subjekter, jf boks 5.

`DomainRequestHandler` og `PresentationRequestHandler` er andre lags eller modulers indgang til funktionaliteten i de to lag (domæne og præsentation). Det

[2] I `C#` kan der oprettes „events“, som interesserede kan abonnere på vha. en „delegate“. Disse abonnenter bliver „notified“, når en hændelser indtræffer. Som oversættelse benyttes i rapporten „hændelse“ om „event“ og „underette“/„underretning“ om „notify“/„notification“.

Boks 5: Observer

Observer definerer subjekter (Subjects), som interesserede observatører (Observers) kan abonnere på. Subjekterne underretter observatørerne om ændringer i subjekterne, så observatørerne kan agere ud fra ændringerne. Dette design pattern er ideen bag event systemet i C# [3, s. 293ff].

er altså mulighederne her, der afgør mulighederne for ydre lag og plug-ins til frameworket.

4.3 Protokoldatabase

I dette afsnit beskrives realiseringen af protokoldatabasen. Problemstillingen er hovedsageligt et spørgsmål om at give en abstraktion af den (unmanaged) struktur, som man kan få fra `NetPDLProtoDB`, se afsnit 2.4.5, så strukturen bliver lettere at anvende i andre dele af systemet. Abstraktionen kan genereres vha. et design pattern, Adapter, se boks 4, s. 53.

Fra `NetPDLProtoDB` fås en række `C` strukturer, der udgør de forskellige dele af protokoldatabasen. De enkelte strukturer oversættes til et objekt, der har de samme egenskaber som deres tilhørende struktur. Enkelte objekter tilføjer egenskaber til de allerede eksisterende. Tabel 4.1 viser relationerne mellem de unmanaged `C` strukturer og de managed klasser.

Unmanaged struktur	Managed klasse
_nbNetPDLDatabase	ProtocolDB
_nbNetPDLElementProto	ProtocolDefinition
_nbNetPDLElementField<X>[3]	FieldDefinition
_nbNetPDLElementBlock	FieldDefinition

Tabel 4.1: Relation mellem unmanaged strukturer og managed klasser.

Som tabellen viser, definerer `NetPDLProtoDB` en struktur for hver type felt. I stedet defininеres her én klasse, der håndterer alle felttyper. Til gengæld har feltet en type tilknyttet via enumeratoren, `FieldType`. `FieldType` defininerer en konstant for hver af de af NetPDL defininerede felttyper, men definerer derudover en konstant mere: `Container`, der benyttes, når et felt udelukkende eksisterer for at indeholde andre felter og ikke har en værdi.

De nye klasser kan bruges som deres korresponderende `C` strukturer, dog tilføjer klassen `FieldDefinition` et felt, `Length`, der angiver længden af feltet i bit, hvis det er muligt. Værdien sættes mere intelligent end det korresponderende `length` felt i `C` strukturen. Det samme gør sig gældende for `GroupLength`, der angiver

[3] <X> er en af følgende: Base, Bit, Fixed, Line, Padding, Plugin, TokenEnded, TokenWrapped, Variable svarende til de forskellige typer af felter.

længden af de enkelte grupper i visningen af feltets værdi. Disse værdier er specielt nyttige, når der skal defineres filtre, hvor formatet af en angivet værdi til et felt skal valideres. Hvis fx et felt i en protokol er 1 byte (8 bit) og vises som et heltal, kan feltet maksimalt have værdien 255. Når et filter defineres skal der opstå fejl, hvis brugeren forsøger at filtrere på pakker, hvor feltet er 1000 (decimalt), da sådanne pakker aldrig vil forekomme. Til at validere '1000' kan man benytte feltets længde angivelse.

Der er lavet en anden udvidelse til `FieldDefinition`, idet det kan være nyttigt at vide, om et felt kan gentages eller være valgfri i en pakke, hvilket i NetPDL vil fremgå ved, at definitionen er i en løkkekonstruktion (hvis den kan gentages) eller i en betingelseskonstruktion (hvis den er valgfri). En `FieldDefinition` får derfor sat en variable, der angiver tilstanden af feltet (`NONE`, `REPEATED`, `CONDITIONAL` eller en kombination af de to sidste).

Med denne nye struktur er `ProtokolDatabase` modulet i stand til at stille services til rådighed, der giver tilstrækkelige oplysninger om de definerede protokoller. Modulet er specielt brugbart, når filtre skal defineres. Denne del ses der derfor nærmere på i det følgende.

4.3.1 Filtrering

Afsnit 3.4.3 gav en BNF notation for strukturen af et filter. I dette afsnit ses der på, hvordan denne notation kan implementeres effektivt.

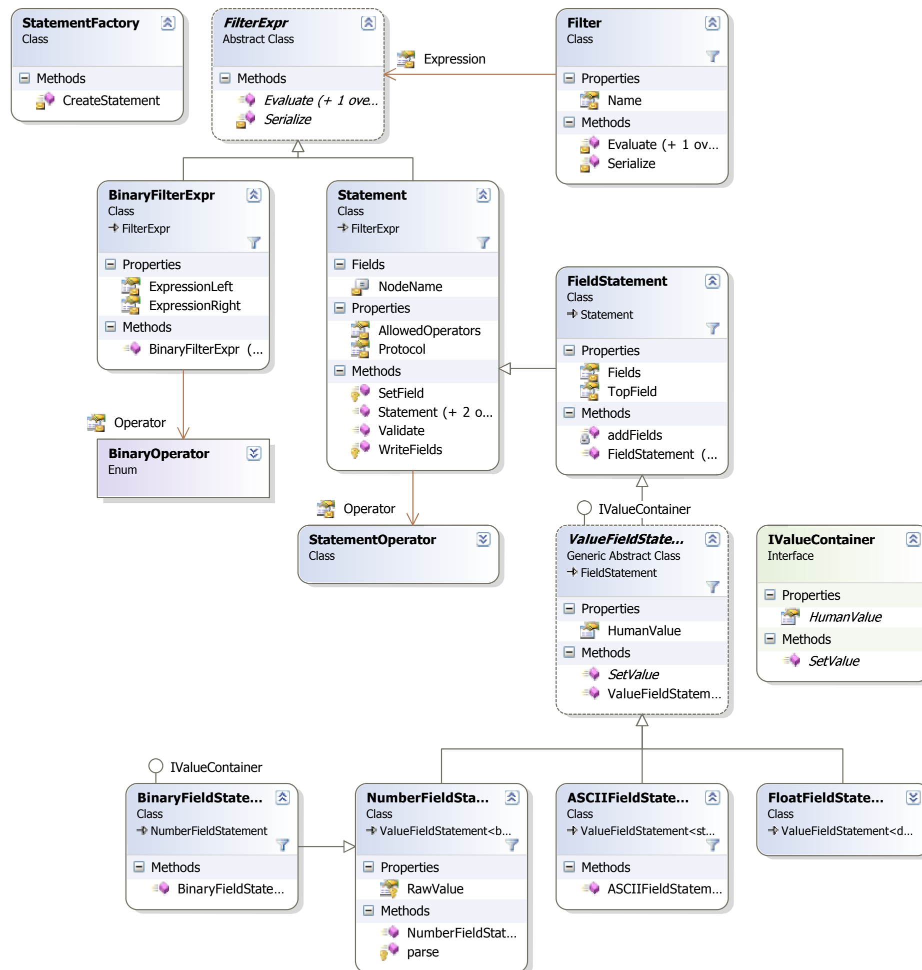
Strukturen givet af BNF notationen kan realiseres ved at oprette en klasse for hver udtryk/erklæring. Et klassehierarki for denne struktur ses på figur 4.2.

Figuren viser, at erklæringer er delt i flere klasser. Dette skyldes, at en erklæring skal kunne validere brugerinput (værdien) og ikke mindst kunne undersøge en pakke for, om den passer på filtret. Afhængig af det valgte felt (dens type) vil sådanne operationer være vidt forskellige. Der er fx stor forskel på sammenligning af tekststrenge og heltalsværdier. Ved heltalsværdier skal der fx være tjek af tallets størrelse. Tekststrenge understøtter også flere af operatorerne (`CONTAINS` og `MATCH`)[4]. Tabel 4.2 viser sammenhængen mellem felttyperne og de definerede klasser i figuren.

Tabellen giver ikke det fulde billede. For valget af erklæringsklasse er afhængig af valget af operator. Vælges en af operatorerne `PRESENT` eller `NOTPRESENT` benyttes klassen `FieldStatement`, da der ikke skal defineres en værdi for erklæringen (`PRESENT` og `NOTPRESENT` er unære operatorer). Endelig hvis der ikke er angivet et felt (og der derfor ikke er en felttype) benyttes `Statement` erklæringsklassen, da denne repræsenterer en erklæring kun bestående af en protokol og en (unær) operator.

De forskellige klasser har metoder til de ønskede operationer. `Validate` undersøger om en given `String` kan bruges som værdi til erklæringen. `Filter` har en metode, `Evaluate`, der undersøger, om en en `Collection` af pakker (fx fra pakkedatabasen) passer på filtret. Dette kald videresendes til det udtryk, som filtret repræsenterer og derfra ned gennem træet, så de pakker, der overholder filtret, findes. Et kald til `Serialize` på et `Filter` objekt sendes ligeledes ned gennem træet, så alle dele bliver serialiseret (vha. `XML`). Erklæringer har ansvaret for, udover at udføre operationer, der kommer fra `Filter`, at afgøre, hvilke operatorer,

[4]Bemærk at sammenligningsoperatorerne godt kan understøttes af tekststrenge, da de kan sammenlignes leksikografisk.



Figur 4.2: Klassehierarki for filtrering.

der er tilladte for den pågældende erklæring. Desuden indeholder klasserne oplysninger om protokollen og operatoren (i `Statement`), felterne (i `FieldStatement` og værdien (i `ValueFieldStatement`). Der findes tre typer erklæringer, der tager sig af felter, hvor en værdi tilknyttes: `NumberFieldStatement`, der tager sig af numeriske felter (både decimale og hexadecimale), `ASCIIFieldStatement`, der tager sig af alle felter med tekststrenge og `BinaryFieldStatement`, der tager sig af alle binære felter, jf. tabel 4.2.

Interne datatyper til heltal i C# er op til 96 bit (`Decimal` klassen). Men felterne i en protokol kan være større, idet NetPDL ikke definerer en maksimal størrelse. Derfor benyttes en klasse `BigInteger` til at håndtere de numeriske værdier[5].

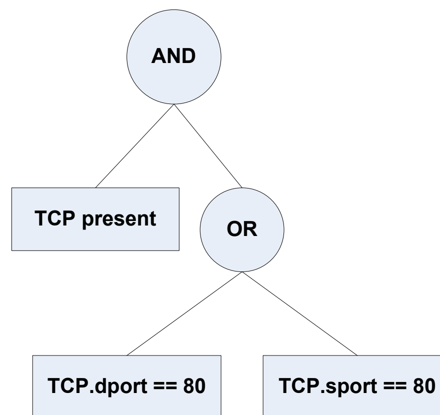
Det forgående har været en abstrakt teoretisk beskrivelse af filtreringen. I det

[5] `BigInteger` er hentet fra `http://www.codeproject.com/csharp/biginteger.asp` – tilgængelig 1/6-2007, der dog er reduceret i størrelse, da der var meget unødig funktionalitet (specielt i forbindelse med beregning af primtal).

NetPDL felttype	Erklæringsklasse
HEX	NumberFieldStatement
HEXNOX	NumberFieldStatement
DEC	NumberFieldStatement
BIN	BinaryFieldStatement
FLOAT	FloatFieldStatement
DOUBLE	FloatFieldStatement
ASCII	ASCIIFieldStatement
HIDE	FieldStatement
CONTAINER	FieldStatement

Tabel 4.2: Sammenhæng mellem felttyper og definerede erklæringsklasser.

følgende gennemgås et eksempel, der viser, hvordan filtreringen foregår i praksis. Figur 4.3 viser dette eksempel på et filter og det træ af erklæringer, som filtret er repræsenteret ved.



Figur 4.3: Eksempel på filter udtryk.

På figuren ses, at filtret består af tre erklæringer sat sammen med to logiske operatorer. Den første erklæring, `TCP present`, er en `Statement` med en protokol (`TCP`) og en operator (`PRESENT`). De to sidste erklæringer (`TCP.dport == 80` og `TCP.sport == 80`) er `NumberFieltStatements`, idet både `dport` og `sport` er af felttypen `DEC`, jf. tabel 4.2. Værdien i begge erklæringer er angivet til 80, som er en gyldig heltalsværdi, som passer på erklæringsklassen. Hvis dette filter sættes på en mængde af pakker vha. `Evaluate` metoden, vil pakker, hvor protokollen `TCP` er repræsenteret og hvor afsender (`sport`) eller modtager porten (`dport`) er 80, være i den resulterende mængde af pakker. Dette er fx alle HTTP pakker[6]. Evalueringen af filtret på en mængde vil først evaluere `TCP present`. Fællesmængden af den resulterende mængde pakker og den resulterende mængde fra evalueringen af den anden gren af træet vil være resultatet af filtrets evaluering. Denne anden gren vil give foreningsmængden af de pakker, der benytter afsenderporten 80 og de

[6]Ville man udelukkende have filtreret HTTP pakker, kunne man have lavet et simpelt filter, `HTTP present`, i stedet.

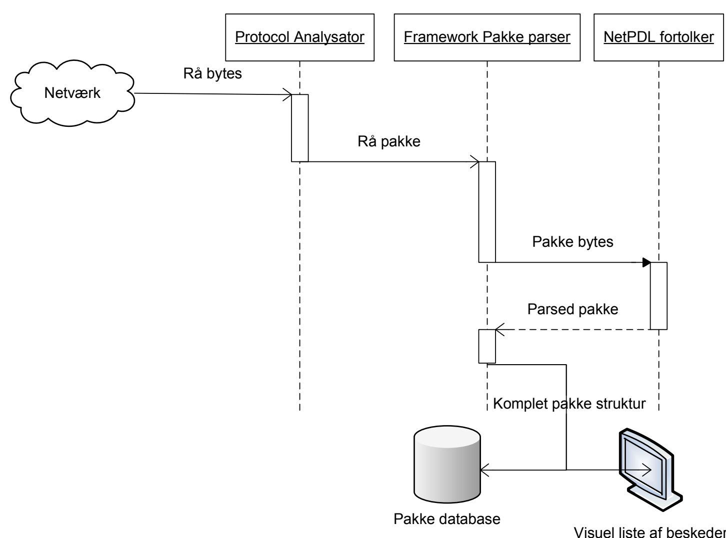
pakker, der benytter modtagerporten 80. Evalueringen vil altså foregå rekursivt ned gennem træet.

Den, der bygger filtret, er ansvarlig for at give den associativitet, som brugeren måtte forvente, fx kan træet i figur 4.3 være resultatet af følgende udtryk, bemærk paranteserne:

$$\mathrm{TCP}\,.\,\mathrm{PRESENT}\,\&\&(\mathrm{TCP}\,.\,\mathrm{dport} == 80 \| \,\mathrm{TCP}\,.\,\mathrm{sport} == 80)$$

4.4 Pakkehåndtering og pakkedatabase

Frameworket modtager en pakke som et array af bytes. Dette array skal parses af den tilknyttede NetPDL engine („NetBee“). Systemet håndterer pakken som beskrevet i afsnit 3.4. Dette udføres af en række objekter, der svarer til de enkelte elementer i figur 3.5 s. 44. Figur 4.4 viser det tilsvarende sekvens diagram over bevægelsen af en pakke i systemet.



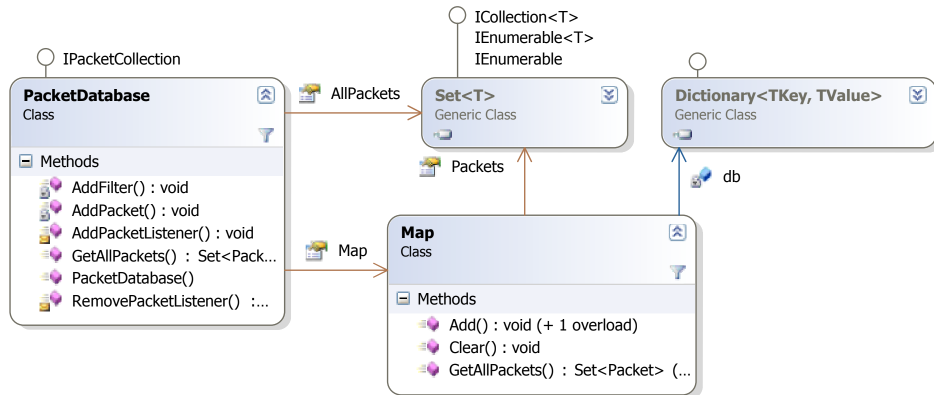
Figur 4.4: Sekvens diagram over en pakkes vej i systemet.

Det interessante er nu, hvordan den endelige pakke håndteres i pakkedatabasen hhv. på skærmen. Især skal de være i stand til at håndtere store datamængder, da det må formodes, at der kan forekomme situationer, hvor der er ankommet mange pakker. Disse to moduler ses der derfor nærmere på i de følgende afsnit. Men også brugen af den tilknyttede NetPDL engine kræver lidt forklaring.

4.4.1 Store datamængder i pakkedatabasen

I afsnit 3.4.2 blev givet en struktur til at håndtere pakkerne på en måde, så der var hurtig adgang til dem. I dette afsnit beskrives, hvordan den beskrevne relation kan implementeres i praksis. Klassen `Map` implementerer den omtalte relation. Klassen

indeholder to datastrukturer: En mængde, `Set`, indeholdende pakkerne (`Packet`) og en hashtabel, `Dictionary`, hvor felt/protokolnavne mappes til en ny relation (`Map`), se figur 4.5.



Figur 4.5: Klasse diagram over pakkedatabasen.

Som figuren viser, indeholder klasserne metoder til almindelige operationer på databasen. En pakke tilføjes til databasen, når den opdages i systemet ved at protokoller/felter gennemløbes rekursivt og tilføjes den rigtige relation (et `Map` objekt). `Map` objekterne oprettes kun, når der er brug for det. Det er muligt at hente en mængde pakker ud, der indeholder en given protokol og en liste af felter.

`PacketDatabase` indeholder desuden metoder til at tilføje eller fjerne filtre fra pakkerne. `AddFilter` tager et filter som parameter og evaluerer pakkedatabasen med det filter. De pakker, filtret matcher, får filtret føjet til deres liste over filtre. Når et filter fjernes fra pakkedatabasen bliver filtret så fjernet fra listen over filtre i de enkelte pakker. På den måde kan visningsmodulet bruge listen over filtre for hver enkelt pakke til at afgøre, om en pakke skal vises.

4.4.2 Store datamængder i pakkevisningen

`ListView`, der er en del `.NET` frameworket, er ideel til at præsentere pakkerne i en liste. `MessageViewList` udvider denne klasse og indbygger funktionalitet til at vise en liste af pakker. `ListViewItem` benyttes til at repræsentere de enkelte pakker. Klassen har en egenskab, `Tag`, hvor selve pakken kan gemmes. Når et element i listen vælges kan man gennem denne egenskab finde ud af, hvilken pakke der blev valgt (efter `Tag` er blevet castet til `Packet`). Når en pakke opdages i systemet, kan der i princippet oprettes en `ListViewItem` som tilføjes listen. Men en tilføjelse til listen inkluderer et stort overhead for listen, idet der skal udlægges et ny layout. Desuden har hver instans af `ListViewItem` et vist ressource forbrug i tillæg til det, en pakke i forvejen bruger. Af denne grund kan et stort antal pakker ankommende hurtigt efter hinanden give systemet problemer med at vise pakkerne hurtigt nok.

Specielt virker det overflødigt at kende til alle pakker grafisk, når listen højest kan vise 20-30 pakker i ét skærmbillede (som så kan scrolle, hvis der er flere pakker). Til at løse dette problem kan man benytte den funktion, som `ListView` understøtter: Virtuelle enheder. `MessageViewList` har ansvaret for at holde styr på, hvor mange enheder (pakker) der er i listen på alle tidspunkter, da dette af-

gør, hvor langt ListView kan scrolles. Men kun de enheder, der er synlige, er kendt. Når billedet scroller, anmoder ListView MessageViewList om de nødvendige enheder vha. et index. MessageViewList indeholder derfor en liste, indexMap med referencer til alle de pakker, der kan vises i øjeblikket. Denne liste ændres, når et filter tilføjes eller fjernes fra pakkerne. Dette håndteres i Instance_FilteredPacketsChanged, der kaldes, når filtrene ændres.

Når der sker en anmodning om at vise en specifik enhed (pakke), oprettes et ListViewItem derfor ud fra pakken, der har det anmodede index i indexMap. Sådanne anmodninger kan komme ofte og også uden at der reelt er sket de store ændringer i listen. Derfor kan det vundne ved denne metode hurtigt tabes igen pga. overheaded ved konstant at skulle oprette nye ListViewItems. For at løse dette benyttes en cache af items, så hvis den samme enhed anmodes flere gange efter hinanden, vil enheden blot blive hentet fra cachen uden de store omkostninger. Dette håndteres i MessageViewList_CacheVirtualItems, der kaldes inden et stort antal enhedsanmodninger. Metoden kan sørge for at de rigtige enheder er i cachen for at gøre anmodningerne hurtigere.

Når pakker kommer ind i systemet, er det en fordel, at vinduet med pakker scroller i takt med, at pakkerne kommer ind på listen. Men dette vil give mange anmodninger efter enheder – og enheder, der ikke er i cachen. Der er derfor brug for, at når der kommer mange pakker indenfor kort tid, ændres listen (tilføjer pakker og scroller ned) stadig kun med et længere tidsinterval end det, hvormed pakkerne ankommer. updateSize metoden opdaterer listens størrelse og sørger for, at det nederste element er synligt vha. ensureVisible metoden i ListView. Men denne metode bliver kun kaldt med et vist interval, idet kaldet til metoden sker gennem et DelayUpdater objekt. DelayUpdater indeholder en tråd, der startes, når der kommer en anmodning på at opdatere. Først når tråden er klar (efter et interval) til en ny anmodning, bliver en sådan udført. Dermed vil et stort antal ankommende pakker kun ses i listen med et vist interval, så systemet kan følge med. Pakkerne kommer så at sige i klumper, hvilket også giver mening for brugeren, der alligevel ikke kan overskue et stort antal pakker på én gang.

4.4.3 Brug af NetBee

Umiddelbart indeholder NetBee funktioner til de operationer, der ønskes udført. Både mht. at få oplysninger om de definerede protokoller og mht. til at parse en pakke. Men det viser sig også i praksis, at NetBee biblioteket stadig er i en udviklingsfase. I projektet er der opstået fejl undervejs pga. fejl i NetBee eller en NetPDL beskrivelse er specificeret efter specifikationen, men alligevel giver NetBee forkerte oplysninger om de definerede protokoller. Og i disse situationer eller situationer, hvor man selv laver fejl, er NetBee dårlig til at give beskrivende fejlmeddelelser. Tilsammen gør disse aspekter NetBee svær at bruge.

Der har heldigvis været mulighed for at lave „work-arounds“ på de forhindringer, der er opstået undervejs i projektet, så det alligevel har været muligt at få en korrekt protokolanalysator. I appendiks D er de fundne forhindringer og evt. løsninger (eller „work-arounds“) beskrevet.

4.5 Integration af Ruby

I afsnit 3.5 blev en løsning til designet af integrationen af Ruby beskrevet. I dette afsnit beskrives implementeringen af dette design.

4.5.1 Ruby hjælpebibliotek

Hjælpebiblioteket består af to Ruby filer: Den ene er uafhængig af protokolanalysatoren og den anden er afhængig af protokolanalysatoren, dvs. den indeholder feltafhængige funktioner, der tager feltets datatype som input og tilføjer disse data til feltets protokol som rå bytes. Den første fil implementerer den definerede protokol for dataudveksling mellem frameworket og Ruby scripts, se afsnit 3.5.2. Når filen eksekveres, dvs. selvstændigt eller fordi den inkluderes i et andet Ruby script, forsøger den at oprette en TCP forbindelse til frameworket (via localhost). Filen definerer desuden funktioner til de 4 ønskede operationer: Start sniffing, stop sniffing, modtag pakke og send pakke. Desuden defineres to klasser: `SendPacket` og `ReceivedPacket`, der er datastrukturer, der kan bruges hhv. til at tilføje protokoldata før afsendelse af en pakke og til at trække data ud af en modtaget pakke. `SendPacket` indeholder en funktion, `add`, der tilføjer protokoldata til pakken. Når pakken sendes, sendes protokoldataene i den rækkefølge, de blev tilføjet pakken. `ReceivedPacket` overskriver index operatoren („`[]`“) så data i pakken kan tilgås som beskrevet i 3.5.1 og i eksempel 3.2. Constructoren i klassen sørger for at oprette datastrukturen ud fra givne bytes.

Filen indeholder desuden funktioner, der serialiserer NetPDL's datatyper til bytes, som kan benyttes, når felternes data skal serialiseres.

Den anden fil i hjælpebiblioteket oprettes af frameworket selv vha. modulet `RubyScriptCreator`, der er i stand til at generere et Ruby script i forhold til de oplysninger, der foreligger i protokoldatabasen. For hver defineret protokol defineres en klasse. Denne klasse indeholder funktioner, der hver især er i stand til at tilføje et specifikt felts data til protokollen. Når alle ønskede felter er tilføjet protokollen, kan denne tilføjes en pakke, `SendPacket`, der kan sendes via frameworket. Funktionerne, der tilføjer felters data til protokollen, benytter funktionerne fra den første fil. Den benyttede funktion vælges ud fra feltets datatype og længde. Disse funktioner kan så benyttes af brugeren til på en simpel måde (én linje) at oprette en pakke, der skal sendes.

Filen oprettes, når frameworket startes vha. Ruby script creator modulet, hvis den ikke allerede eksisterer i forvejen. Filen lægges i biblioteket „ruby“ under det bibliotek, hvorfra frameworket eksekveres.

4.5.2 Eksekvering af scripts

Ruby script executor modulet benytter `Ruby.exe` til at eksekvere scripts, der er indtastet i Ruby editor modulet. Scriptet eksekveres ved at starte `Ruby.exe` og dernæst sende hele scriptet til standard input for processen. Data på standard output og error output opfanges og der genereres en event i `FrameworkController`, så interesserede moduler kan få outputtet (fx de grafiske outputboks moduler). Når processen er afsluttet og output og error datastrømmene er tømt, er scriptet termineret og der sendes information om dette til `FrameworkController`. Når `Ruby.exe` startes, specificeres det, at hjælpebiblioteket skal loades som det første, så brugerens script kan gøre brug af biblioteket. Dette kan udføres med switchen `-r` til `Ruby.exe`.

Ruby editor modulet extender `RichTextBox`, der giver mulighed for at formatere den indeholdte tekst. Dette benyttes til at lave „syntax highlighting“ på teksten. Når brugeren indtaster kode, tjekkes syntaksen af den linje, som ændres. Hvis der indsættes tekst fra udklipsholderen, bliver al denne tekst ligeledes tjekket

(asynkront, så store mængder data ikke får programmet til at afvise brugerinput).

Syntaks tjekket foretages vha. Ruby syntaks reader moduler (RubySyntax-Reader). Dette modul kategoriserer ordene i en given tekst, så fx keywords identificeres. Disse oplysninger kan Ruby editoren bruge til at give de enkelte ord den rigtige farve, font osv.

4.5.3 Kommunikation mellem framework og Ruby scripts

Afsnit 4.5.1 beskrev, hvordan Ruby skabte forbindelse til frameworket. I dette afsnit beskrives frameworkets side af kommunikationen. Det er RubyInterface modulet, der tager sig af kommunikationen. Når et script startes, lytter modulet efter Ruby scriptets forbindelse vha. TcpListener klassen. Når denne forbindelse er oprettet, venter modulet på forespørgsler fra Ruby scriptet (der følger den definerede protokol) gennem den TcpClient og NetworkStream, der var resultatet af den succesfuldt oprettede forbindelse. Når en forespørgsel modtages, parses denne i henhold til protokollen. Den ønskede operation udføres gennem FrameworkController, og skal der lyttes efter sniffede pakker, fordi Ruby scriptet vil modtage en besked, oprettes en PacketListener, der modtager besked fra PacketDatabase, når en relevant pakke modtages. Når forespørgslen er parset sendes et svar (evt. med fejlkode) tilbage til Ruby scriptet gennem den stadigt åbne forbindelse.

Det bemærkes, at netværksoperationerne udføres i en separat tråd, så brugerinterfacet stadig svarer på brugerinput, mens netværkskommunikationen finder sted.

4.6 Brug af frameworket

I de foregående afsnit er det blevet beskrevet, hvordan frameworket er implementeret med de enkelte moduler. I dette afsnit ses der på, hvordan en protokolanalysator rent praktisk kan benytte frameworket, og hvilket arbejde, det kræver. Der ses på implementeringen af interfacet mellem frameworket og protokolanalysatoren, dvs. implementeringen af designet beskrevet i afsnit 3.6.

Interfacet er implementeret i PaInterface assembly. Her er defineret en abstrakt klasse, Kangaroo, som en protokolanalysator skal extende. Ved implementering af de abstrakte egenskaber og metoder, får frameworket adgang til de nødvendige dele i protokolanalysatoren. Disse egenskaber og metoder inkluderer de følgende:

SendInterfaces En liste med objekter af typen ISendInterface, så frameworket har adgang til alle definerede interaktionsenheder.

ReceiveInterfaces En liste med objekter af typen IReceiveInterface. Denne giver frameworket adgang til alle definerede sniffingenheder.

NetPDLfile En String, der angiver position og navnet på NetPDL beskrivelses filen.

IReceiveInterface og ISendInterface er interfaces, som indeholder de operationer, der blev angivet i afsnit 3.6. Instanser af klasser, der implementerer disse interfaces, stilles til rådighed for frameworket gennem ISendInterfaces og IReceiveInterfaces. Operationen for at sætte specifikke indstillinger for en netværksenhed er lidt speciel. Det varetages af metoden, HasSettings, der har to

formål: Dels at fortælle, om interfacet overhovedet har specifikke indstillinger, der kan sættes, og dels hvilken metode, der skal kaldes for at sætte disse indstillinger. `HasSettings` returnerer derfor en `bool`, er angiver, om interfacet kan sætte indstillinger og tager en `ref` parameter af delegate typen `SettingHandler`. Dette er den metode i protokolanalysatoren, der skal kaldes for at sætte indstillingerne.

Men hvordan initialiserer frameworket protokolanalysator klassen? Det gør den under opstart, hvor den kigger i assemblies i programmets bibliotek. For hver assembly finder den alle de indeholdte klasser, og hvis der er en klasse, der extender `Kangaroo`, kan den bruges som protokolanalysator. Dette gøres vha. reflektion med .NET's `System.Reflection` namespace.

`Kangaroo` klassen indeholder desuden de services, som frameworket stiller til rådighed for protokolanalysatoren. Disse er tilgængelige for protokolanalysatoren, da denne extender `Kangaroo` klassen, men de er ikke tilgængelige for plug-ins, da disse ikke har adgang til `Kangaroo` klassen (protected constructor). Dette er dermed som designet i afsnit 3.6. Disse services stilles til rådighed gennem følgende metoder:

AddSummaryHeader Tilføjer en kolonne til pakkevisningen. Værdien af kolonnen for en pakke afgøres vha. nedenstående metode.

PutSummaryDefinition Definerer en sammenhæng mellem en kolonne og definerede protokoller. Metoden tager et protokolnavn som parameter og et `Dictionary`, hvor kolonnenavne associeres med feltnavne fra den pågældende protokol. Hvis der modtages en pakke, hvor den øverste[7] protokol er den angivne, vil kolonnerne blive udfyldt med værdierne af de felter, som protokollerne er blevet associeret med.

OnNewFilter Definerer et nyt filter i frameworket.

OnNewPacketListener Sætter en `PacketListener` i pakkedatabasen, som derved underrettes, når en given type pakke (der passer på et givent filter) modtages.

OnNewPacket Underretter om en ny pakke, der er sniffet på netværket. Denne metode benyttes også, hvis en „pseudo“ pakke, dvs. en pakke, der ikke er fra netværket, men i stedet er sammensat af pakker fra netværket, opdages. Denne metode sætte hele NetPDL pakke parsingen i gang i frameworket.

`On<X>` metoderne realiserer deres funktion ved, at `<X>` eventen underrettes, hvilket navnet også henviser til, idet `C#` har tradition for at metoder, der underretter events får navnet `On<X>`, hvor `<X>` er eventens navn.

4.7 Plug-ins

I afsnit 3.7 blev det beskrevet, hvordan et interface til plug-ins skabes. Afsnittet behandlede også, hvordan et sådant interface kunne realiseres i `C#`.

I udviklingen af de beskrevne moduler er de access-modifiers, hhv. klasser, properties, metoder og events har fået, blevet overvejet nøje i overensstemmelse

[7] Med „øverste“ refereres til protokolstakken, dvs. den protokol, der er identificeret længst inde i byte strømmen. HTTP protokollen er fx den øverste protokol i en Ethernet-IP-TCP-HTTP pakke.

med beskrivelsen i afsnit 3.7. `public` og `protected` er kun blevet brugt, hvis det er blevet vurderet, at servicen er relevant for plug-ins.

I forvejen er frameworket delt op i 5 assemblies. Disses interaktion foregår vha. services, der også er tilgængelige for plug-ins, idet de er erklæret `protected` eller `public`. Derfor kan services med disse access-modifiers inddeles i følgende grupper:

- Services, der benyttes af frameworket alene.
- Services, der er udstillet til plug-ins og derfor (kan) benyttes af plug-ins.
- Services, der opfylder begge ovenstående behov.

I sidste ende er disse services dog imidlertid kun én gruppe, idet hver assembly kun udstiller de services, der er sikre for andre at bruge – uanset om det er plug-ins eller andre dele af frameworket; det er en del af det modulopbyggede design. Ovenstående gruppering er derfor mere et billede af, hvilke services, der er relevante for hvem.

4.7.1 Integration af plug-ins

Idet frameworket i sagens natur ikke kender plug-ins på forhånd, må de hentes dynamisk ind i programmet. Et plug-in defineres i sin egen assembly og lægges derefter i samme bibliotek som frameworket, som frameworket kan hente det fra. Dette kan gøres vha. reflektion og C#'s `System.Reflection` namespace. Frameworket kigger som sagt efter plug-ins i et specificeret bibliotek, men for at frameworket kan vide, hvilke klasser i de fundne plug-ins, der skal opfattes som plug-ins, defineres en attribute, `PlugInAttribute`, som skal sættes på de klasser, der er plug-ins. Denne klasse initialiseres i frameworkets initialiseringsproces vha. en standard constructor, dvs. en constructor uden parametre. Findes en sådan constructor ikke, ignoreres det pågældende plug-in af frameworket, dvs. klassens brug af `PlugInAtribute` bliver uden betydning.

For at summere op, så er alt der kræves for at lave et plug-in, at man laver en klasse med attribut'en, `PlugInAttribute`, implementerer en standard constructor og lægger de konstruerede assemblies i frameworkets bibliotek.

4.8 Konklusion på implementering og teknisk test

Kapitlet har givet en kortfattet gennemgang af implementeringen af prototypen af frameworket. Kapitlet indledtes med en beskrivelse af den benyttede implementeringsmetode, idet Test-Driven Development er et oplagt valg og derfor er brugt i de dele af implementeringsprojektet, hvor det har været muligt. Grafiske dele og multitrådede dele er ikke egnet til TDD.

Den generelle opbygning af frameworket er blevet beskrevet, dvs. hvordan et fleksibelt modulopbygget design er opnået. Specielt kommunikationen mellem modulerne skal implementeres med omtanke for at undgå et uoverskueligt net af referencer på kryds og tværs af moduler. En sammenbygning af Mediator, Singleton og Observer design patterns er benyttet til dette formål.

Den implementerede protokoldatabase, og specielt hvordan filtrering på denne er realiseret, er gennemgået. Filtre er opbygget i en træstruktur med filtererklæringer som blade i træet. En pakke eller en samling af pakker kan evalueres ved at gennemløbe træet.

Også aspektet omkring håndtering af pakkerne internt i frameworket er behandlet. Den interne pakkedatabase er beskrevet. Håndteringen af store mængder pakke er implementeret vha. dedikerede datastrukturer, der er lette at filtrere på, og en virtuel liste af pakker er benyttet til at minimere arbejdet med at vise pakkerne grafisk.

Et interface er skabt til Ruby, så frameworket kan eksekvere Ruby scripts. Frameworket stiller et Ruby bibliotek til rådighed, så arbejdet med at skrive scripts lettes for brugeren. En TCP kommunikation er implementeret, så Ruby scripts kan kommunikere med C# frameworket.

Frameworket benyttes af en protokolanalysator ved at extende en klasse i frameworket. Dermed tvinges implementatoren af protokolanalysatoren til at implementere de nødvendige dele og samtidig udstilles de dedikerede services fra frameworket til protokolanalysatoren, der kan benytte dem eller lade være.

Frameworket har indbygget mulighed for udvidelse vha. plug-ins, idet en assembly kan lægges i samme bibliotek som frameworket, der derefter inkluderer dets funktionalitet automatisk.

Den samlede implementering af en prototype af frameworket er fuldendt. Tabel 4.3 giver overblik over kodens omfang. Prototypen giver mulighed for implementering af protokolanalysatorer. Dette er emnet for de to næste kapitler, hvor to protokolanalysatorer implementeres. Disse kan danne baggrund for en brugertest af frameworket, der er vigtig for, hvad der skal være af ændringer i prototypen.

Sprog	**Assembly**	**Linier kode**
Framework		
C#	Common	1238
C#	Domain	3330
C#	Kangaroo	477
C#	Presentation	8114
C#	Utils	1934
C++/CLI	NetPDLwrapper	5043
Protokol analysatorer (ekskl. NetPDL beskrivelser)		
C#	I2CKangaroo	880
C#	CANgaroo	1896
C# og C++/CLI	Kangareal	2024
Plug-ins		
C#	PlugIns	351
Totaler		
C#	I alt	18410
C++/CLI	I alt	6877
C# og C++/CLI	Total	25287

Tabel 4.3: Omfang af implementeringen.

Kapitel 5

Case: I^2C protokolanalysator

Kapitlet beskriver en case, hvor frameworket benyttes i en konkret situation. Der skal laves en protokolanalysator til I^2C netværket. Idet situationen på FOSS under projektet har ændret sig og en I^2C ikke længere er relevant, er denne case medtaget, fordi det dels var projektets udgangspunkt dels kan vise fleksibiliteten i frameworket. Men af denne grund implementeres en I^2C protokolanalysator med et minimum af funktionalitet. Bl.a. udelades interaktionsdelen og i stedet koncentreres der om sniffing delen. Kapitel 6 behandler en mere kompleks protokolanalysator. Af denne grund udføres brugertests ikke her, men i stedet i kapitel 6.

I kapitlet beskrives designet af denne protokolanalysator, herunder hvordan der skabes kontakt til netværket samt hvordan trafikken kan beskrives, så frameworket kan benyttes. I denne beskrivelse defineres trafikken for alle lag af protokolstakken. Dernæst beskrives, hvordan dette design konkret er implementeret.

Indholdsfortegnelse

5.1 Realisering af I^2C protokolanalysator **71**

5.1 Realisering af I^2C protokolanalysator

NetPDL beskrivelsen af protokolstakken findes i appendiks C.2. Beskrivelsen af I^2C protokollen blev forklaret i afsnit 2.4.2 og beskrivelsen af de resterende protokoller (Octet, Hextet og T123) er trivielle (T123 benyttes i øvrigt igen i næste kapitel), så yderligere beskrivelse af NetPDL delen af I^2C protokolanalysatoren udelades her.

I^2C protokolanalysatoren definerer kolonner i frameworket, der er passende for denne protokol. Fra I^2C protokollen hentes adresser på afsender og modtager, og der er kolonner til disse oplysninger. Er beskeden en T123 besked, tages adressen herfra. Desuden er der kolonner til T1, T2 og T3 felterne.

Protokolanalysatoren skal også tage sig af beskeder, der sendes over flere pakker. Hextet definerer en måde at gøre dette på, så relevante Hextet pakker skal modtages, sættes sammen og sendes tilbage til frameworket, som beskrevet i afsnit 4.6. Protokolanalysatoren opretter filtre, så Hextet beskeder med `0x1E` værdien i command feltet sendes til protokolanalysatoren. Disse beskeder er nemlig lange beskeder, idet de fylder flere Hextet pakker. Klassen `MultiPacket` benyttes til at håndtere disse pakker. Når en pakke af denne type modtages i protokolanalysatoren, opbygges den samlede pakke vha. `MultiPacket` klassen. „Packet id" feltet angiver, hvornår en pakke er „færdigsamlet", idet den tæller ned til 0, dvs. feltet angiver, hvor mange pakker der er tilbage. Når `MultiPacket` registrerer, at en pakke er færdig, sendes den til frameworket og det angives vha. `jumpprotocol` variablen, at den skal parses som en T123 besked.

Som beskrevet i afsnit 1.2.3 har FOSS en USB-enhed, en „Beagle" til rådighed, der kan sniffe data på et I^{2}C netværk. Til denne USB-enhed eksisterer en API, der kan benyttes til at sende kommandoer til enheden. Denne API er en unmanaged dll. Klassen `Beagle` tilføjer en adapter (se boks 4, s. 53) til dette bibliotek af `C` funktioner, så I^{2}C protokolanalysatoren kan kommunikere med I^{2}C netværket. En Beagle kan kun sniffe data på netværket, men kan ikke interagere med det. Derfor implementerer `Beagle` klassen kun `IReceiveInterface`. `c_beagle_enable` og `c_beagle_disable` benyttes til hhv. at forbinde og afbryde forbindelsen til I^{2}C netværket. `c_beagle_i2c_read` benyttes i en løkke i en `BackgroundWorker` til at læse pakker på I^{2}C netværket. Når en pakke modtages, sendes den til frameworket, så den kan parses.

Dermed er en minimal implementering af en I^{2}C protokolanalysator produceret. Hvis en mere brugbar version skulle realiseres, skulle der først og fremmest tilføjes funktionalitet til interaktion med netværket. Et andet interface skulle tilkobles (en Aardwark), som skaber denne mulighed. Næste kapitel betragter en protokolanalysator, hvor interaktionsdelen er realiseret.

Kapitel 6

Case: CAN protokolanalysator

Kapitlet beskriver en CAN protokolanalysator til CAN bussen. Sammen med den første case, I^2C protokolanalysatoren, viser den fleksibiliteten i frameworket.

Indholdsfortegnelse

6.1 Motivation for en CAN protokolanalysator **73**
6.2 Design af CAN protokolanalysator **74**
6.3 Implementering af CAN protokolanalysator **78**
6.4 Brugertests **80**

6.1 Motivation for en CAN protokolanalysator

I den periode, som projektet har kørt, er der sket ændringer hos FOSS. I kapitel 1 blev ESWP arkitekturen beskrevet, se figur 1.1 s. 3. Den kommunikationsstak, som bruges i modulerne i FOSS' apparater blev beskrevet, idet apparaterne benyttede I^2C bussen. Dette er imidlertid blevet ændret, idet I^2C bussen viste svaghedstegn, som kun kunne løses ved at skifte bussen ud. Valget faldt på CAN bussen og ovenpå denne er en bærer af T123 beskeder blevet konstrueret. Men derudover har dette ikke ændret opbygningen af ESWP'en. Og fra applikationernes synspunkt er ændringen helt gennemsigtig, da beskeder stadig sendes i T123 format.

Ændringen har betydet, at FOSS' ønske om et værktøj til diagnosticering af I^2C trafik er afløst af et ønske om et værktøj til diagnosticering af CAN trafik. Men da projektet startede ud med visionen om en I^2C protokolanalysator er denne vision bibeholdt. Dog kan ændringen til CAN bussen give mulighed for at vise styrken i, at frameworket er uafhængigt af netværk, jf. FOSS' ønsker i afsnit 1.2.4 – specielt punkt 4 s. 4. Dette kapitel fungerer derfor som en ekstra case, der beskriver, hvordan frameworket kan benyttes af en CAN protokolanalysator.

En del af projektet har været at indsamle brugerkrav for at lave en kravspecifikation. Brugertests af en prototype af værktøjet kan bruges til at undersøge, om brugerkravene er opfyldt. Brugertests af frameworket kræver, at der er en protokolanalysator til stede, da frameworket jo ikke fungerer i sig selv. Fra starten har det selvfølgelig været mest hensigtsmæssigt at foretage brugertests på I^2C protokolanalysatoren. Men nu er det mest praktisk at foretage dem på CAN protokolanalysatoren, da denne bruges/skal bruges i dagligdagen på FOSS. En beskrivelse af resultaterne af brugertests på frameworket præsenteres derfor i dette kapitel.

6.2 Design af CAN protokolanalysator

For at opbygge en protokolanalysator, skal der stilles de services til rådighed, som blev identificeret i afsnit 3.6. Som interface til netværket benyttes en USB-enhed, „Kvaser Leaf Light“[1], hvortil der medfølger en API, der kan bruges, når der skal sniffes på/interageres med netværket.

Formatet af CAN protokollen er forholdsvist simpelt. Der eksisterer 4 typer af CAN pakker: Datapakke, remotepakke, errorpakke og overloadpakke. Disse typer har forskellige formater. Det er hovedsageligt datapakker, der benyttes, da disse bærer de egentlige data. En datapakke kan være i to forskellige formater: Standard og udvidet. Datapakker består af en identifier, der udover at blive brugt til at angive prioriteten på netværket, kan bruges af brugeren af CAN netværket. Den er 11 (standard format) eller 29 (udvidet format) bit, der altså kan fortolkes af brugeren som ønsket. Det eneste krav er, at to pakker med samme identifier ikke må sendes samtidig, da dette bringer CAN netværket i en fejltilstand. Hos FOSS bruges denne identifier fx til bl.a. at angive modtager og afsender adresser, hvilket dermed opfylder kravet om, at der ikke sendes to ens identifiers samtidig. En CAN datapakke kan indeholde op til 8 bytes data.

6.2.1 Format af protokollerne

I USB-enheden abstraheres der en smule fra CAN protokolformatet, når der sniffes på netværket. Dvs. de bit, man får fra CAN API'en, er udformet som beskrevet ovenfor. API'en tager sig fx af at undersøge, om en datapakke er i standard eller udvidet format. Det CRC tjek, som er i alle CAN pakker, bliver ligeledes behandlet i API'en, så pakker med forkert CRC slet ikke „opdages“. Dette betyder, at NetPDL beskrivelsen af protokollen skal følge det format, der gives af API'en, da protokolanalysatoren og dermed frameworket aldrig ser det oprindelige format. Når der sniffes på netværket fås følgende fra API'en:

Identifier (32 bit) De 11 (ved standard format) eller 29 (ved udvidet format) mindst signifikante bit indeholder CAN pakkens identifier.

Data (0-8 bytes) Indeholder pakkens data.

Længde (32 bit) Angiver, hvor meget data, der er. Værdien er således 0-8, dvs. reelt bruges kun de 4 mindst signifikante bit.

[1] Produktet kan ses på `http://www.kvaser.com/prod/hardware/leaf_light.htm` – tilgængelig 23/5-2007.

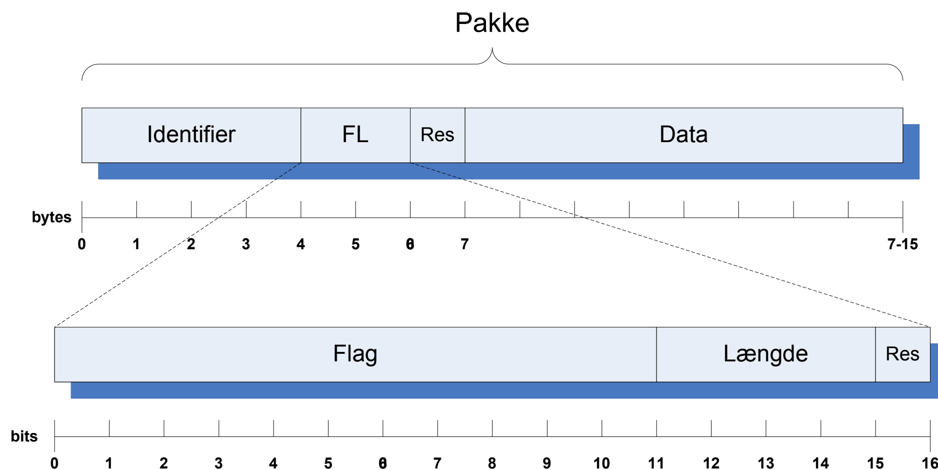
Flag (32 bit) De 11 mindst signifikante bit angiver 10 flag, der beskriver pakken, se tabel 6.1. Bemærk at bitten med position 9 ikke benyttes.

Tid (32 bit) Angiver det tidspunkt (i μs siden sniffingen startede), hvor pakken optrådte på netværket.

Flag	Position	Betydning
canMSG_RTR	1	Remotepakke
canMSG_STD	2	Datapakke i standard format
canMSG_EXT	3	Datapakke i udvidet format
canMSG_WAKEUP	4	Wakeup besked
canMSG_MSG_NERR	5	Errorpakke (kritisk)
canMSG_ERROR_FRAME	6	Errorpakke
canMSG_TXACK	7	Har fået acknowledge
canMSG_TXRQ	8	Sendt til CAN controlleren
canMSGERR_HW_OVERRUN	10	Hardware buffer fyldt
canMSGERR_SW_OVERRUN	11	Software buffer fyldt

Tabel 6.1: Flag i en CAN pakke. Positionen er angivet fra den mindst signifikante bit.

Disse data kan serialiseres til en række bytes med det udseende, der ses på figur 6.1. „Res“ angiver, at de pågældende bit eller bytes ikke benyttes, men er *res*erveret til fremtidig brug. Det viste format er det format, NetPDL beskrivelsen skal specificere.



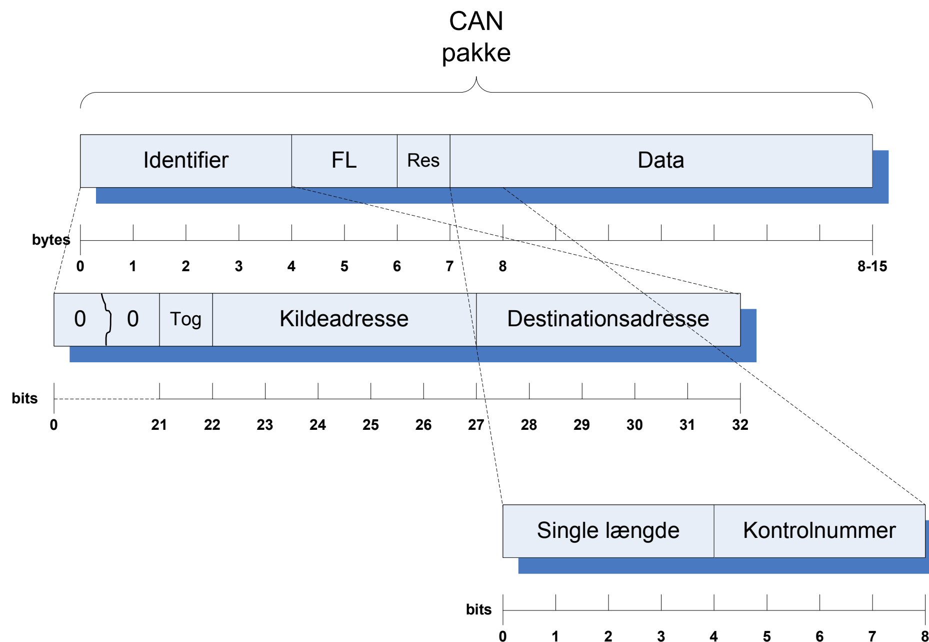
Figur 6.1: Formatet af CAN protokollen, som NetPDL skal specificere det.

6.2.2 FOSS specifikke protokoller

Som nævnt har FOSS konstrueret en protokol (ESWP CAN) med det formål at kunne sende beskeder med en længde, der er større end de 8 bytes, der normalt er muligt på CAN bussen. Det ligger uden for dette projekt at give en detaljeret

beskrivelse af protokollen. For dette projekt er det kun nødvendigt at kende protokol strukturen og strukturen i sammensatte pakker. I nedenstående fokuseres der derfor på denne struktur.

Protokollen specificerer, hvordan en stor pakke deles i mindre CAN pakker. Figur 6.2 viser, hvordan en CAN pakke fortolkes iflg. denne protokol.



Figur 6.2: Format af en CAN pakke med FOSS' fortolkning.

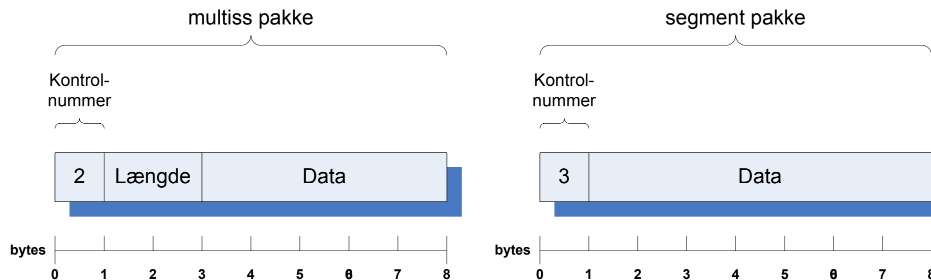
Kontrolnummeret angiver pakkens type. Tabel 6.2 viser de forskellige typer.

Kontrolnummer	Type	Betydning
1	single	Single segment pakke
2	multiss	Multi segment pakke
3	segment	Segment pakke
8	linkupr	Linkup request
9	linkupa	Linkup acknowledge
10	ready	Ready signal
11	flowstart	Signal om overbelastning
12	flowstop	Signal om overbelastning slut

Tabel 6.2: Betydningen af kontrolnumre i ESWP CAN. De numre, der ikke er defineret er reserverede til fremtidig brug.

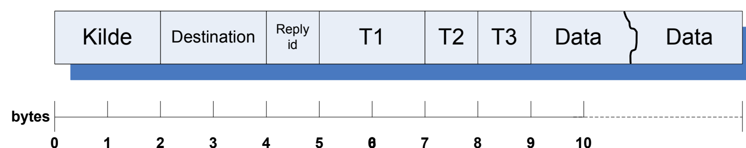
Det er kun pakker med kontrolnumre 1-3, der har yderligere data tilknyttet. De resterende (8-12) bruges til at sætte kommunikationen op mellem moduler (8-9), til at holde forbindelsen åben (10) og til at styre mængden af data for ikke at blive overbelastet (11-12). Kontrolnummer 1 benyttes, hvis der skal sendes data, der ikke er mere end 7 bytes og derfor kan være i én CAN pakke. Antallet af benyttede bytes angives i de 4 mest signifikante bit i kontrolbyten, se figur 6.2.

Kontrolnumre 2-3 benyttes, hvis der skal sendes data med en længde på mere en 7 bytes[2] - disse beskeder benævnes herefter som „lange pakker". Kontrolnummer 2 benyttes som den første CAN pakke i en lang pakke og kontrolnummer 3 benyttes i de resterende pakker. Figur 6.3 viser strukturen i sådanne pakker.



Figur 6.3: Formatet af lange pakker.

Ovenstående har beskrevet den protokol, som FOSS har konstrueret og som CAN bussen skal bære. Som tidligere nævnt benytter FOSS protokollen til at sende T123 beskeder. Disse beskeder kan ses som en protokol, som er payloaden i lange pakker. Figur 6.4 viser formatet af sådanne beskeder.



Figur 6.4: Formatet af T123 beskeder.

Ovenstående har beskrevet formatet af de protokoller, der skal specificeres vha. NetPDL. Figurerne lader sig forholdsvist nemt oversætte til en NetPDL beskrivelse, som det senere vil fremgå.

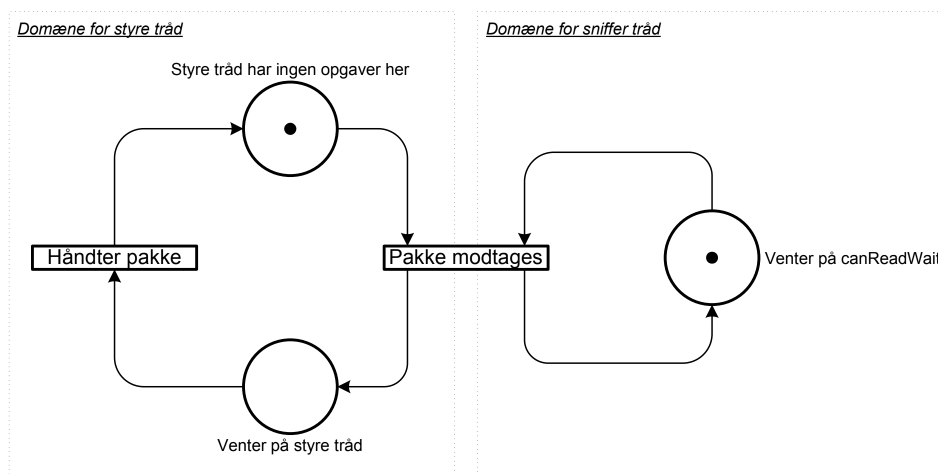
6.2.3 Brug af CAN API

Den medfølgende API til USB-enheden forefindes bl.a. i en managed .NET assembly, hvilket gør den meget tilgængelig i dette projekt. De fleste funktioner er ligetil at benytte. Funktionerne til at sniffe pakker fra netværket med er dog lidt specielle, hvorfor en procedure til sniffing forklares her.

Når pakker opdages på netværket bliver de gemt i en buffer i driveren til USB-enheden. Det er derefter protokolanalysatorens opgave at tømme bufferen ved at læse pakkerne en efter en. API'en stiller forskellige funktioner til rådighed til dette. `canReadWait` kan med fordel benyttes, da denne blokerer, indtil der er en pakke i bufferen, der kan læses. Dog kun indtil en fastsat timeout. CAN protokolanalysatoren kan benytte denne funktion fra en selvstændig tråd (sniffer tråd). Styre tråden (dvs. „Main" tråden, som er den tråd, der fra frameworket opretter protokolanalysatoren) kan samarbejde med sniffertråden om at give frameworket besked

[2] Da al kommunikation i ESWP benytter T123 beskeder, der er minimum 9 bytes, er dette den eneste metode, der benyttes til dataoverførsel på nuværende tidspunkt.

om pakkerne på netværket. Figur 6.5 viser et Petri-net [8] over synkroniseringen mellem de to tråde.



Figur 6.5: Håndtering af pakke sniffing.

Et andet aspekt, der skal tages højde for er, at man nemmest håndterer USB-enheden (enhederne) fra den samme tråd, da den identifikation på enhederne, man får fra API'en, kun kan bruges fra den tråd, som fik den. Dette kan håndteres fra en central tråd, der har en kø af API kommandoer, der ønskes udført på CAN driveren. Kommandoerne udføres én efter én og er der ikke kommandoer til stede, venter tråden blot på den næste kommando.

I næste afsnit ses der nærmere på den konkrete implementering af dette system.

6.3 Implementering af CAN protokolanalysator

I dette afsnit beskrives kort de ikke-trivielle dele af implementeringen af CAN protokolanalysatoren. Det drejer sig om realisering af designbeskrivelsen i forrige afsnit.

En NetPDL beskrivelse af protokollerne er vedlagt i appendiks C.3. Beskrivelsen er en direkte oversættelse fra figurerne i forrige afsnit, hvorfor den ikke beskrives yderligere her.

CAN protokolanalysatoren består af følgende klasser:

CanInterface Extender `Kangaroo` og er derfor en klasse, frameworket finder, når den leder efter en protokolanalysator. Klassen implementerer de abstrakte dele af `Kangaroo`, sætter håndteringen af lange pakker op og finder tilsluttede CAN USB-enheder. Desuden sætter den de ønskede kolonner op i pakkevisningen og associerer felterne i de definerede protokoller med de enkelte kolonner.

CanDirectedCommand Benyttes når der skal udføres en kommando på et CAN interface. Kommandoen og resultatet af kommandoen er pakket ind i instanser af denne klasse.

CanNetwork Abstrakt basisklasse for alle CAN interfaces. Indeholder funktionalitet til at forbinde og afbryde forbindelsen, ligesom der er metoder, som giver mulighed for indstiling af CAN interfacets bitrate.

Sender Extender `CanNetwork` og implementerer `ISendInterface` og giver dermed mulighed for at sende data til netværket.

Receiver Extender `CanNetwork` og implementerer `IReceiveInterface` og giver dermed mulighed for at sniffe data fra netværket. Bruger desuden en `BackgroundWorker` til at implementere en sniffer som beskrevet i afsnit 6.2.3 og på figur 6.5. Sniffertråden er den implementerede `BackgroundWorker` og styre tråden er den tråd, der opretter `Receiver` klassen, dvs. den tråd, frameworket bruger til at finde protokolanalysatorer med. Sniffer tråden sniffer dataene ved at benytte de sniffe kommandoer, `CanController` tilbyder.

CanPacket Indkapsler alle nødvendige data i en CAN pakke modtaget af snifferen. Dataene i en instans af denne klasse serialiseres i det specificerede format fra figurerne i afsnit 6.2.1 (der forstås af NetPDL) inden den sendes til frameworket. Indkapslingen bruges altså kun internt i protokolanalysatoren.

CanController Har tilknyttet en tråd (en `BackgroundWorker`), der udfører kommandoer på CAN USB-enhederne. Alle kommandoer udføres gennem denne klasse, hvorved det opnås, at kun én tråd tilgår CAN USB-enhederne som beskrevet i afsnit 6.2.3. De tilgængelige kommandoer er defineret vha. `CanCommands` enumeratoren. Klassen indeholder en metode til at hente resultatet af en kommando. Denne metode blokerer den kaldende tråd, indtil et resultat er tilgængeligt. På den måde kan kommandoer udføres både synkront (der ventes på resultat) og asynkront (resultatet ignoreres, eller det hentes senere).

CanChannel Indkapsler alle de data, driveren til USB-enhederne stiller til rådighed om de „channels“, der er til rådighed på CAN netværket. Klassen bruges til at repræsentere hver channel på CAN netværket i protokolanalysatoren.

Multipacket Opsamler data fra lange pakker og underretter, når den komplette lange pakke er samlet af de mindre. Klassen er i stand til at samle lange pakker mellem forskellige kommunikerende enheder i netværket. En lang pakke identificeres vha. afsender- og modtageradresser – se afsnit 6.3.1.

SrcDest Indkapsler data om kommunikerende enheder i netværket, idet adresserne gemmes. Overskriver `GetHashCode` og `Equals` metoderne, så `Multipacket` kan sammenligne objekter af denne klasse.

SettingForm Extender `Form` klassen. Viser de mulige indstillinger (bitrate) for og oplysninger om en benyttet USB-enhed (eller channel) for brugeren, så brugeren kan sætte indstillingerne i forhold til det benyttede netværk.

6.3.1 Implementering af håndtering af lange pakker

Som beskrevet ovenfor, kan `Multipacket` klassen benyttes til at håndtere en lang pakke. `CanInterface` opstiller to filtre i frameworket, og tilknytter en `PacketListener` til filtrene, så protokolanalysatoren underrettes, når der kommer multiss

hhv. segmentpakker, jf. tabel 6.2. Når der modtages en multiss-pakke, oprettes en ny `MultiPacket` og den kilde og destination, der kan findes i CAN identifier'en benyttes til at identificere pakken efterfølgende. Pakken gemmes derfor i et statisk `Dictionary`, hvor kilde/destination er nøgle og pakken selv er værdien. Fra multiss pakken findes længden af den samlede lange pakke, og de op til 5 bytes data, der er i multiss pakken, gemmes i en liste af bytes.

Når der modtages en segment pakke, findes pakken (`MultiPacket` i det omtalte `Dictionary` og dataene i segment pakken tilføjes listen af den samlede data. Når denne liste indeholder den forventede mængde bytes (fundet i længde feltet i multiss pakken) er pakken færdigbehandlet. `MultiPacket` sætter derfor en `bool`, `Processed`, til `true`, der angiver, at pakken er færdigbehandlet. Dette opdager `CanInterface`, der sender den samlede pakke til frameworket. `jumpprotocol` sættes til 1, hvilket betyder, at der er samlet en lang pakke. Dette benyttes i NetPDL, som det ses i appendiks C.3, idet en pakke med `jumpprotocol` sat til 1 parses som en T123 besked.

6.4 Brugertests

I det foregående er det blevet beskrevet, hvordan den første prototype af frameworket er implementeret, og hvordan denne prototype er blevet brugt til at konstruere en CAN protokolanalysator. Frameworket er blevet testet teknisk vha. unittests (i forbindelse med TDD) og manuelt, men det er ikke klargjort, om frameworket indeholder den funktionalitet, der blev fundet under kravsanalysen. Med en funktionel protokolanalysator er dette nu muligt. Som tidligere nævnt er det CAN protokolanalysatoren, der danner baggrund for brugertests, da det er dette værktøj der p.t. er relevant for FOSS.

En sådan brugertest har følgende formål:

1. At undersøge om kravspecifikationen er opfyldt.

2. At finde tekniske fejl, der ikke er fundet under den tekniske test.

3. At stille nye krav til produktet, som ikke blev fundet under den indledende kravsanalyse.

4. At finde dele i applikationen, der ikke er noget af ovenstående, men snarere et „funktionelt problem“, dvs. elementer, som har nogle uheldige bivirkninger.

Hvis de tekniske tests har været grundige nok, burde punkt to være minimalt. Erfaringen viser dog, at i praksis findes der yderligere tekniske fejl. Dette skyldes, at man i almindelig brug over længere tid vil udsætte applikationen for aspekter, der ikke er tænkt på eller ikke har været mulige at teste i den tekniske test. Sådanne aspekter kan fx være store datamængder, kørsel over meget lang periode, anderledes netværk, anderledes opsætning i platformen e.l.

Der gør sig noget lignende gældende for punkt 3. Hvis den indledende kravsanalyse har været grundig nok burde også dette punkt være minimalt, men i højere grad end ved punkt 2 er der her mulighed for at finde nye krav, når applikationen bliver benyttet i almindelig brug. Der opstår irritationsmomenter, eller man får brug for funktionalitet, der ikke kunne forudsiges. Dette skyldes, at en indledende kravsanalyse er teoretisk, og en oplevelse af en applikationen er vidt forskellig i teori og praksis. I øvrigt er det meningen med en iterativ udviklingsproces, at der

kan opstå nye krav. Man inkluderer kun den funktionalitet, man ved er krævet. Anden (mindre prioriteret) funktionalitet udsættes til en senere iteration, hvis et reelt behov viser sig.

Bemærk at ovenstående underbygger, at applikationen skulle fremstilles som en iterativ proces, og at designet skulle opbygges modulært og fleksibelt.

Der findes mange mulige metoder til at foretage brugertests. Disse metoder relaterer sig til metoder, der kan benyttes i forbindelse med en kravsanalyse (da man principielt udfører en ny kravsanalyse), jf. afsnit 2.1.2. Her benyttes to metoder, der hver især giver forskellige typer resultater (fordelt på ovenstående 4 grene) [11]:

- Ledsaget brug af applikation. En bruger benytter applikationen, mens hans brug overvåges. Denne metode afslører problemer med brugerinterfacet og generelt problemer med intuitionen forbundet med applikationen. Der er fx et problem, hvis brugeren leder længe efter en funktionalitet eller benytter en funktionalitet forkert.
- Feedback på brug gennem længere tid. En bruger benytter applikationen i sit daglige arbejde og inrapporterer krav, der ikke er opfyldt, fejl eller nye krav.

Den første metode er relevant, når brugeren ikke har benyttet applikationen før, mens den anden metode sker som en løbende proces fra applikationen er færdig.

I forbindelse med de brugertests, der er foretaget i dette projekt, er fundne tekniske fejl blevet rettet. Det følgende koncentrerer sig derfor om opfyldelsen af kravene, evt. nye krav og funktionelle problemer.

Appendiks B opsummerer resultaterne af de foretagede brugertests. I resultaterne er alle fundne aspekter medtaget. Der er fx forslag til nye krav, som blot én bruger er fremkommet med. Resultaterne skal derfor ses som *forslag* til en videreudvikling af prototypen. Inden en implementering af nye krav foretages, bør det undersøges, om kravet er repræsenteret mere generelt i brugergruppen.

I appendicet ses, at der ikke er mange krav til applikationen. Dette skyldes hovedsageligt, at applikationen p.t. ikke har været i brug over en længere periode. Da det er et værktøj, der kan udbygges med funktionalitet i det uendelige, kan man sagtens forestille sig, at der opstår nye krav løbende. Desuden ses af resultaterne, at der ikke er mange krav fra kravspecifikationen, som brugerne har manglet. De krav fra kravspecifikationen, der er identificeret, blev identificeret som „nice to have“. Men da det nu i praksis har vist sig, at der er et behov for funktionaliteten, er dette oplagte emner til ny version. Fx er eksportering til csv en oplagt ide til et plug-in. Dette gælder desuden også andre af de (nye og gamle) krav. Og kan de ikke det, er der måske mulighed for at inkludere det i protokolanalysatoren. Men det bør man kun gøre med protokolspecifikke krav.

Problemer med brugergrænsefladen er blandt de funktionelle problemer. Det er typisk problemer, der er identificeret vha. en ledsaget brug af applikationen, hvor det har vist sig, at en bruger har haft problemer med brugergrænsefladen. Ofte ligger løsningen ligefor i disse situationer, fx blev det identificeret, at det forvirrer, at der er brugt ens ikoner til „styringsknapperne“ til sniffing hhv. interaktion. Den oplagte løsning er her at skifte ikonerne ud!

Anderledes er det med funktionelle problemer, der har nogle mere tekniske bivirkninger. I brugertesten blev det identificeret, at der kan være problemer med kompabiliteten af Ruby scripts fra computer til computer, da Ruby scripts kan

afhænge af to ting: Definitionerne af filtre og den pågældende protokolanalysator. Flytter man fx et Ruby script, der benytter et filter til at modtage pakker, fra en computer til en anden, kræves det, at den anden computer har de benyttede filtre defineret. Dette aspekt giver unødvendig forvirring for brugeren, idet scriptet fejler, hvis et filter ikke er tilstede. I en fremtideig version af frameworket bør det overvejes, hvordan et Ruby script kan definere udefinerede filtre i frameworket, så de kan benyttes.

Den anden del af problemet, hvor et Ruby script er afhængig af protokolanalysatoren, må betragtes som et mindre problem, da det kan forudsættes, at et Ruby script tilknyttes én bestemt protokolanalysator. Det samme gør sig i øvrigt gældende for filterdatabasen, der gemmer definerede filtre i en ekstern datafil. Denne datafil er ligeledes tilknyttet én specifik protokolanalysator. Problemet er som nævnt ikke stort, da det ikke har været et krav, at frameworket skulle kunne understøtte flere protokolanalysatorer på én gang. Bliver dette et krav i fremtiden, skal dels Ruby scripts dels filterfilen indbygge oplysninger om, hvilken protokolanalysator de er tilknyttet.

Generelt viser resultaterne af brugertestene en god overensstemmelse mellem kravspecifikationen og brugerkrav. Men de viser også, at der er potentiale for udvidelser både internt i frameworket, i plug-ins og i protokolanalysatoren i den næste iteration af udviklingsprocessen, som allerede er startet med de fundne krav.

Kapitel 7

Eksempler på plug-ins

I dette afsnit beskrives eksempler på forskellige plug-ins til frameworket, der er lavet for at vise mulighederne i frameworket. Meningen med muligheden for plug-ins er som tidligere gennemgået at give mulighed for i fremtiden at tilføje funktionalitet til frameworket uden at dette skal rekompileres. Derfor er det principielt ikke en del af projektet at implementere plug-ins, men muligheden for at gøre det belyses bedst vha. eksempler.

I kapitlet gennemgås to eksempler på plug-ins, der hver især implementerer en „nice to have“ funktionalitet. Disse funktionaliter blev identificeret i afsnit 2.2.

Det første eksempel implementerer muligheden for at sætte farver på pakkerne i pakkevisningen for at lette læseligheden af oversigten. Det andet eksempel implementerer muligheden for at dumpe en markeret del af en pakkes data til en ekstern datafil.

I afsnittene lægges vægt på kommunikationen med frameworket og ikke de tekniske dele af plug-in'ene, da det er sammenhængen, der er interessant og ikke eksemplerne i sig selv.

Indholdsfortegnelse

7.1 Farver i pakkevisningen **83**
7.2 Dump af pakkedata **84**

7.1 Farver i pakkevisningen

I brugerundersøgelsen stod det klart, at alt, hvad der kan hjælpe på læseligheden af beskederne, er en fordel, jf. afsnit 2.2.1. En metode til at adskille pakkerne fra hinanden er ved at give dem forskellige farver. Dette plug-in giver mulighed for at give pakker, der opfylder et givent filter, en given farve. Plug-in'et udnytter de eksisterende filtre og tilknytter en farve til hvert filter. Når pakker skal vises i pakkevisningen, får plug-in'et besked om det og sætter baggrundsfarven på det

pakkeelement, der repræsenterer farven – men kun hvis pakken passer på et af de filtre, der har fået tilknyttet en farve.

Dette plug-in er implementeret i følgende klasser:

ColorPackets Benytter attributten `PlugIn`, hvorfor det er denne klasse, frameworket identificerer som plug-in. Klassen tilføjer en knap til sniffing-værktøjslinjen. Når der klikkes på denne åbnes et skærmbillede af typen `ColorFilter`. Klassen abonnerer desuden på events i frameworket: `NewViewPacket`, så nye pakker i pakkevisningen sendes til denne klasse og `Shutdown`, så de definerede farve/filter sammenhænge kan gemmes på disken.

ColorFilter Et skærmbillede (typen extender `Form`) der viser en liste over de definerede filtre. Denne liste vedligeholdes af en `FilterHandler` fra præsentationslaget. Desuden vises en dropdown liste af farver, som man kan vælge sammen med et filter. Når der vælges en sammenhæng, gemmes dette i et tilknyttet `ColorMap`. Filtre vises i en `ListBox` og farverne i en `ComboBox`.

ColorMap Er en datastruktur, der gemmer sammenhængen mellem filtre og farver i en `HashTable`[1]. Klassen tilbyder metoder til at indsætte og udtrække filter/farve sammenhænge. Klassen kan også ud fra de tilknyttede sammenhænge give en farve til en given pakke. Dette benyttes af `ColorPackets`, når der modtages en pakke fra frameworket. Desuden er der metoder til serialisering og deserialisering af den tilknyttede `Hashtable`.

ColorItem Repræsenterer et element i den `ComboBox`, der i `ColorFilter` viser de mulige farver.

Dette plug-in benytter services i frameworket til at udføre dets opgave. `RequestToolBar` i `PresentationRequestHandler` benyttes til at skaffe sniffing-værktøjslinjen, så den nye knap kan tilføjes. Når der klikkes på denne knap vises et `ColorFilter` vindue. Dette vindue får framework vinduet som „parent“, så framework vinduet ikke kan benyttes, før det nye vindue er lukket. Denne „parent“ sættes ud fra `MainForm` i `PresentationController`.

`Shutdown` fra `DomainController` benyttes til at få oplysning om, at frameworket er ved at lukke ned, så det tilknyttede `ColorMap` kan gemmes på disken. Når dette plug-in eksekveres af frameworket, bliver dette `ColorMap` i øvrigt initialiseret fra den gemte fil, hvis den eksisterer.

`NewViewPacket` fra `PresentationController` benyttes til at få oplysning om, når en ny pakke skal vises på skærmen. Metoden `Instance_NewViewPacket` kaldes og denne sætter baggrunden på den nye pakke i pakkevisningen i henhold til de tilknyttede filter/farve sammenhænge.

Det eneste, plug-in'et skal bekymre sig om i forhold til at blive hentet ind, er, at den skal sætte attributten `PlugIn` på `ColorPackets` klassen og den genererede assembly skal lægge i applikationsbiblioteket, så frameworket kan lokalisere det.

7.2 Dump af pakkedata

Dette plug-in skal gøre det muligt at lave et dump, dvs. en fil, af en markeret del af de bytes, en markeret pakke indeholder. Plug-in'et skal for brugeren fungere ved,

[1] Der vælges en `Hashtable` i stedet for den generiske `Dictionary`, da `Hashtable` kan serialiseres direkte.

at man i den nederste del af skærmen i sniffingdelen (datavisningen) markerer de bytes, man vil have gemt i en fil. Når de bytes er markeret, højreklikker man og vælger et menupunkt, „Dump data to file" fra den menu, der popper op. Derefter får man en „gem fil" dialog, hvor man kan bestemme den fil, dataene skal gemmes i.

Plug-in'et er noget mere simpelt end det første plug-in, da frameworket har services, der relativt direkte giver adgang til den ønskede funktionalitet. Det er derfor bygget op af blot én klasse, `Dumper`. Klassen benytter i initialiseringen `RequestAddContextMenuItem` i `PresentationRequestHandler` til at tilføje det ønskede menupunkt til den højrekliksmenu, der vises, når der højreklikkes i datavisningen. Når der klikkes på dette menupunkt, eksekveres `item_Click`.

`item_Click` benytter `RequestHexBoxSelection` defineret i `PresentationRequestHandler` til at få de bytes, der er markeret i datavisningen. Disse bytes gemmes derefter i en fil vha. en `BinaryWriter` (i `System.IO` namespace). Filen, der gemmes i, vælges vha. en `SaveFileDialog` (i `System.Windows.Forms` namespace).

Klassen `Dumper` er tilknyttet attributten `PlugIn` og den assembly, der genereres, lægges i applikationsbiblioteket, så frameworket kan lokalisere og hente det ind.

Kapitel 8

Konklusion

Rapporten har redegjort for det udførte projekt. I dette kapitlet samles op på de resultater og konklusioner, der kan drages af projektet. Der tages udgangspunkt i projektets vision (afsnit 1.2.5) og der redegøres for, hvordan visionen er blevet opfyldt.

Indholdsfortegnelse

8.1 Overordnede betragtninger **87**
8.2 Konklusioner om frameworket **88**
8.3 Konklusioner om protokolanalysatorer **88**
8.4 Bruger konklusioner **89**
8.5 Perspektivering **89**
8.6 Virksomhedskonklusion på studenterprojekt **90**
8.7 Endelig konklusion **90**
A.1 Interview med Mogens Velsing **91**
A.2 Uformelle interviews med ESWP udviklere **93**

8.1 Overordnede betragtninger

Projektet har vist, at det er muligt at skabe et fleksibelt framework, der kan benyttes på et vilkårligt computernetværk. I projektet er dette framework blevet designet og implementeret. Frameworket kan bruges til at udvikle en protokolanalysator, og implementeringen af denne protokolanalysator afgør, hvilket computernetværk, værktøjet kan benyttes på. Frameworket er designet, så det kan udvides med funktionalitet vha. plug-ins.

8.2 Konklusioner om frameworket

Den implementerede funktionalitet blev fundet ud fra en undersøgelse af brugerkrav. Her er brugerne de medarbejdere på FOSS, der dels udvikler applikationer til ESWP dels udviklerne af selve ESWP. Bruger undersøgelserne resulterede derfor i overordnede krav til det endelige værktøj, der skal bruges på FOSS' netværk i ESWP. Disse krav blev delt i krav til frameworket og krav til protokolanalysatoren. De fundne krav blev ligeledes prioriteret, og højtprioriteret funktionalitet blev implementeret direkte, mens lavere prioriteret funktionalitet blev overladt som mulige udbygninger – fx som plug-ins.

Frameworket er designet fleksibelt. Dette fleksible design er opnået ved et modulopbygget design og ved benyttelse af en række design patterns. Dette har skabt muligheden for at frameworket kan udvides vha. plug-ins. Begrænsninger i mulighederne for et plug-in ligger i de muligheder, interfacet til frameworket tilbyder.

Til definition af protokoller i et givent netværk viste NetPDL teknologien at udmærke sig. Denne teknologi gør det muligt på en intuitiv måde at specificere protokollerne i XML. Frameworket benytter NetPDL specifikationen fra protokolanalysatoren til at finde protokolmønstre i en rå datastrøm.

Frameworket håndterer store mængder af data, dvs. pakker, ved benyttelse af dedikerede datastrukturer i dels en pakkedatabase dels på skærmen i listen over pakker. Pakkedatabasens datastruktur gør effektiv filtrering mulig – køretiden afhænger ikke af antallet af pakker i databasen.

Frameworket letter udviklingen af en protokolanalysator, da udviklingen er begrænset til en definition af protokolstakken samt at skabe et interface til det specifikke netværk. Disse elementer registreres i frameworket af protokolanalysatoren. Protokolanalysatoren kan dog tilpasse frameworket den specifikke protokol på flere måder – men dette er valgfrit.

Frameworket er testet teknisk vha. unittests og manuelle tests til områder, hvor unittests er mindre velegnede.

8.3 Konklusioner om protokolanalysatorer

Som eksempel på brugen af frameworket er der designet og implementeret en I^2C specifik protokolanalysator, da det var en del af projektets vision at udvikle en sådan. Dette skyldtes, at projektet var motiveret af FOSS' behov for et sådant værktøj.

Under projektforløbet blev frameworkets fleksibilitet dog sat på en prøve, idet FOSS besluttede at skifte I^2C bussen ud med en CAN bus. Derved ændrede FOSS' behov sig til et ønske om et CAN værktøj. Derfor er der i projektet implementeret en I^2C protokolanalysator for at bibeholde projektets vision og en CAN protokolanalysator for at teste frameworkets fleksibilitet. Idet CAN protokolanalysatoren er FOSS' nye behov, er I^2C protokolanalysatoren implementeret med minimal funtionalitet. Det har vist to ting: En protokolanalysator kan implementeres uden meget arbejde og det er muligt at implementere vidt forskellige protokolanalysatorer.

I begge protokolanalysatorer er den FOSS specifikke protokolstak beskrevet i NetPDL. Interfacet til CAN-netværket er skabt vha. en adapter, der både kan sniffe og interagere. Til I^2C-netværket er en adapter brugt (Beagle), der kun kan sniffe – derfor er interaktionsdelen udeladt i I^2C protokolanalysatoren.

8.4 Bruger konklusioner

Frameworket er designet ud fra en kravspecifikation, der er resultatet er brugerundersøgelser. Efterfølgende er den fremstillede prototype brugertestet for at finde ud af, om kravspecifikationen er opfyldt. Desuden giver brugertestene ideer til fremtidige forbedringer.

De udførte brugertests viste, at kravspecifikationen er opfyldt i forhold til de højt prioriterede punkter, mens et par lavere prioriteret punkter har vist sig at være ønsket. Disse kan implementeres i en fremtidig version. Brugertestene har også vist enkelte uhensigtmæssigheder i brugerinterfacet – dette bør ligeledes forbedres i en fremtidig version.

Frameworket er designet med en iterativ udviklingsproces som metode. Derfor er designet også fleksibelt mht. ændringer fx i form af plug-ins. De gennemførte brugertests kan derfor fungere som udgangspunkt for den næste iteration i udviklingsprocessen.

8.5 Perspektivering

Der er brugerkrav, der ikke er implementeret i værktøjet, da de ikke har været prioriteret højt nok til at dette syntes nødvendigt. Desuden er der som det ses ovenfor fundet funktionalitet i brugertests, der ikke på forhånd var tænkt på (eller var lavt prioriteret), men som i brug (ved tests) syntes nødvendig. Endelig er der funktionelle problemer som fx problemet med overførsel af Ruby scripts fra én computer til en anden. Dette er funktionalitet, der skal overvejes i fremtidige versioner. Men generelt kan frameworket og tilhørende protokolanalysatorer udvides på et utal af måder. Projektet har blot implementeret den funktionalitet, der blev fundet nødvendig i kravsanalysen.

Projektet har også givet andre erfaringer. Undervejs er det blevet mere og mere klart, at NetBee er i en udviklingsfase. Der er opstået en del fejl som følge af fejl i NetBee, men de har heldigvis ikke været mere alvorlige, end der kunne laves „work-arounds“ på dem. En anden ulempe ved NetBee har vist sig at være et stort pladsforbrug. Biblioteket og biblioteker, som NetBee er afhængig af, fylder ca. 3,2 mb, hvilket udgør ca. 84% af CAN protokolanalysatoren. Størstedelen af dette er i øvrigt den tilhørende XML parser (xerces), der fylder 2,3 mb (61%). Dog har biblioteket været tilstrækkeligt til at lave frameworket, men man kunne overveje at implemetere en NetPDL engine i C#. Derved sparer man som minimum XML parseren, da en sådan er en del af .NET frameworket.

På trods af de mangler, der er identificeret i prototypen, der er emner i videreudviklingen, og de problemer, der har været undervejs i forløbet, er resultatet et særdeles brugbart værktøj. Det, at frameworket er fremstillet uden afhængig af det netværk, det skal benyttes på, har allerede vist sin store værdi, da FOSS besluttede at skifte den benyttede bus i deres apparater. Og lignende eksempler må kunne findes overalt, hvor nye teknologier konstant udskifter gammel teknologi og gør gamle værktøjer ubrugelige. Men hvis det ikke kræver meget arbejde at ændre værktøjerne til de nye forhold, er vejen til en succesfuld implementering af den nye teknologi også kortere og dermed billiger.

Så selvom der „kun“ er fremstillet en prototype, har dels brugertests dels ovenstående betragtninger givet håb om, at vejen ikke er lang til et færdigt framework.

8.6 Virksomhedskonklusion på studenterprojekt

(Konklusion på projektet set fra virksomhedens (FOSS') side. Skrevet af Lars Jeppesen.)

Et værktøj til protokolanalyse er særdeles brugbart for FOSS, idet det er vigtigt at kunne analysere den interne kommunikation i vores apparater. Værktøjet er brugbart i forbindelse med vores omlægning fra at producere meget individuelle apparater til at producere apparater bestående af standardiserede moduler, idet modulerne benytter den samme ESWP platform. ESWP platformen specificerer en standard kommunikationsform, som derfor skal analyseres og diagnosticeres.

Et positivt sammenfald i Kristians afsluttende projekt i DTU regi, hans opsøgende natur og hans erfaring med lignende projekter sammen med FOSS' ønske om en protokolanalysator gjorde det naturligt at indlede et samarbejde omkring projektet. FOSS' krav til værktøjet kan opsummeres som følger:

- Sniffer og injektor netværksværktøj.
- Realtid.
- Brugervenligt.

Kristian fik til opgave at konstruere en egentlig kravspecifikation, der skulle danne basis for implementeringen af en prototype af værktøjet efter en analyse af projektets hovedproblemstillinger.

I analysen fandt Kristian i samråd med vejledere ud af, at FOSS havde et behov for et mere generelt framework til en protokolanalysator, så værktøjet stadig kunne bruges, hvis kommunikationsbussen i apparaterne blev skiftet. Denne opdeling viste sig særdeles relevan, da bussen under projektet blev skiftet fra en I^2C bus til en CAN bus.

På sigt regner vi med på FOSS, at frameworket til protokolanalysatoren kan hjælpe FOSS i udviklingen af ESWP platformen samt i udviklingen af applikationer til ESWP platformen. Værktøjet skal især bruges til fejlfinding og tests under udviklingen og dermed spare kostbar udviklingstid.

8.7 Endelig konklusion

Projektet har vist, at det er muligt at skabe et fleksibelt framework til protokolanalysatorer. I ovenstående blev der redegjort for, at det kan konkluderes, at projektets vision om at skabe et fleksibelt framework er opfyldt. Visionen om at det skulle resultere i et brugbart værktøj for FOSS er ligeledes opfyldt. Tests (brugertests og tekniske tests) har vist, at prototypen lever op til de stillede krav, men også at videreudvikling af prototypen er nødvendig.

APPENDIX A

Undersøgelser af brugerkrav

Indholdsfortegnelse

B.1 Resultater **95**
C.1 Ethernet-IP-TCP-HTTP stakken **97**
C.2 I^2C protokollen **108**
C.3 FOSS' CAN protokol **111**
D.1 Problemer ved definition af felt **117**
D.2 Problemer med visualiserings **118**
E.1 Domain **121**
E.2 Presentation **122**
E.3 PaInterface **123**
E.4 NetPDLwrapper **123**
E.5 Utils **124**

Appendikset giver et resume af sessioner, hvor brugerkrav er blevet undersøgt.

A.1 Interview med Mogens Velsing

Interviewet blev foretaget d. 28/9 2006 på FOSS. Interviewet blev holdt formelt i et mødelokalet, men møde formen var forholdsvis uformel. Interviewet blev holdt som beskrevet i afsnit 2.1.3. I dette afsnit specificeres de vigtigste konklusioner, der kan drages ud fra interviewet.

A.1.1 Om Mogens Velsing

Mogens Velsing (MV) har arbejdet med implementeringen af software i FOSS' apparater. Hans erfaring med protokol analyse har koncentreret sig om FOSS Electric NetSniffer/Emulator, der benyttes på ARCNET netværk. MV vejledede i det projekt, hvor programmerne blev udviklet.

A.1.2 Resultater af interviewet

I dette afsnit opsummeres de fundne krav fra MV. Kravene er delt i sniff specifikt og interaktion specifikt og yderligere kategoriseret som beskrevet i afsnit 2.1.3.

Sniffing krav

1. Skal have
    - Start/Stop af sniffingen
    - Oversigt over indkomne T123 beskeder (underliggende kommunikation er irrelevant)
    - Visning af detaljer for de enkelte beskeder.
    - Filtrering af beskeder
        - Vis alt der indeholder `filter` (se definition af `filter` nedenfor)
        - Vis intet, der indeholder `filter` (se definition af `filter` nedenfor)
        - `filter` kan fx være
            * Oprindelses/destinations adresse
            * Specifik type besked
            * Specifik data i et specifikt felt (fx T1 feltet indeholder '13')
            * Data der indeholder en specifik data sekvens
        - Simpel måde at definere filtrene på
    - Gem/hent mulighed af gamle optagelser
    - Fortolkning af T123 beskeder (fortolkning på de enkelte felter)
    - Oversættelse af binært datafelt (eller specificerede dele af) til ASCII tekst
2. Bør have
    - Alt, hvad der kan hjælpe på læseligheden af beskederne er en fordel (fx farvning)
    - „Word-agtig“ håndtering af filer
    - Flere åbne filer på én gang (godt til sammenligning)
    - Eksportering til gængse fil formater (csv, xml, m.m.)
3. Kan have
    - Definition af format af binært datafelt til automatisk oversættelse til specifik data
    - Automatisk sammenligning af filer
    - Understøttelse af udklipsholder
    - Fildump af datafelt
4. Har ikke
    - Print understøttelse (overflødiggøres, hvis der kan eksporteres)

Interaktion krav

1. Skal have
   - At svar kan afventes og der skal være let adgang til svarets detaljer
   - At svardata fra et modul kan benyttes i det efterfølgende script (på kørselstidspunkt)
   - Understøttelse af det simple, dvs. selvom det skal være fleksibelt, skal man stadig hurtigt kunne udføre simple operationer.
   - Definition af data i en besked vha. fil
   - Definition af data i en besked vha. tekst
2. Bør have
   - Et script skal kunne „afspilles“ fra alle steder

A.2 Uformelle interviews med ESWP udviklere

Interviewet er foregået som uformelle samtaler fordelt over hele analyse fasen. Det er sket i samarbejde med forskellige personer i gruppen, der hver i sær har kunnet bidrage med forskellige ting.

A.2.1 Resultater

Det fundne opsummeres i nedenstående. Resultaterne opdeles som i afsnit A.1.2.

Sniffing krav

1. Skal have
   - Detaljeret visning af de enkelte pakker på alle niveauer (ned til I^2C)
   - Timing på beskederne (som de eksisterede på netværket).
2. Bør have
   - Mulighed for let udvidelse af protokol specifikationen, så værktøjet hurtigt kan tage højde for ny funktionalitet i protokollen.

Interaktion krav

1. Skal have
   - Benyttelse af et eksisterende simpelt scriptsprog (fx Ruby), der kan interface til et `C` bibliotek.
2. Bør have
   - Afvikling af eksternt script (fra fil)
   - Hjælp til simple sessioner
3. Kan have
   - Indbygget editor, der kan vise/redigere/køre et script.

4. Har ikke
    - Selvopfundet script sprog (det har ikke fungeret med Emulator)

Appendix B

Foretagede brugertests

Appendikset opsummerer de foretagede brugertests.

B.1 Resultater

Kravstype	Beskrivelse
Nyt krav	Mulighed for at slå automatisk rul i pakkevisningen til/fra.
Udvidelse af eksisterende krav	Mulighed for at vise heltal decimalt og hexadecimalt
Nyt krav	Vise delta tider for pakker
Nyt krav (PA)	Gemme interface indstillinger
Manglende eksisterende krav	Husk feltvisningen ved skift til en pakke af samme type
Nyt krav	Linjenumre i indtastet script
Nyt krav	Undo/redo i interaktionsvisning
Nyt krav	Mulighed for indrykning af større tekstblok i interaktionsvisningen
Manglende eksisterende krav	Capture filters
Manglende eksisterende krav	Eksport til csv formatet
Nyt krav	Hurtig adgang til ofte brugte scripts, fx i en drop down liste.
Nyt krav	Markering i pakkevisningen af pakker, der stammer fra interaktion af et Ruby script.
Funktionelt problem	Ikoner til „styringsknapper“ til sniffing hhv. interaktion er ens og forvirrer derfor.
Funktionelt problem	Afhængighed mellem Ruby scripts og filter navne. Giver problemer ved flytning af scripts.

Fortsættes næste side

Kravstype	Beskrivelse
Funktionelt problem	Flere protokolanalysatorer kan ikke håndteres samtidig af frameworket. Filter filen og Ruby scripts er afhængige af NetPDL beskrivelsen, der skifter.

Appendix C

Eksempler på NetPDL protokol specifikationer

Indholdsfortegnelse

Resultater 97

Appendikset giver eksempler på protokol specifikationer i NetPDL sproget.

C.1 Ethernet-IP-TCP-HTTP stakken

Afsnittet viser definitionen af hhv. Ethernet, IP, TCP og HTTP protokollerne i NetPDL[1].

C.1.1 Ethernet

```
<?xml version="1.0" encoding="utf-8"?>

<netpdl name="Example1" version="1.0" date="08-02-2007">
  <protocol name="ethernet" longname="Ethernet">
    <format>
      <fields>
        <field type="fixed" name="dst" longname="MAC␣Destination"
            size="6"/>
        <field type="fixed" name="src" longname="MAC␣Source" size="6"
            />
        <field type="fixed" name="type" longname="Ethertype␣-␣Length"
            size="2"/>
      </fields>
    </format>

    <encapsulation>
```

[1] Uddrag af `NetPDL.xml.`, der følger med nbDevPack pakken - se afsnit 2.4

```
      <switch expr="buf2int(type)">
        <case value="0x800">
          <nextproto proto="#ip"/>
        </case>
      </switch>
    </encapsulation>
  </protocol>
</netpdl>
```

C.1.2 IP

```
<?xml version="1.0" encoding="utf-8"?>

<netpdl name="Example1" version="1.0" date="08-02-2007">
  <protocol name="ip" longname="Internet␣Protocol">
    <format>
      <fields>
        <field type="bit" name="ver" longname="Version" mask="0xF0"
            size="1"/>
        <field type="bit" name="hlen" longname="Header␣length" mask="
            0x0F" size="1"/>
        <field type="fixed" name="tos" longname="Type␣of␣service"
            size="1"/>
        <field type="fixed" name="tlen" longname="Total␣length" size=
            "2"/>
        <field type="fixed" name="identification" longname="
            Identification" size="2"/>
        <block name="ffo" longname="Flags␣and␣Fragment␣offset">
          <field type="bit" name="unused" longname="Unused" mask="0
              x8000" size="2"/>
          <field type="bit" name="df" longname="Don't␣fragment" mask=
              "0x4000" size="2"/>
          <field type="bit" name="mf" longname="More␣fragments" mask=
              "0x2000" size="2"/>
          <field type="bit" name="foffset" longname="Fragment␣offset"
               mask="0x1FFF" size="2"/>
        </block>
        <field type="fixed" name="ttl" longname="Time␣to␣live" size="
            1"/>
        <field type="fixed" name="nextp" longname="Next␣protocol"
            size="1"/>
        <field type="fixed" name="hchecksum" longname="Header␣
            Checksum" size="2"/>
        <field type="fixed" name="src" longname="Source␣address" size
            ="4"/>
        <field type="fixed" name="dst" longname="Destination␣address"
             size="4"/>

        <block name="option" longname="IP␣Options">
          <loop type="size" expr="(buf2int(hlen)␣*␣4)␣-␣20">
            <switch expr="buf2int($packet[$currentoffset:1])␣bitwand␣
                0x1F">
              <case value="0">
                <includeblk name="IP_OPT_EOL"/>
              </case>
              <case value="1">
                <includeblk name="IP_OPT_NOP"/>
              </case>
              <case value="2">
                <includeblk name="IP_OPT_SEC"/>
              </case>
              <case value="3">
```

```
          <includeblk name="IP_OPT_LSR"/>
        </case>
        <case value="4">
          <includeblk name="IP_OPT_TS"/>
        </case>
        <case value="5">
          <includeblk name="IP_OPT_EX_SEC"/>
        </case>
        <case value="7">
          <includeblk name="IP_OPT_RR"/>
        </case>
        <case value="8">
          <includeblk name="IP_OPT_SID"/>
        </case>
        <case value="9">
          <includeblk name="IP_OPT_SSR"/>
        </case>
        <case value="18">
          <includeblk name="IP_OPT_TR"/>
        </case>
        <case value="20">
          <includeblk name="IP_OPT_RA"/>
        </case>
        <default>
          <includeblk name="IP_OPT_UNK"/>
        </default>
      </switch>
    </loop>
  </block>
</fields>

<block name="IP_OPT_EOL" longname="End of Options List">
  <field type="fixed" name="type" longname="Option Type" size="
      1">
    <field type="bit" name="copy" longname="Copied flag" mask="
        0x80" size="1"/>
    <field type="bit" name="class" longname="Option Class" mask
        ="0x60" size="1"/>
    <field type="bit" name="number" longname="Option Number"
        mask="0x1F" size="1"/>
  </field>
  <field type="padding" name="padding" longname="Padding" align
      ="4"/>
</block>

<block name="IP_OPT_NOP" longname="No Operation Option">
  <field type="fixed" name="type" longname="Option Type" size="
      1">
    <field type="bit" name="copy" longname="Copied flag" mask="
        0x80" size="1"/>
    <field type="bit" name="class" longname="Option Class" mask
        ="0x60" size="1"/>
    <field type="bit" name="number" longname="Option Number"
        mask="0x1F" size="1"/>
  </field>
</block>

<block name="IP_OPT_SEC" longname="Security Option">
  <field type="fixed" name="type" longname="Option Type" size="
      1" >
    <field type="bit" name="copy" longname="Copied flag" mask="
        0x80" size="1"/>
```

```
      <field type="bit" name="class" longname="Option␣Class" mask
          ="0x60" size="1"/>
      <field type="bit" name="number" longname="Option␣Number"
          mask="0x1F" size="1"/>
    </field>
    <field type="fixed" name="length" longname="Length" size="1"/
        >
    <field type="fixed" name="sec" longname="Security" size="2"/>
    <field type="fixed" name="comp" longname="Compartments" size=
        "2"/>
    <field type="fixed" name="hr" longname="Handling␣Restrictions
        " size="2"/>
    <field type="fixed" name="tcc" longname="Transmission␣Control
        ␣Code" size="3"/>
  </block>

  <block name="IP_OPT_EX_SEC" longname="Extended␣Security␣Option"
      >
    <field type="fixed" name="type" longname="Option␣Type" size="
        1">
      <field type="bit" name="copy" longname="Copied␣flag" mask="
          0x80" size="1"/>
      <field type="bit" name="class" longname="Option␣Class" mask
          ="0x60" size="1"/>
      <field type="bit" name="number" longname="Option␣Number"
          mask="0x1F" size="1"/>
    </field>
    <field type="fixed" name="length" longname="Length" size="1"/
        >
    <field type="fixed" name="asecfc" longname="Additional␣
        Security␣Info␣Format␣Code" size="1"/>
    <field type="variable" name="asec" longname="Additional␣
        Security␣Info" expr="buf2int(length)␣-␣3"/>
  </block>

  <block name="IP_OPT_LSR" longname="Loose␣Source␣Routing␣Option"
      >
    <field type="fixed" name="type" longname="Option␣Type" size="
        1">
      <field type="bit" name="copy" longname="Copied␣flag" mask="
          0x80" size="1"/>
      <field type="bit" name="class" longname="Option␣Class" mask
          ="0x60" size="1"/>
      <field type="bit" name="number" longname="Option␣Number"
          mask="0x1F" size="1"/>
    </field>

    <field type="fixed" name="length" longname="Length" size="1"/
        >
    <field type="fixed" name="pointer" longname="Pointer" size="1
        "/>

    <block name="LSR_alist" longname="Address␣list">
      <loop type="size" expr="buf2int(length)␣-␣3">
        <field type="fixed" name="raddr" longname="Router␣Address
            " size="4"/>
      </loop>
    </block>
  </block>

  <block name="IP_OPT_SSR" longname="Strict␣Source␣Routing␣Option
      ">
```

```
    <field type="fixed" name="type" longname="Option␣Type" size="
        1">
      <field type="bit" name="copy" longname="Copied␣flag" mask="
          0x80" size="1"/>
      <field type="bit" name="class" longname="Option␣Class" mask
          ="0x60" size="1"/>
      <field type="bit" name="number" longname="Option␣Number"
          mask="0x1F" size="1"/>
    </field>

    <field type="fixed" name="length" longname="Length" size="1"/
        >
    <field type="fixed" name="pointer" longname="Pointer" size="1
        "/>

    <block name="SRR_alist" longname="Address␣list">
      <loop type="size" expr="buf2int(length)␣-␣3">
        <field type="fixed" name="raddr" longname="Router␣Address
            " size="4"/>
      </loop>
    </block>
  </block>

  <block name="IP_OPT_RR" longname="Record␣Route␣Option">
    <field type="fixed" name="type" longname="Option␣Type" size="
        1">
      <field type="bit" name="copy" longname="Copied␣flag" mask="
          0x80" size="1"/>
      <field type="bit" name="class" longname="Option␣Class" mask
          ="0x60" size="1"/>
      <field type="bit" name="number" longname="Option␣Number"
          mask="0x1F" size="1"/>
    </field>

    <field type="fixed" name="length" longname="Length" size="1"/
        >
    <field type="fixed" name="pointer" longname="Pointer" size="1
        "/>

    <block name="RR_alist" longname="Address␣list">
      <loop type="size" expr="buf2int(length)␣-␣3">
        <field type="fixed" name="raddr" longname="Router␣Address
            " size="4"/>
      </loop>
    </block>
  </block>

  <block name="IP_OPT_SID" longname="Stream␣Identifier␣Option">
    <field type="fixed" name="type" longname="Option␣Type" size="
        1">
      <field type="bit" name="copy" longname="Copied␣flag" mask="
          0x80" size="1"/>
      <field type="bit" name="class" longname="Option␣Class" mask
          ="0x60" size="1"/>
      <field type="bit" name="number" longname="Option␣Number"
          mask="0x1F" size="1"/>
    </field>
    <field type="fixed" name="length" longname="Length" size="1"/
        >
    <field type="fixed" name="sid" longname="Stream␣ID" size="2"/
        >
  </block>
```

```
<block name="IP_OPT_RA" longname="Router␣Alert␣Option">
  <field type="fixed" name="type" longname="Option␣Type" size="
      1">
    <field type="bit" name="copy" longname="Copied␣flag" mask="
        0x80" size="1"/>
    <field type="bit" name="class" longname="Option␣Class" mask
        ="0x60" size="1"/>
    <field type="bit" name="number" longname="Option␣Number"
        mask="0x1F" size="1"/>
  </field>
  <field type="fixed" name="len" longname="Length" comment="The
      ␣value␣of␣this␣field␣must␣be␣4" size="1"/>
  <field type="fixed" name="value" longname="Value" size="2"/>
</block>


<block name="IP_OPT_TR" longname="Trace␣Route␣Option">
  <field type="fixed" name="type" longname="Option␣Type" size="
      1">
    <field type="bit" name="copy" longname="Copied␣flag" mask="
        0x80" size="1"/>
    <field type="bit" name="class" longname="Option␣Class" mask
        ="0x60" size="1"/>
    <field type="bit" name="number" longname="Option␣Number"
        mask="0x1F" size="1"/>
  </field>

  <field type="fixed" name="len" longname="Length" size="1"/>
  <field type="fixed" name="id" longname="ID␣Number" size="2"/>
  <field type="fixed" name="ohc" longname="Outbound␣Hop␣Count"
      size="2"/>
  <field type="fixed" name="rhc" longname="Return␣Hop␣Count"
      size="2"/>
  <field type="fixed" name="TR_addr" longname="Originator␣IP␣
      Address" size="4"/>
</block>


<block name="IP_OPT_TS" longname="Timestamp␣Option">
  <field type="fixed" name="type" longname="Option␣Type" size="
      1">
    <field type="bit" name="copy" longname="Copied␣flag" mask="
        0x80" size="1"/>
    <field type="bit" name="class" longname="Option␣Class" mask
        ="0x60" size="1"/>
    <field type="bit" name="number" longname="Option␣Number"
        mask="0x1F" size="1"/>
  </field>

  <field type="fixed" name="length" longname="Length" size="1"/
      >
  <field type="fixed" name="pointer" longname="Pointer" size="1
      "/>
  <block name="ovf_flags" longname="Overflow␣and␣flag">
    <field type="bit" name="ts_ovf" longname="Overflow" mask="0
        xF0" size="1"/>
    <field type="bit" name="ts_flag" longname="Flag" mask="0x0F
        " size="1"/>
  </block>

  <block name="TS_list" longname="Timestamps">
```

```
          <loop type="size" expr="buf2int(length)␣-␣4">
            <switch expr="buf2int(ts_flag)">
              <case value="1">
                <field type="fixed" name="TS_addr" longname="Address"
                     size="4"/>
                <field type="fixed" name="TS_tstamp" longname="Time␣
                    Stamp" size="4"/>
              </case>
              <default>
                <field type="fixed" name="TS_tstamp" longname="Time␣
                    Stamp" size="4"/>
              </default>
            </switch>
          </loop>
        </block>
      </block>

      <block name="IP_OPT_UNK" longname="Unknown␣or␣Unsupported␣
          Option">
        <field type="fixed" name="type" longname="Option␣Type" size="
            1">
          <field type="bit" name="copy" longname="Copied␣flag" mask="
              0x80" size="1"/>
          <field type="bit" name="class" longname="Option␣Class" mask
              ="0x60" size="1"/>
          <field type="bit" name="number" longname="Option␣Number"
              mask="0x1F" size="1"/>
        </field>
        <field type="variable" name="OptionData" longname="Option␣
            Data" expr="(buf2int(hlen)␣*␣4)␣-␣$currentprotooffset"/>
      </block>
    </format>

    <encapsulation>
      <if expr="buf2int(foffset)␣==␣0">
        <if-true>
          <if expr="buf2int(nextp)␣==␣6">
            <if-true>
              <nextproto proto="#tcp"/>
            </if-true>
          </if>
        </if-true>
      </if>
    </encapsulation>
  </protocol>
</netpdl>
```

C.1.3 TCP

```
<?xml version="1.0" encoding="utf-8"?>

<netpdl name="Example1" version="1.0" date="08-02-2007">
  <protocol name="tcp" longname="TCP">
    <format>
      <fields>
        <field type="fixed" name="sport" longname="Source␣port" size=
            "2"/>
        <field type="fixed" name="dport" longname="Destination␣port"
            size="2"/>
        <field type="fixed" name="seq" longname="Sequence␣number"
            size="4"/>
```

```
<field type="fixed" name="ack" longname="Acknowledgement␣
    Number" size="4"/>
<field type="bit" name="hlen" longname="Header␣length" mask="
    0xF000" size="2"/>
<field type="bit" name="res" longname="Reserved␣(must␣be␣zero
    )" mask="0x0FC0" size="2"/>
<field type="bit" name="flags" longname="Flags" mask="0x003F"
     size="2">
  <field type="bit" name="urg" longname="Urgent␣pointer" mask
      ="0x0020" size="2"/>
  <field type="bit" name="ackf" longname="Ack␣valid" mask="0
      x0010" size="2"/>
  <field type="bit" name="push" longname="Push␣requested"
      mask="0x0008" size="2"/>
  <field type="bit" name="rst" longname="Reset␣requested"
      mask="0x0004" size="2"/>
  <field type="bit" name="syn" longname="Syn␣requested" mask=
      "0x0002" size="2"/>
  <field type="bit" name="fin" longname="Fin␣requested" mask=
      "0x0001" size="2"/>
</field>
<field type="fixed" name="win" longname="Window␣size" size="2
    "/>
<field type="fixed" name="crc" longname="Checksum" size="2"/>
<field type="fixed" name="urg" longname="Urgent␣Pointer" size
    ="2"/>

<block name="options" longname="TCP␣Options">
  <loop type="size" expr="(buf2int(hlen)␣*␣4)␣-␣20">
    <switch expr="buf2int($packet[$currentoffset:1])">
      <case value="0">
        <includeblk name="eol"/>
        <field type="padding" name="padf" longname="Padding"
            description="Field␣used␣to␣re-align␣the␣PDU␣to␣a␣
            word" align="4"/>
      </case>
      <case value="1">
        <includeblk name="noperation"/>
      </case>
      <case value="2">
        <includeblk name="mss"/>
      </case>
      <case value="3">
        <includeblk name="winscale"/>
      </case>
      <case value="4">
        <includeblk name="sackpermitted"/>
      </case>
      <case value="5">
        <includeblk name="sackformat"/>
      </case>
      <case value="8">
        <includeblk name="timestamp"/>
      </case>
      <case value="19">
        <includeblk name="md5signature"/>
      </case>
      <default>
        <includeblk name="unknown"/>
      </default>
    </switch>
  </loop>
```

```
  </block>
</fields>

<block name="eol" longname="End␣of␣Option␣List">
  <field type="fixed" name="endopt" longname="End␣of␣Option"
      size="1" description="Indicates␣the␣end␣of␣the␣option␣
      list"/>
</block>

<block name="noperation" longname="No␣Operation">
  <field type="fixed" name="type" longname="Type" description="
      This␣option␣code␣may␣be␣used␣between␣options" size="1"/>
</block>

<block name="mss" longname="Maximum␣Segment␣Size">
  <field type="fixed" name="type" longname="Type" size="1"/>
  <field type="fixed" name="length" longname="Option␣length"
      description="must␣be␣4" size="1"/>
  <field type="fixed" name="maxssize" longname="Maximum␣Segment
      ␣Size" size="2"/>
</block>

<block name="winscale" longname="TCP␣Windows␣Scale␣Option">
  <field type="fixed" name="type" longname="Type" description="
      Used␣to␣enable␣window␣scaling" size="1"/>
  <field type="fixed" name="length" longname="Option␣Length"
      description="must␣be␣3" size="1"/>
  <field type="fixed" name="shift.cnt" longname="Shift␣Count"
      description="If␣window␣scaling␣is␣enabled␣the␣sender␣will
      ␣right-shift␣its␣receive␣window␣values␣by␣'shift.cnt'"
      size="1"/>
</block>

<block name="sackpermitted" longname="Sack-Permitted␣Option">
  <field type="fixed" name="type" longname="Type" size="1"/>
  <field type="fixed" name="length" longname="Option␣Length"
      description="must␣be␣2" size="1"/>
</block>

<block name="sackformat" longname="Sack␣Option␣Format">
  <field type="fixed" name="type" longname="Type" size="1"/>
  <field type="fixed" name="length" longname="Option␣Length"
      description="(8␣x␣n_blocks)␣+␣2" size="1"/>
  <block name="Blocks␣Received" longname="Blocks␣Received">
    <loop type="times2repeat" expr="(buf2int(length)␣-␣2)␣div␣8
        ">
      <field type="fixed" name="leftedge" longname="Left␣Edge␣
          of␣Block" size="4"/>
      <field type="fixed" name="rightedge" longname="Right␣Edge
          ␣of␣Block" size="4"/>
    </loop>
  </block>
</block>

<block name="timestamp" longname="TCP␣Timestamp␣Option">
  <field type="fixed" name="type" longname="Type" size="1"/>
  <field type="fixed" name="length" longname="Option␣length"
      description="must␣be␣10" size="1"/>
  <field type="fixed" name="tsval" longname="Timestamp␣Value"
      size="4"/>
  <field type="fixed" name="tsechoreply" longname="Timestamp␣
      Echo␣Reply" size="4"/>
```

```
      </block>

      <block name="md5signature" longname="MD5␣Signature␣Option">
        <field type="fixed" name="type" longname="Type" size="1"/>
        <field type="fixed" name="length" longname="Option␣length"
            comment="must␣be␣18" size="1"/>
        <field type="fixed" name="md5digest" longname="MD5␣Digest"
            size="16"/>
      </block>

      <block name="unknown" longname="Unknown␣TCP␣Option">
        <field type="fixed" name="type" longname="Type" size="1"/>
        <field type="fixed" name="length" longname="Option␣length"
            size="1"/>
        <field type="variable" name="value" longname="Value" expr="
            buf2int(length)"/>
      </block>
    </format>

    <encapsulation>
      <if expr="buf2int(sport)␣==␣80">
        <if-true>
          <nextproto-candidate proto="#http"/>
        </if-true>
      </if>

      <if expr="buf2int(dport)␣==␣80">
        <if-true>
          <nextproto-candidate proto="#http"/>
        </if-true>
      </if>
    </encapsulation>
  </protocol>
</netpdl>
```

C.1.4 HTTP

```
<?xml version="1.0" encoding="utf-8"?>

<netpdl name="Example1" version="1.0" date="08-02-2007">
  <protocol name="http" longname="HTTP␣(Hyper␣Text␣Transfer␣Protocol)
      " showsumtemplate="http">
    <format>
      <fields>
        <if expr="($packet[$currentoffset␣:␣3]␣==␣'GET')␣or␣($packet
            [$currentoffset␣:␣4]␣==␣'POST')␣or␣($packet[$
            currentoffset␣:␣4]␣==␣'HTTP')">
          <if-true>
            <block name="header" longname="HTTP␣Header">
              <if expr = "($packet[$currentoffset␣:␣3]␣==␣'GET')␣or␣
                  ($packet[$currentoffset␣:␣4]␣==␣'POST')">
                <if-true>
                  <field type="line" name="cmdline" longname="Command
                      ␣Line">
                    <field type="tokenended" name="method" longname="
                        Method" endtoken="␣"/>
                    <field type="tokenended" name="url" longname="URL
                        " endtoken="␣"/>
                    <field type="line" name="version" longname="
                        Version"/>
                  </field>
                </if-true>
```

```
  <if-false>
    <field type="line" name="statusline" longname="
        Status␣Line">
      <field type="tokenended" name="version" longname=
          "Version" endtoken="␣"/>
      <field type="tokenended" name="statuscode"
          longname="Status␣Code" endtoken="␣"/>
      <field type="line" name="reasonphrase" longname="
          Reason␣Phrase"/>
    </field>
  </if-false>
</if>

<loop type="size" expr="$packetlength␣-␣$currentoffset"
    >
  <if expr="buf2int($packet[$currentoffset␣:␣2])␣==␣0
      x0D0A">
    <if-true>
      <field type="line" name="endheader" longname="End
          ␣Of␣Header"/>
      <loopctrl type="break"/>
    </if-true>
  </if>

  <switch expr="extractstring($packet[$currentoffset:␣
      0],␣'[^:]*',␣1,␣0)">
    <case value="'User-Agent'">
      <field type="line" name="useragent" longname="
          User␣Agent"/>
    </case>
    <case value="'Accept'">
      <field type="line" name="accept" longname="Accept
          ␣MIME␣Types"/>
    </case>
    <case value="'Accept-Language'">
      <field type="line" name="acceptlanguage" longname
          ="Accept␣Language"/>
    </case>
    <case value="'Server'">
      <field type="line" name="server" longname="Server
          " />
    </case>
    <case value="'Content-Type'">
      <field type="line" name="contenttype" longname="
          Content␣Type"/>
    </case>
    <case value="'Host'">
      <field type="line" name="host" longname="Host"/>
    </case>
    <case value="'Content-Encoding'">
      <field type="line" name="contentencoding"
          longname="Content␣Encoding"/>
    </case>
    <case value="'Content-Length'">
      <field type="line" name="contentlength" longname=
          "Content␣Length"/>
    </case>
    <case value="'Date'">
      <field type="line" name="date" longname="Date"/>
    </case>
    <case value="'Expires'">
```

```
                    <field type="line" name="expires" longname="
                        Expires"/>
                  </case>
                  <case value="'From'">
                    <field type="line" name="from" longname="From"/>
                  </case>
                  <case value="'If-Modified-Since'">
                    <field type="line" name="ifmodifiedsince"
                        longname="If Modified Since"/>
                  </case>
                  <case value="'Last-Modified'">
                    <field type="line" name="lastmodified" longname="
                        Last Modified"/>
                  </case>
                  <case value="'Location'">
                    <field type="line" name="location" longname="
                        Location"/>
                  </case>
                  <case value="'Pragma'">
                    <field type="line" name="pragma" longname="Pragma
                        "/>
                  </case>
                  <case value="'Referer'">
                    <field type="line" name="referer" longname="
                        Referer"/>
                  </case>
                  <case value="'WWW-Authenticate'">
                    <field type="line" name=" wwwauthenticate"
                        longname="WWW Authenticate"/>
                  </case>
                  <default>
                    <field type="line" name="option" longname="Option
                        "/>
                  </default>
                </switch>

              </loop>
            </block>
          </if-true>
        </if>

        <block name="header" longname="HTTP Object">
          <loop type="size" expr="$packetlength - $currentoffset">
            <field type="line" name="data" longname="HTTP data"/>
          </loop>
        </block>

      </fields>
    </format>
  </protocol>
</netpdl>
```

C.2 I²C protokollen

Nedenfor er I²C protokollen defineret i det format, der beskrevet i afsnit 5.1.

```
<?xml version="1.0" encoding="utf-8"?>

<netpdl name="FOSS' I2C protocols" version="0.9" creator="Kristian
    Kjær" date="15-11-2005">
  <protocol name="startproto" longname="Starting Protocol (used only
      for beginning the parsing)">
  <execute-code>
```

```
    <init>
    <variable name="$packetlength" type="number" validity="thispacket
        "/>
    <variable name="$currentoffset" type="number" validity="
        thispacket"/>
    <variable name="$packet" type="refbuffer" validity="thispacket"/>
    </init>
  </execute-code>

  <encapsulation>
    <nextproto proto="#i2c"/>
  </encapsulation>
  </protocol>

  <protocol name="i2c" longname="I2C">
  <execute-code>
    <init>
    <variable type="number" name="$addresslength" validity="
        thispacket"/>
    </init>
    <before>
    <if expr="buf2int($packet[0:1])␣ge␣0xF0">
      <if-true>
      <assign-variable name="$addresslength" value="10"/>
      </if-true>
      <if-false>
      <assign-variable name="$addresslength" value="7"/>
      </if-false>
    </if>
    </before>
  </execute-code>

  <format>
    <fields>
    <if expr="$addresslength␣==␣10">
      <if-true>
      <field type="bit" name="tenaddress1" longname="10␣bits␣address␣
          part␣1" mask="0x06" size="1" showtemplate="iic"/>
      <field type="bit" name="rw" longname="Read␣or␣Write" mask="0x01
          " size="1" showtemplate="iic"/>
      <field type="fixed" name="tenaddress2" longname="10␣bits␣
          address␣part␣2" size="1" showtemplate="iic"/>
      </if-true>
      <if-false>
      <field type="bit" name="sevenaddress" longname="7␣bits␣address"
           mask="0xFE" size="1" showtemplate="iic"/>
      <field type="bit" name="rw" longname="Read␣or␣Write" mask="0x01
          " size="1" showtemplate="iic"/>
      </if-false>
    </if>
    </fields>
  </format>

  <visualization>
    <showtemplate name="iic" showtype="dec" />
  </visualization>

  <encapsulation>
    <switch expr="buf2int(command)">
    <case value="0x1F">
      <nextproto proto="#hextet" />
    </case>
```

```
    <default>
      <nextproto proto="#octet"/>
    </default>
    </switch>
  </encapsulation>
  </protocol>

  <protocol name="octet" longname="Octet">
  <format>
    <fields>
    <field type="fixed" name="command" longname="Command" size="1"
        showtemplate="FieldHex"/>
    <field type="fixed" name="data" longname="Data" size="6"
        showtemplate="FieldHex"/>
    <field type="fixed" name="crc" longname="CRC" size="1"
        showtemplate="FieldHex"/>
    </fields>
  </format>

  <encapsulation>
    <nextproto proto="#t123"/>
  </encapsulation>
  </protocol>

  <protocol name="hextet" longname="Hextet">
  <format>
    <fields>
    <field type="fixed" name="command" longname="Command" size="1"
        showtemplate="FieldHex"/>
    <field type="fixed" name="data" longname="Data" size="12"
        showtemplate="FieldHex"/>
    <field type="fixed" name="packid" longname="Pack Identifier" size
        ="2" showtemplate="FieldHex"/>
    <field type="fixed" name="crc" longname="CRC" size="1"
        showtemplate="FieldHex"/>
    </fields>
  </format>

  <encapsulation>
    <nextproto proto="#t123"/>
  </encapsulation>
  </protocol>

  <protocol name="T123" longname="T123">
  <format>
    <fields>
    <field name="src" longname="Source" type="fixed" size="2"
        showtemplate="FieldHex" bigendian="yes"/>
    <field name="dest" longname="Destination" type="fixed" size="2"
        showtemplate="FieldHex" bigendian="yes"/>
    <field name="replyid" longname="Reply ID" type="fixed" size="1"
        showtemplate="FieldHex"/>
    <field name="t1" longname="T1" type="fixed" size="2" showtemplate
        ="FieldHex"/>
    <field name="t2" longname="T2" type="fixed" size="1" showtemplate
        ="FieldHex"/>
    <field name="t3" longname="T3" type="fixed" size="1" showtemplate
        ="FieldHex"/>
    </fields>
  </format>
  </protocol>
```

```
  <protocol name="defaultproto" longname="Other␣data">
  <format>
    <fields>
    <field type="variable" name="payload" longname="Data␣payload"
        expr="$packetlength␣-␣$currentoffset" showtemplate="
        Field4BytesHex"/>
    </fields>
  </format>
  </protocol>

  <visualization>
  <showsumstruct>
    <sumsection name="NUMBER" longname=""/>
  </showsumstruct>

  <showsumtemplate name="defaultproto">
  </showsumtemplate>
  <showtemplate name="Field4BytesHex" showtype="hex" showgrp="4"
      showsep="␣"/>
  </visualization>
</netpdl>
```

C.3 FOSS' CAN protokol

Nedenfor er CAN protokollen defineret i det format, der beskrevet i afsnit 6.2.1.

```
<?xml version="1.0" encoding="utf-8"?>

<netpdl name="Protocol␣Definition␣for␣FOSS␣CAN␣network" version="1.0"
      creator="Kristian␣Kjær" date="09-05-2007">
  <protocol name="startproto" longname="Starting␣Protocol␣(used␣only␣
      for␣beginning␣the␣parsing)">
  <execute-code>
    <init>
    <!-- NetPDL default variables - these have to be here, otherwise
        Netbee will fail -->
    <variable name="$linklayer" type="number" validity="static"/>
    <variable name="$jumpproto" type="number" validity="static"/>
    <variable name="$framelength" type="number" validity="thispacket"
        />
    <variable name="$packetlength" type="number" validity="thispacket
        "/>
    <variable name="$currentoffset" type="number" validity="
        thispacket"/>
    <variable name="$currentprotooffset" type="number" validity="
        thispacket"/>
    <variable name="$timestamp_sec" type="number" validity="
        thispacket"/>
    <variable name="$timestamp_usec" type="number" validity="
        thispacket"/>
    <variable name="$packet" type="refbuffer" validity="thispacket"/>
    <variable name="$nextproto" type="protocol" validity="thispacket"
        />

    <!-- Local variables -->
    <variable name="$canIdentifier" type="number" validity="
        thispacket"/>
    </init>
  </execute-code>

  <encapsulation>
```

```
  <switch expr="$jumpproto">
  <case value="1">
    <nextproto proto="#T123"/>
  </case>

  <default>
    <switch expr="$linklayer">
    <case value="2">
      <nextproto proto="#ESWP_CAN"/>
    </case>
    <default>
      <nextproto proto="#can"/>
    </default>
    </switch>
  </default>
  </switch>
</encapsulation>
</protocol>

<protocol name="can" longname="Controller␣Area␣Network" comment="
    Parses␣the␣CAN␣protocol␣as␣basically␣defined␣without␣
    interpretation␣of␣identifier">
<execute-code>
  <after>
  <assign-variable name="$canIdentifier" value="buf2int(identifier)
      "/>
  </after>
</execute-code>

<format>
  <fields>
  <field name="identifier" longname="Identifier" type="fixed" size=
      "4" showtemplate="FieldDec">
    <field name="tog" longname="Toggle" type="bit" mask="0x0000400"
        size="4" showtemplate="FieldDec"/>
    <field name="src" longname="Source" type="bit" mask="0x00003E0"
        size="4" showtemplate="FieldDec"/>
    <field name="dest" longname="Destination" type="bit" mask="0
        x000001F" size="4" showtemplate="FieldDec"/>
  </field>
  <field type="bit" name="flength" longname="Frame␣length" mask="0
      x001E" size="2" showtemplate="FieldDec" />
  <field type="bit" name="flags" longname="Flags" mask="0xFFE0"
      size="2" showtemplate="FieldBin">
    <field name="swOverrun" longname="Software␣overrun" type="bit"
        mask="0x8000" size="2" showtemplate="FieldDec"/>
    <field name="hwOverrun" longname="Hardware␣overrun" type="bit"
        mask="0x4000" size="2" showtemplate="FieldDec"/>
    <field name="txrq" longname="TX_REQ" type="bit" mask="0x1000"
        size="2" showtemplate="FieldDec"/>
    <field name="txack" longname="TX_ACK" type="bit" mask="0x0800"
        size="2" showtemplate="FieldDec"/>
    <field name="nerr" longname="NERR" type="bit" mask="0x0200"
        size="2" showtemplate="FieldDec"/>
    <field name="errFrame" longname="Error␣frame" type="bit" mask="
        0x0400" size="2" showtemplate="FieldDec"/>
    <field name="wakeup" longname="Wakeup␣message" type="bit" mask=
        "0x0100" size="2" showtemplate="FieldDec"/>
    <field name="ext" longname="Extended␣CAN" type="bit" mask="0
        x0080" size="2" showtemplate="FieldDec"/>
    <field name="std" longname="Standard␣CAN" type="bit" mask="0
        x0040" size="2" showtemplate="FieldDec"/>
```

```
      <field name="rtr" longname="Remote␣request" type="bit" mask="0
          x0020" size="2" showtemplate="FieldDec"/>
    </field>
    <field name="res" longname="Reserved" type="fixed" size="1"
        showtemplate="FieldHide"/>
    <field name="length" longname="Data␣length" type="bit" size="1"
        mask="0x70" showtemplate="FieldDec"/>
    <field name="type" longname="Message␣type" type="bit" size="1"
        mask="0x0F" showtemplate="FieldType"/>
    </fields>
  </format>

  <encapsulation>
    <nextproto proto="#defaultproto"/>
  </encapsulation>

  <!-- Workaround for bug in Netbee. The summary is not used, but
      this is poorly handled by Netbee -->
  <visualization>
    <showsumtemplate name="whatever" />
  </visualization>
  </protocol>

  <protocol name="ESWP_CAN" longname="ESWP␣CAN" comment="Parses␣the␣
      CAN␣protocol␣as␣basically␣defined␣with␣interpretation␣of␣
      identifier">
  <execute-code>
    <after>
    <assign-variable name="$canIdentifier" value="buf2int(identifier)
        "/>
    </after>
  </execute-code>

  <format>
    <fields>
    <field name="identifier" longname="Identifier" type="fixed" size=
        "4" showtemplate="FieldDec">
      <field name="tog" longname="Toggle" type="bit" mask="0x0000400"
          size="4" showtemplate="FieldDec"/>
      <field name="src" longname="Source" type="bit" mask="0x00003E0"
          size="4" showtemplate="FieldDec"/>
      <field name="dest" longname="Destination" type="bit" mask="0
          x000001F" size="4" showtemplate="FieldDec"/>
    </field>
    <field type="bit" name="flength" longname="Frame␣length" mask="0
        x001E" size="2" showtemplate="FieldDec" />
    <field type="bit" name="flags" longname="Flags" mask="0xFFE0"
        size="2" showtemplate="FieldBin">
      <field name="swOverrun" longname="Software␣overrun" type="bit"
          mask="0x8000" size="2" showtemplate="FieldDec"/>
      <field name="hwOverrun" longname="Hardware␣overrun" type="bit"
          mask="0x4000" size="2" showtemplate="FieldDec"/>
      <field name="txrq" longname="TX␣REQ" type="bit" mask="0x1000"
          size="2" showtemplate="FieldDec"/>
      <field name="txack" longname="TX␣ACK" type="bit" mask="0x0800"
          size="2" showtemplate="FieldDec"/>
      <field name="nerr" longname="NERR" type="bit" mask="0x0200"
          size="2" showtemplate="FieldDec"/>
      <field name="errFrame" longname="Error␣frame" type="bit" mask="
          0x0400" size="2" showtemplate="FieldDec"/>
      <field name="wakeup" longname="Wakeup␣message" type="bit" mask=
          "0x0100" size="2" showtemplate="FieldDec"/>
```

```
      <field name="ext" longname="Extended CAN" type="bit" mask="0
          x0080" size="2" showtemplate="FieldDec"/>
      <field name="std" longname="Standard CAN" type="bit" mask="0
          x0040" size="2" showtemplate="FieldDec"/>
      <field name="rtr" longname="Remote request" type="bit" mask="0
          x0020" size="2" showtemplate="FieldDec"/>
    </field>
    <field name="res" longname="Reserved" type="fixed" size="1"
        showtemplate="FieldHide"/>
    <field name="length" longname="Data length" type="bit" size="1"
        mask="0x70" showtemplate="FieldDec"/>
    <field name="type" longname="Message type" type="bit" size="1"
        mask="0x0F" showtemplate="FieldType"/>
    <!--<field name="pad" longname="Padding" type="padding" size="1"
        showtemplate="Hide"/>-->
    </fields>
  </format>

  <encapsulation>
    <switch expr="buf2int(type)">
    <case value="2">
      <nextproto proto="#multiss" />
    </case>
    <default>
      <nextproto proto="#defaultproto"/>
    </default>
    </switch>
  </encapsulation>

  <!-- Workaround for bug in Netbee. The summary is not used, but
      this is poorly handled by Netbee -->
  <visualization>
    <showsumtemplate name="whatever" />

    <showtemplate name="FieldType" showtype="hex">
    <showmap>
      <switch expr="buf2int(this)">
      <case value="0" show="Reserved"/>
      <case value="1" show="Single"/>
      <case value="2" show="Multiss"/>
      <case value="3" show="Segment"/>
      <case value="4" show="Reserved"/>
      <case value="5" show="Reserved"/>
      <case value="6" show="Reserved"/>
      <case value="7" show="Reserved"/>
      <case value="8" show="Linkupr"/>
      <case value="9" show="Linkupa"/>
      <case value="10" show="Ready"/>
      <case value="11" show="Flow start"/>
      <case value="12" show="Flow stop"/>
      <case value="13" show="Reserved"/>
      <case value="14" show="Reserved"/>
      <case value="15" show="Reserved"/>
      </switch>
    </showmap>
    </showtemplate>
  </visualization>
  </protocol>

  <protocol name="multiss" longname="Multi segment">
  <format>
    <fields>
```

```
    <field name="length" longname="Length" type="fixed" size="2"
        showtemplate="FieldDec" bigendian="yes"/>
    <field name="src" longname="Source" type="fixed" size="2"
        showtemplate="FieldHex" bigendian="yes"/>
    <field name="dest" longname="Destination" type="fixed" size="2"
        showtemplate="FieldHex" bigendian="yes"/>
    </fields>
  </format>
  </protocol>

  <protocol name="T123" longname="T123">
  <format>
    <fields>
    <field name="src" longname="Source" type="fixed" size="2"
        showtemplate="FieldHex"/>
    <field name="dest" longname="Destination" type="fixed" size="2"
        showtemplate="FieldHex"/>
    <field name="replyid" longname="Reply␣ID" type="fixed" size="1"
        showtemplate="FieldHex"/>
    <field name="t1" longname="T1" type="fixed" size="2" showtemplate
        ="FieldHex"/>
    <field name="t2" longname="T2" type="fixed" size="1" showtemplate
        ="FieldHex"/>
    <field name="t3" longname="T3" type="fixed" size="1" showtemplate
        ="FieldHex"/>
    </fields>
  </format>
  </protocol>

  <protocol name="defaultproto" longname="Payload">
  <format>
    <fields>
    <field type="variable" name="payload" longname="Data␣payload"
        expr="$packetlength␣-␣$currentoffset" showtemplate="
        Field4BytesHex"/>
    </fields>
  </format>
  </protocol>

  <visualization>
  <!-- Workaround for bug in Netbee. The summary is not used, but
      this is poorly handled by Netbee -->
  <showsumstruct>
    <sumsection name="whatever"/>
  </showsumstruct>

  <showtemplate name="Field4BytesHex" showtype="hex" showgrp="4"
      showsep="␣"/>
  <showtemplate name="FieldBin" showtype="bin"/>
  <showtemplate name="FieldDec" showtype="dec"/>
  <showtemplate name="FieldHex" showtype="hex"/>
  <showtemplate name="FieldHide" showtype="hide"/>
  </visualization>
</netpdl>
```

Appendix D

NetPDL problemer

I dette afsnit præsenteres nogle af de problemer, der er opstået i projektet på baggrund af brugen af NetBee biblioteket. Det præsenteres ligeledes, hvilke „work-arounds“, der kan laves for at løse problemet.

D.1 Problemer ved definition af felt

- Fejl værdi i „type“ attributten i et felt giver udefineret fejl. Så tjek denne attribut nøje!
- Man kan ikke have et bit felt med nestede felter og så et bit felt bagefter. Men det virker, hvis det andet bitfelt skrives først i stedet, fx virker dette ikke:

```
<field type="bit" name="flags" longname="Flags" mask="0xFFE0"
     size="2">
  <field name="nest" longname="Nested field" type="bit" mask="0
      x0020" size="2"/>
</field>
<field type="bit" name="length" longname="Length" mask="0x001E"
     size="2" />
```

  Men dette gør:

```
<field type="bit" name="length" longname="Length" mask="0x001E"
     size="2" />
<field type="bit" name="flags" longname="Flags" mask="0xFFE0"
     size="2">
  <field name="nest" longname="Nested field" type="bit" mask="0
      x0020" size="2"/>
</field>
```

  Eneste forskel er rækkefølgen, felterne sendes til frameworket i, dvs. de vil optræde i samme rækkefølge der.

- Der er problemer med håndteringen af bit felter. Betragt følgende

```
<field type="bit" name="flags" longname="Flags" mask="0x01FF"
    size="2">
  <field name="nest" longname="Nested␣field" type="bit" mask="0
      x0001" size="2"/>
</field>
<field name="another" longname="Another␣field" type="bit" mask="
    0xDD" size="1"/>
```

Feltet, „another“, virker ikke. Det kan løses ved at enten droppe indlejringen af felter eller ved at indsætte et „pseudo“-felt.

```
<field type="bit" name="flags" longname="Flags" mask="0x01FF"
    size="2">
  <field name="nest" longname="Nested␣field" type="bit" mask="0
      x0001" size="2"/>
</field>
<field name="pseudo" longname="Pseudo␣field" type="fixed" size="
    1" showtemplate="FieldHide"/>
<field name="another" longname="Another␣field" type="bit" mask="
    0xDD" size="1"/>
```

Den løsning kræver selvfølgelig, at man kan få byte strømmen til at overholde dette format (altså indsætte en „tom“ byte, der ikke benyttes eller vises.

- Fejl i BigEndian håndtering i sammenhæng med decimal visning. Følgende vises forkert:

```
<field type="fixed" name="t1" longname="T1" size="2" bigendian="
    yes" showtemplate="FieldDec"/>
<visualization>
  <showtemplate name="FieldDec" showtype="dec"/>
</visualization>
```

Den ene af de to bytes i T1 feltet benyttes ikke i visningen, dvs. fx bliver 0xEEFF til 0xEE00. Begge nedenstående eksempler er dog korrekte:

```
<field type="fixed" name="t1" longname="T1" size="2"
    showtemplate="FieldDec"/>
<visualization>
  <showtemplate name="FieldDec" showtype="dec"/>
</visualization>
```

```
<field type="fixed" name="t1" longname="T1" size="2" bigendian="
    yes" showtemplate="FieldHex"/>
<visualization>
  <showtemplate name="FieldHex" showtype="hex"/>
</visualization>
```

Der er ikke en god „work-around“ til problemet udover at undgå kombinationen af decimal visning og big-endian format.

- Der er generelt forvirring omkring big-endian og little-endian. Sættes attributten, `bigendian` i `<Field>` til „yes“, får man en little-endian fortolkning, mens en „no“ værdi giver en big-endian fortolkning. Dette er meget forvirrende!

D.2 Problemer med visualiserings

- I den globale `<visualization>` skal der være en `<showsumstruct>` definition – også selvom den ikke bruges, fx

```
<showsumstruct>
  <sumsection name="whatever"/>
</showsumstruct>
```

- I `startproto` protokollen skal der være et ikke-tomt `<visualization>` element – også selvom det ikke benyttes, fx

```
<visualization>
  <showsumtemplate name="whatever" />
</visualization>
```

Appendix E

Oversigt over klasser i systemet

Indholdsfortegnelse

Problemer ved definition af felt **119**
Problemer med visualiserings **120**

I dette appendiks gives en oversigt over alle fremstillede typer i systemet. Tabeller viser hvilke typer, der ligger i hvilke moduler. Typerne er i tabellerne markeret med et bogstav, der angiver hvilken type, der er tale om. Tabel E.1 viser, hvad de enkelte forkortelser betyder. Hvis der ikke er angivet en forkortelse, er typen en almindelig klasse.

Forkortelse	Betydning
a	Abstrakt klasse
d	Delegate definition
e	Enumerator
i	Interface

Tabel E.1: Forkortelser brugt på typerne.

Udover en angivelse af typens art, viser tabellerne også, hvor en given type kommer fra, dvs. hvilken type, den extender. Dette markeres med C# notation.

De følgende fem afsnit giver en oversigt over typer i de fem fremstillede assemblies.

E.1 Domain

Modul	Klasse
Pakkedatabase ť	PacketSetOperation
	PacketDatabase
	MatchNotification (d)
	PacketListener
Pakke	Packet
Pakkeparser	PacketParser
Protocoldatabase	*Fra NetPDLwrapper*
Filter	FilterCollection (e)
	CollectionOperation (e)
	FilterOperation
	Filter
	FilterExpr (a)
	BinaryOperator (e)
	BinaryFilterExpr : FilterExpr
	IValueContainer (i)
	Statement : FilterExpr
	FieldStatement : Statement
	ValueFieldStatement<T> : FieldStatement, IValueContainer
	NumberFieldStatement : ValueFieldStatement<byte[]>
	FloatStatement : ValueFieldStatement<double>
	BinaryFieldStatement : NumberFieldStatement, IValueContainer
	ASCIIFieldStatement : ValueFieldStatement<String>
	StatementOperator
	StatementFactory
Filterdatabase	FilterDatabase : Dictionary<String, Filter>
Ruby script eksekverer	OutputType (e)
	RubyExecutor
	RubyOutputEventArgs : EventArgs
Ruby interface	MethodIds (e)
	Errors : byte (e)
	RubyInterface
Ruby script generator	RubyScriptCreator
	Writer : IDisposable
Ruby syntaks læser	RubySyntaxReader
	TokenData
	TokenType (e)
IO handler	FileOperations (e)
	FileHandler (a)
	IOHandler
	BinaryHandler : FileHandler
	PDMLHandler : FileHandler
	RubyHandler : FileHandler
	FileExtension
	PersistentFilters

E.2 Presentation

Modul	Klasse
Tabulator visning	TabView : TabControl
	ExtendedTabpage : TabPage (a)
Sniffing visning	SniffingView : ExtendedTabpage
Interaction visning	InteractionTab : ExtendedTabpage
Pakkevisning	MessageViewList : ListView
Detaljevisning	FieldPanel : SplitContainer
	FieldView : TreeView
	DetailView : FlowLayoutPanel
	ProtocolNode : TreeNode
	FieldNode : TreeNode

Fortsættes næste side

Modul	Klasse
Datavisning	HexCasing (e)
	Notation (e)
	BytePositionInfo
	HexBox : Control
	NativeMethods
	IByteProvider (i)
	HexFontEditor : FontEditor
	ByteCollection : CollectionBase
	DynamicByteProvider : IByteProvider
	FileByteProvider : IByteProvider, IDisposable
Filtervisning	FilterView : ListView
	FilterItem : ListViewItem
	FilterHandler
Script editor	CodeWindow : RichTextBox
Outputbokse	*TextBox*
Kodehjælper	DynamicFieldPanel : FlowLayoutPanel, IEnumerable<Field-Definition>
	FieldDefinitionEnumerator : IEnumerator<FieldDefinition>
Værktøjslinjer	SaveMode (e)
	ToolBase : ToolStrip (a)
	ToolBars (e)
	ToolBar : ToolStripPanel
Menu bar	MenuBar : ToolBase
Interfacevælger	KangarooSelector : Form
Filter editor	FilterEditor : Form
	PartialFilterList : ListBox
	BorderPanel : Panel
IO handler	IOHandler

E.3 PaInterface

Modul	Klasse
PaInterface	Kangaroo (a)
	KangarooErrors (e)
	KangarooError : Exception
	INetworkInterface (i)
	IReceiveInterface : INetworkInterface
	ISendInterface : INetworkInterface
	RawPacket

E.4 NetPDLwrapper

Modul	Klasse
NetPDL protokoldatabase	FieldState (e)
	ProtocolDB : Dictionary<String, ProtocolDefinition>
	BaseDefinition
	FieldType (e)
	ProtocolDefinition : BaseDefinition
	FieldDefinition : BaseDefinition
	ValuePair
	PDMLReader
	NetPDLengineException : Exception
	NetPDLengineWarning : Exception
NetPDL pakkeparser	Parser
	ParsedPacket

Fortsættes næste side

Modul	Klasse
	ParsedBase ParsedProtocol : ParsedBase ParsedField : ParsedBase BasicHeader

E.5 Utils

Modul	Klasse
Ikke tilknyttede typer	CompareFunction (d) ArrayCompare BigInteger Updater (d) DelayUpdater EventArgs<T> : EventArgs NetworkMode (e) Set<T> : ICollection<T> TimeUtils

Litteratur

[1] Kent Beck. *Test-Driven Development - By Example*. Addison-Wesley, 2002.

[2] Martin Fowler. *Patterns of enterprise application architecture*. Addison Wesley, 2003.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison Wesley, 1995.

[4] Sune Gynthersen. Developing a version control system using test-driven development. *IMM-B.Eng-2006-15*, pages 1–67, 2006.

[5] Gordon Hogenson. *C++/CLI: The Visual C++ Language for .NET*. Apress, 2006.

[6] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson, 2003.

[7] Craig Larman. *Applying UML and patterns; an introduction to object-oriented analysis and design and iterative development*. Prentice Hall Professional Technical Reference, 2005.

[8] Hans Henrik Løvengreen. Basic concurrency theory. *IMM, Note for course 02152*, ver. 1.1, 2005.

[9] K. U. Mätzel and W. R. Bischofberger. Evolution of object systems - how to tackle the slippage problem. *Computer Science Research at Ubilab, Research Projects 1995/96; Proceedings of the Ubilab Conference '96,*, pages 99–119, 1996.

[10] Charles Petzold. *Programming Microsoft Windows with C#*. Microsoft, 2002.

[11] J. Preece. *Interaction design : beyond human-computer interaction*. John Wiley & Sons, Inc., New York., 2002.

[12] F. Risso and M. Baldi. Netpdl: An extensible xml-based language for packet header description. *Computer Networks*, 50(5):688–706, 2006.

[13] `http://www.foss.dk/AboutFOSS.aspx`.

[14] `http://www.nbee.org/Docs/NetPDL/`.

[15] `http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf`.