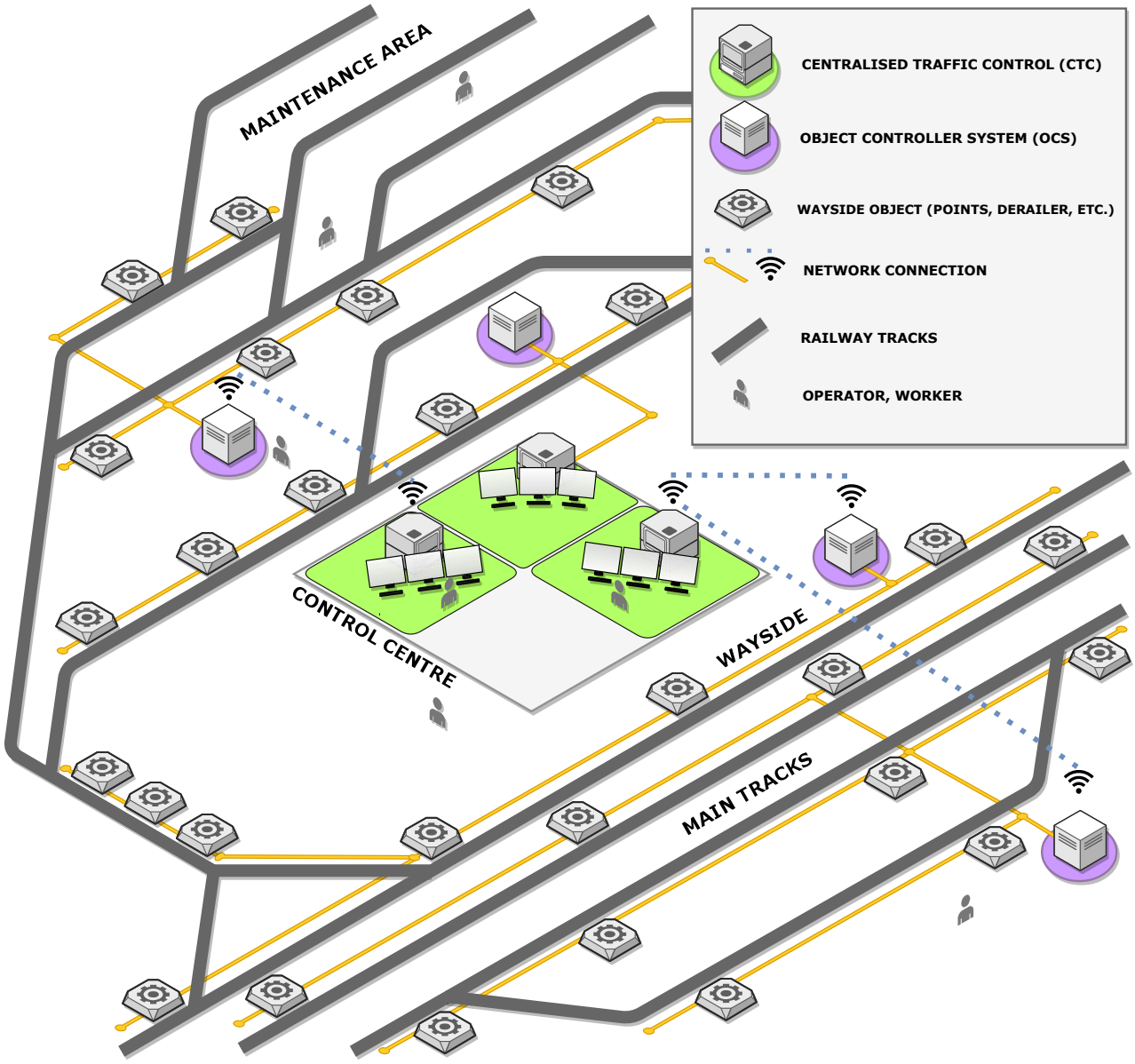
CHALMERS



Datakommunikation för signalsystem

Examensarbete inom data- och informationsteknik

QUANG LUONG

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
Göteborg, Sverige 2016

EXAMENSARBETE

Datakommunikation för signalsystem

Quang Luong

CHALMERS

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2016

Datakommunikation för signalsystem

© QUANG LUONG, 2016

Examinator: Peter Lundin

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
412 96 Göteborg, Sverige
Telefon: 031-772 1000

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Omslag:
Bild av signalsystemets anslutna enheter i ett möjligt, fiktionellt scenario. Ikoner motsvarar inte enheternas fysiska form.

Institutionen för data- och informationsteknik
Göteborg 2016

Datakommunikation för signalsystem

QUANG LUONG

Institutionen för data- och informationsteknik
Chalmers tekniska högskola

Sammanfattning

Bombardiers signalsystem för järnväg, INTERFLO 150, består av flera delsystem som interagerar med varandra. Delsystemen inkluderar Centralised Traffic Control (CTC), Traffic Control Centre (TCC) och Object Controller System (OCS). Dessa tre delsystem tillåter fjärrstyrd kontroll av objekt längs järnvägsspår. Detta arbete är del av ett projekt som gick ut på att undersöka möjligheterna med samma kontroll, utan mellanliggande TCC för att skapa ett simplare signalsystem. Detta system skulle innebära en förmånligare lösning där det komplexa säkerhetssystemet TCC skulle vara överflödigt. Systemet skulle även kunna användas för att underlätta styrning av objekt, eftersom ett komplext mellansteg inte längre skulle behövas. Detta kan underlätta för t.ex. testning och användningsområden där ökad manuell styrning är önskvärd. Detta arbete skapade direkt kommunikation mellan CTC och OCS. För att möjliggöra detta utvecklades en kommunikationsmodul baserad på nätverksprotokollet vilket OCS använder för att kommunicera med TCC. Därefter integrerades denna modul i CTC. Lösningen har utvecklats i programmeringspråket Java, med Eclipse som utvecklingsmiljö. Projektet har utförts hos Bombardier Transportation Rail Control Solutions i Göteborg, Sverige.

Nyckelord: Styrning, styrsystem, signalsystem, Bombardier, Bombardier Transportation, Rail Control Solutions, Centralised Traffic Control, Traffic Control Centre, Object Controller System, Java, C#, nätverk, datakommunikation.

Data communication for railway signalling system

QUANG LUONG

Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Bombardier's railway signalling system INTERFLO 150, consists of several subsystems interacting with each other. The subsystems include Centralised Traffic Control (CTC), Traffic Control Centre (TCC) and Object Controller System (OCS). These three subsystems allow for remote control of wayside objects. This work is part of a project which sought to allow the same control without intermediate TCC to create a simpler signalling system. This system would entail a more affordable solution where the complex security features of TCC would be excessive. This system would also have uses that would allow easier control of wayside objects, by not being restricted by a complex intermediate step. This would make e.g. testing more efficient, as well as have use cases where increased manual control is desirable. This work created direct communication between CTC and OCS. This was done by developing a communication module based on the network protocol which OCS is using to communicate with TCC. This module was then integrated into CTC. The solution was developed using the Java programming language, using Eclipse as integrated development environment. This project was carried out at Bombardier Transportation Rail Control Solutions in Gothenburg, Sweden.

Keywords: Control, control system, railway signalling system, Bombardier, Bombardier Transportation, Rail Control Solutions, Centralised Traffic Control, Traffic Control Centre, Object Controller System, Java, C#, network, data communication.

Förord

Detta arbete har utförts för LMTX38 Examensarbete vid data- och informationsteknik på Chalmers tekniska högskola. Projektet är ett uppdrag för Bombardier Transporation Rail Control Solutions i Göteborg, där projektet främst har utvecklats. För detta arbete vill jag tacka Johan Sjöberg och Linn Leiulfsrud för samarbetet under projektets gång. Sjöberg och Leiulfsrud har parallellt arbetat på styrlogik inom samma signalsystem. Från Bombardier tackar jag Torbjörn Hildesson, som gjort projektet möjligt, Anders Palmér som satte mig in i arbetets kontext och bidragit med stöd för OCS, och Nicola Bottini som gett tekniskt stöd för CTC. Från Chalmers tekniska högskola tackar jag Christer Carlsson som handlett mig genom examensarbetet.

Förkortningar och terminologi

Akronym	Term	Förklaring
API	Application Programming Interface	Gränssnitt för samling kod/applikation som kan användas för att ge viss funktionalitet.
CCU	Communication Controller Unit	Bombardiers term för komponent i OCS vilket där hanterar centrala kommunikationen.
COC	CTC-OCS Communicator	Kommunikationsmodul baserad på FFFIS TCC-OCS. Hanterar kommunikation för CTC mot OCS.
CRC	Cyclic Redundancy Check	En metod vilket beräknar koder för att upptäcka möjliga fel efter sändning.
CTC	Centralised Traffic Control	Bombardiers grafiska program med vilket man övervakar och styr anläggningens olika objekt, inklusive spår, växlar, spårspärrar m.m.
FFFIS	Form Fit Function Interface Specification	Specifikation för gränssnitt för ett ämne. T.ex. innehåller FFFIS TCC-OCS specifikationen för kommunikation mellan TCC och OCS.
GUI	Graphical user interface	Grafiskt användargränssnitt.
OC	Object Controller	Bombardiers term för komponent i OCS som direkt styr objekt längs järnvägsspår. Se figur 2.1.
OCS	Object Controller System	Bombardiers term för det delsystem med objekt och relaterat som styr längs järnvägsspåren.

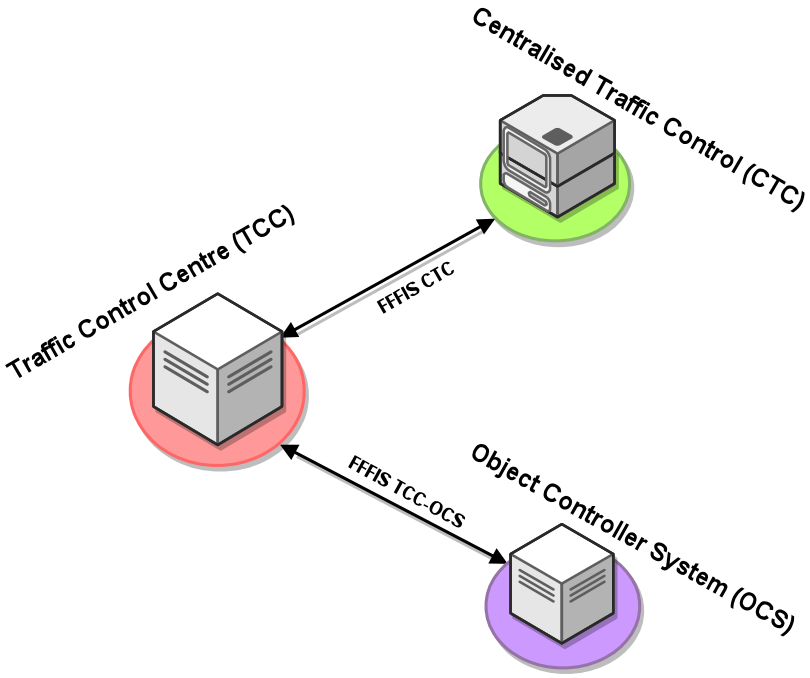
RCS	Rail Control Solutions	Division av Bombardier Transportation.
TCP/IP	Transmission Control Protocol/Internet Protocol, eller Internet Protocol Suite	Internets samling av kommunikationsprotokoll. Används även för andra nätverk.
TCC	Traffic Control Centre	Bombardiers centrala system som verifierar all input och output mellanliggande flertal system, inklusive CTC och OCS.
VLAN	Virtual Local Area Network	Lokalt nätverk utan fysisk form eller fysiska enheter, utan är simulerad i mjukvara.
XML	Extensible Markup Language	Universellt märkspråk. Används för att strukturera data.

Innehållsförteckning

1 Inledning

1.1 Bakgrund

Bombardier Transportations division Rail Control Solutions (RCS) i Göteborg utvecklar säkerhetskritiska signalsystem. RCS Göteborgs produkter inkluderar INTERFLO 150 och 550 [1]. INTERFLO 150 innefattar, bland andra, delsystemen Centralised Traffic Control (CTC), Traffic Control Centre (TCC) och Object Controller System (OCS). OCS styr och förmedlar status över objekt längs järnvägsspår. TCC är det centrala delsystem som verifierar all input och output mellan alla de andra delsystemen. TCC ansvarar för all säkerhetslogik för hela signalsystemet. CTC är ett program med grafiskt användargränssnitt vilket förmedlar operatörens förfrågningar för att styra objekten längs spår och visar en översikt av anläggningen med samtliga objekt inklusive spår, växlar, spärrar, etc.



Figur 1.1. *Förbindelser mellan CTC, TCC och OCS.*

Bombardier vill undersöka möjligheterna med att styra OCS direkt med CTC, istället för att gå genom den intermediära TCC:n. Detta är tänkt att ha tillämpningar i kundernas verkstäder där Bombardier för närvarande inte har lämpliga lösningar. Med denna variant av systemet, utan TCC, skulle förmånligare lösningar kunna tas fram för de användningsområden där TCC är för dyr och/eller komplex.

1.2 Syfte

Syftet med detta projekt är att undersöka möjligheterna, d.v.s. bekräfta att det är möjligt i form av ett *proof of concept*, med ett system utan TCC. Detta system tänks ha tillämpningar i lokverkstäder som i dagsläget ligger utanför Bombardiers kontroll. Idén för konceptet är inspirerat av saknad kontroll av spår inom dessa verkstadsområden, mer specifikt i Luossavaara-Kiirunavaara AB:s (LKAB:s) automatiserade gruvverksamhet i Kiruna, Sverige. I gruvan bidrar Bombardiers lösning INTERFLO 150 till autonom styrning av tåg [2].

1.3 Mål

Projektet går ut på att utesluta det centrala TCC-systemet för att tillåta lokal kontroll av OCS med CTC. Då alla andra delsystem är utvecklade för att kommunicera med TCC, blir alla nuvarande kommunikationskanaler oanvändbara. Ny kommunikation kommer att behöva skapas direkt mellan CTC och OCS, vilket är målet för detta arbete. Borttagning av TCC innebär även förlust av den centrala säkerhetskritiska och automatiserade funktionaliteten. Ersättning av automatiska ordrar behålls genom utveckling av säkerhetslogik/styrlogik i ett parallellt arbete [3].

Målet för projektet är därmed uppdelat i två arbeten. Den ena ansvarar för kommunikation och den andra för styrlogik. Tillsammans skall de två arbetena utgöra en funktionell prototyp av ett enklare signalsystem som tillåter lokal kontroll av OCS, utan det centrala TCC-systemet.

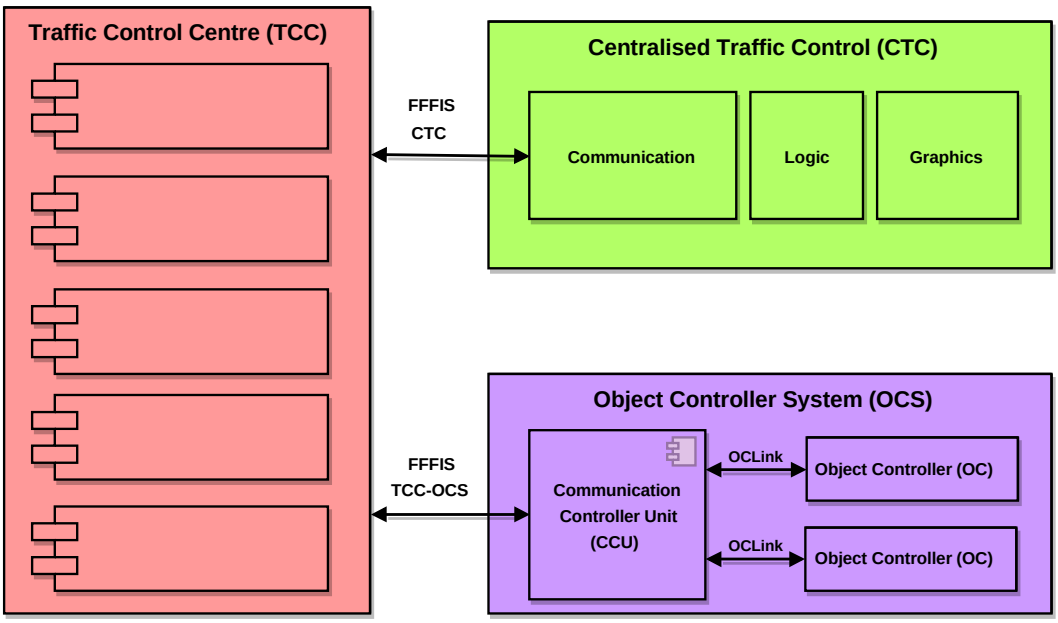
1.4 Avgränsningar

Detta arbete avgränsas till kommunikation mellan CTC och OCS. Detta innebär implementering av kommunikationsmodulen för CTC som tillåter OCS att ansluta sig till den, samt utökad funktionalitet av CTC för att anpassas till den nya kommunikationen.

Uppgifterna som skall utföras är översiktligt redan skapade innan arbetets början, vilka skall uppfyllas. Stora avvikande förslag och metoder från uppgifterna anammas inte. Arbetet skall inte beröra hårdvaran i OCS. OCS-mjukvara skall användas för testning. Analys av komponenter i OCS kan även bidra till ökad förståelse av systemets kommunikation.

2 Teknisk bakgrund

Arbetet berör ett styrsystem/signalsystem av flera delsystem och deras interaktioner. Förbindelserna mellan delsystemen specifieras i så kallade *Form Fit Function Interface Specifications*. Delsystemen och specifikationerna för kommunikation beskrivs i detta kapitel.



Figur 2.1. *Förbindelser mellan de olika delsystemen TCC, CTC och OCS med vissa interna detaljer.*

2.1 Centralised Traffic Control (CTC)

Centralised Traffic Control är det program med grafiskt användargränssnitt (GUI) som ger en översikt av ett helt område (även kallad *site*) med flertals olika objekt specificerade i konfigurationsfiler. Detta program kommunicerar med Traffic Control Centre och har fler användningsområden än styrning av objekt längs järnvägsspår. Detta program är skrivet i Java.

2.2 Object Controller System (OCS)

Object Controller System är en samling av flertal komponenter, bl.a. Communication Controller Unit (CCU), som hanterar den centrala kommunikationen i OCS. När kommunikation med OCS sker, är det mer precist CCU som menas. I arbetets berörda OCS används CCU6, vilket utvecklats för att ersätta äldre CCU2 [4].

Utöver CCU finns även flertal Object Controllers (OC:s) i OCS, vilka är ansvariga för att direkt kontrollera objekt längs järnvägsspår. Objekt som styrs av OC:s kan vara spårspärrar, växlar, signalljus, bommar, etc. Varje OC har i uppgift att ta hand om ett objekt. CCU tar

hanterar upp till sexton OC:s, och varje OCS innehåller en CCU-enhet. CCU är utvecklat i Linux med programmeringspråket C.

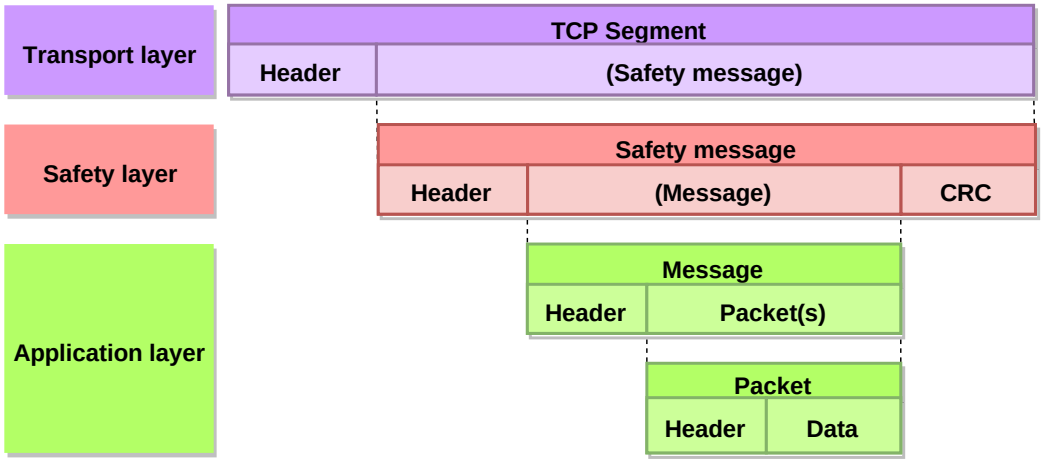
2.3 Traffic Control Centre (TCC)

Traffic Control Centre är det centrala system som verifierar all input och output mellan alla de andra systemen för att leva upp till bestämda säkerhetskrav. TCC ansvarar för *interlocking*. TCC har i uppgift att automatiskt styra objekt vid olika händelser. Detta inkluderar till exempel lyftande av bom när tåg passerat järnvägskorsning (*level crossing*). Detta arbete syftar på att utvecklat ett signalsystem utan TCC.

2.4 Form Fit Function Interface Specification (FFFIS) TCC-OCS

Form Fit Function Interface Specification TCC-OCS är det protokoll som beskriver kommunikationen mellan TCC och OCS. Detta protokoll skall användas för kommunikation mellan CTC och OCS i detta arbete. Protokollet beskriver flera skikts dataenhet, även kallad Protocol Data Unit (PDU), samt behandling av dataenheterna över respektive skikt. Skikten i specifikationen använder sig av skikt liknande de i TCP/IP och OSI-modellerna. Specifikationen behandlar applikationsskikt och transportskikt, med ett eget definierat mellanliggande trygghetsskikt (*safety layer*). Även ett säkerhetsskikt (*security layer)* behandlas för att enkryptera kommunikationen på öppna nätverk mot säkerhetsintrång [5].

På transportskiktet används TCP strömmar, vilket innebär att transportskiktets PDU är segment. På trygghetsskiktet definieras en PDU bestående av en header, ett meddelande, samt en CRC och kallas trygghetsmeddelande (*safety message*). Meddelandet i trygghetsskiktet är ej obligatoriskt, utan beror på typen av trygghetsmeddelande. På applikationskiktet kallas en PDU för meddelande (*message*), vilket innehåller en header samt ett eller fler paket (*packet*). Ett paket är inte en självständig PDU, men innehåller sin egen header och beroende på paketets typ finns olika data för olika objekt. Figur 2.2 sammanfattar PDU:erna.



Figur 2.2. *Transport-, trygghet- och applikationskiktens respektive PDU-format.*

2.5 Form Fit Function Interface Specification (FFFIS) CTC

Form Fit Function Interface Specification CTC är specifikationen som beskriver kommunikationens gränssnitt för CTC. FFFIS CTC beskriver de olika XML-meddelandena och deras strukturer som CTC stöder [6].

3 Metod

3.1 Procedur

Informationsinhämtning sker främst genom tillhandahållen dokumentation. Information om CTC och OCS finns i diverse intern dokumentation och wikis, som kan bidra till förståelse. Handledare kommer även bidra med kunskap om systemen. För programmeringsspråken C# och Java finns källor på Internet för bättre förståelse av språken och ramverken [7], [8], [9].

Kommunikationen mellan CTC och OCS skall följa det existerande protokollet specifierat i FFFIS TCC-OCS. CTC är tänkt att agera som TCC-änden för prototypen, vilket leder till att OCS inte behöver ändra sitt kommunikationslager. I CTC skapas en ny modul som skall hantera kommunikation mot OCS och översätta inkommande och utgående meddelanden på lämpligt sätt som logiken för CTC skall kunna behandla och visa upp. Översättning av meddelanden skall följa FFFIS CTC, som specifierar meddelanden för CTC.

En C#-variant av TCC-OCS protokollet kan används som referens för att påskynda implementationen i Java. Viss kod från C#-implementationen tillåts återanvändas i detta arbete. Implementationen kommer kräva skilda lösningar där direkt översättning inte går. Alternativa lösningar och strukturer kommer att tillämpas för att uppnå goda objektorienterade principer och mönster.

Kod bör utvecklas på sådant sätt att användning och underhåll av koden förenklas. Detta innebär till exempel att föredra vanliga förekommande lösningar, till skillnad från oklara, långsökta lösningar. Problem bör även lösas genom att använda existerande gränssnitt för att inte återskapa det som redan finns. Enkla lösningar kan användas som första utgångspunkter och ge ökad förståelse för problem och eventuell vidareutveckling.

Råd om kodningsprinciper från mer erfarna Java-utvecklare kommer tillämpas, bl.a. via *Effective Java*, som innehåller en samling goda tekniker för att skriva effektivare Java-kod [10]. Många punkter är användbara för att förstå hur kod bör struktureras, göra koden mer lätt förståelig och följa principer som inkapsling.

3.2 Verktyg

Eclipse kommer att användas som utvecklingsmiljö, vilket tidigare använts för utveckling av CTC. Kod kommer att skrivas i programmeringsspråket Java. Bombardiers interna program används, inklusive OCS-relaterade simulatorer och CTC. Git/Gerrit används för versionshantering. Utvecklingsplattformen kommer köra 64-bitars Windows. Körning av OCS-programvara kräver Linux, för vilket Oracle VM VirtualBox ska användas för att virtuellt skapa en Linux/Ubuntu-miljö.

4 Genomförande

Arbetet är en vidareutveckling av mjukvara utvecklad och designad av ett flertal tidigare. Till en början krävdes inhämtande av omfattande information avseende systemets funktion. Tillgång till OCS- och CTC-projekteten erhölls via Git/Gerrit. Möten om projektets bakgrund och mål ordnades för att komma in i sammanhanget. CTC:s uppbyggnad förklarades kort, varefter arbetet började med att implementera protokollet FFFIS TCC-OCS.

4.1 Implementation av kommunikationsmodul

Kommunikationsmodulen, CTC-OCS Communicator (COC), implementeras i detta arbete baserat främst på specifikationer i FFFIS TCC-OCS. Modulen är menad att integreras i kommunikationslagret för CTC för att möjliggöra direkt kommunikation med OCS.

4.1.1 Skillnader mellan C# och Java

Tillgång till en C#-implementation av protokollet användes som referens för att skapa Java-implementationen. Vissa delar översattes från C# till Java, men trots liknande syntax i båda språken finns många olikheter i tillhörande ramverk.

Unsigned

Konceptet unsigned stöds inte av Javas standardtyper. C#-unsigned översattes därför till typer av ekvivalenta storlekar i Java. Detta är möjligt eftersom ingen aritmetik behövde utföras på typerna, utan bara lagras innan sändning och/eller avläsning. Figur 4.1 visar vad C#-typer översattes till i Java.

```
C#                                  Java
// Instance variables               // Instance variables
private ushort x;                   private Short x;
private uint y;                     private Integer y;
private ulong z;                    private Long z;
```

Figur 4.1. *C# unsigned typers motsvarande typer i Java-implementationen.*

I FFFIS TCC-OCS specifieras till exempel att en 16-bitarsvariabel skall kunna variera mellan [0, 65535], d.v.s. [0, 2^{16}-1]. Javas standardtyper är alla signed i tvåkomplementsform, vilket kan misstolkas då en short av 16 bitar, antar [-2^{15}, 2^{15}-1] = [-32768, 32767]. Hänsyn till unsigned behöver visas vid avläsning av talen i Java. Negativa tal finns inte enligt protokollet.

Tanken att skapa egendefinierade typer som skulle representera 16, 32, och 64 bitar genom att hålla bytefält övervägdes, men i Java 8 finns funktionalitet för de primitiva typernas omslagna motsvarigheter (*boxed primitives*), som förenklar behandling av unsigned, som `Short.toUnsignedInt(aShort)`, och `Short.BYTES`.

Följande kodexempel, i figur 4.2, visar ett tankefel som kan ske utan hänsyn till Javas tvåkomplementform. En metod skall inkrementera en tänkt 32-bitarsvariabel, vilket motsvaras av en int, för att sedan slå om till 0 igen när det nått det största värdet 32-bitar kan nå, d.v.s. hexadecimalt: FF FF FF FF (decimalt: 4 294 967 295, binärt: 11111111 11111111 11111111 11111111). Negativa tal exkluderas här, eftersom syftet är att simulera unsigned. I vänstra metoden returneras 0 när ett godtyckligt (positivt) tal skickas som inparameter, eftersom 0xFFFFFFFF motsvaras av -1 i tvåkomplementsform. Detta problem löses med Integers `toUnsignedLong()`-metod för att kunna läsa av talet som tänkt.

```
 // Supposed to increment any
 // 32-bit integer and wrap
 // around to 0, but returns 0
 // for any positive int i.
private int increment(int i) {
    if (i > 0xFFFFFFFF) {
        i=0;
    } else {
        i++;
    }
    return i;
}
```

```
// Supposed to increment any
// 32-bit integer and wrap
// around to 0, and does so.
private int increment(int i) {
    if (i >
Integer.toUnsignedLong(0xFFFFFFFF)) {
        i=0;
    } else {
        i++;
    }
    return i;
}
```

Figur 4.2. *Vänster metod inkrementerar inte som tänkt. Höger metod läser av det hexadecimala talet och slår om till 0 igen som menat.*

Enumeration

Enumeration, eller enum, i C# är mycket lik den i C med automatiska värden. Denna funktionalitet finns dock inte i Javas enum. Översättning till Java enums fick istället egendeklarerade variabler och konstruktorer för att associeras med önskade värden. Java enums metoden `ordinal()` ger index i enumen, men börjar från 0, vilket i vissa fall inte är det som behövs. Figur 4.3 visar hur en C#-enum implementeras i Java.

C#

```
private enum Packet {
    A = 1,
    B,
    C
};
```

Java

```
private enum Packet {
    A(1),
    B(2),
    C(3);
    private int id;
    Packet(int id) {
        this.id = id;
    }
}
```

Figur 4.3. *C# enum-motsvarighet i Java-implementationen.*

Timer

Timers behövs för att periodvis utföra uppgifter som att skicka information. Timers utnyttjar TimerTasks vilka implementeras som anonyma klasser i COC för att kapsla in dem i relevanta klasser. Figur 4.4 visar ett exempel på användning av Timer med en TimerTask.

```
import java.util.Timer;
import java.util.TimerTask;

public class APeriodicTask {
    private Timer timer;

    // Starts a task which runs every period milliseconds starting after
    // delay milliseconds. The task is defined in the method run().
    APeriodicTask(int delay, final int period) {
        timer = new Timer();
        timer.scheduleAtFixedRate(new TimerTask() {
            @Override
            public void run() {
                // Do stuff
                System.out.println(period/1000 + " seconds passed.");
            };
        }, delay, period);
    }
}
```

Figur 4.4. *En Java-klass vilken periodvis exekverar en TimerTask.*

4.1.2 Förändringar i implementation

Utöver direkta översättningar från C#-referensen implementeras vissa saker annorlunda i kommunikationsmodulen av diverse anledningar, bl.a. bättre inkapsling och en tydligare Model-Controller-orienterad struktur. Nedan behandlas några av dessa skillnader.

Sockets

Sockets skiljer sig åt i ramverken för Java och C#. Istället för de synkrona sockets som används i C#-implementationen, används Javas server och client sockets. Client sockets används i en klass, kallad Connection, vilken representerar en hel anslutning med all relaterad funktionalitet för att skicka och ta emot meddelanden. Klassen ansvarar även för pålitlighetsfunktionalitet som ser till att data inte ändras på vägen, till exempel i form av signalstörningar. Server socket används för en lyssnartråd i en klass som kallas SocketManager. Denna klass väntar på inkommande förfrågningar för att ansluta sig.

Figur 4.5 visar ett minimalt server-program som tar emot nya klienter. Programmet startar en server socket på port 55 555 och väntar på att klienter ansluter sig. Vid klientanslutning sparas nyligen skapad Socket i en lista som kan användas för att få ut strömmar, IP-address, port, etc.

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;

public class Server {
    private List<Socket> socketList;
    private ServerSocket serverSocket;
    private Thread thread;

    // A Server listens to a port number for new connections
    Server(int port) throws IOException {
        socketList = new ArrayList<>();
        this.serverSocket = new ServerSocket(port);
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                while(!thread.isInterrupted()) {
                     try {
                         Socket s = serverSocket.accept();
                         System.out.println("Client connected.");
                         socketList.add(s);
                     } catch (IOException e) {
                         // Error handling
                     }
                }
            }
        });
        this.thread = t;
    }
    public void startListening() {
        this.thread.start();
    }
    public static void main(String [] args) {
        try {
            new Server(55555).startListening();
        } catch (IOException e) {
            // Error handling
        }
    }
}
```

Figur 4.5. *En minimal server-klass i Java.*

Skapande av paket

Meddelanden i C#-implementationen skapar paket (*packets*) direkt, baserat på identifikationsnummer, vilket betyder att varje paket håller reda på varenda existerande nummer. I Java refaktoriseras detta istället till en Factory-klass kallad *PacketFactory*, enligt designmönstret Factory, som fick hålla reda på existerande identifikationsnummer och skapa paket framöver.

Felhantering

Kontroll av data anses vara bra för att utesluta felaktigheter i efterföljande steg. Flera kontroller av inparametrar lades till för att försäkra att inmatade värden var inom tillåtna intervall. För att underlätta felsökning vid manuell testning skrivs ej tillåtna värden ut på standard strömmen med `System.out.println()`. Efter testning ersätts utskrivningen på standard strömmen ut till en existerande logg i CTC-projektet, som ofta används i CTC-projektet. Med denna logg lagras fel i textfiler som senare kan analyseras. I C#-implementationen finns även events för att trigga olika händelser för fel, varningar, info m.m. som inte implementerades i Java-implementationen.

Konstruktorer

Det är möjligt att ha överlagrade konstruktorer i klasser, d.v.s. flera konstruktorer som har olika signaturer. För att göra sådana klasser mer lättanvända övervägdes designmönstret Builder som ett mellansteg innan konstruktorn anropas. Buildern skulle bli en egen klass som tar inparametrarna för konstruktorn och inte förrän man explicit kallar på builders delegat för att anropa konstruktorn skulle instansen som man initierat skapas och returneras av buildern.

Detta blev onödigt komplicerat och konstruktorerna förenklades med mer konsekvent användning när det fanns många parametrar. De klasser som även krävde många parametrar försöktes hållas omuterbara. När detta inte gick, fick de minimalt med mutators/setters för att minska buggar och feltillstånd som lätt kan uppstå.

Bitmanipulering

För att omvandla bytefält till objektrepresentationer, och vice versa, behövs bitmanipulering och ordning på bitar som skickas. I C#-implementationen används *bit shifting* för att placera specifika bytes på korrekt plats för att lagra dem. I Java-implementationen används ByteBuffers där det gick, för att göra koden mer lättläslig. C#-ramverket har motsvarande ByteBuffer-funktionalitet med MemoryStream. För CRC-beräkning behölls algoritmen i C#, direkt översatt till Java.

Figur 4.6 visar motsvarande metoder med de nämnda sätten för att omvandla typer tillbaka till bytefält. Med större typer (som long) och fler fält, blir koden betydligt kortare med ByteBuffer.

```
// C# bit shifting
private const int HeaderLength = sizeof(ushort) + sizeof(uint);
public byte[] ToByteArray(ushort a16bitField, uint a32bitField) {
     byte[] byteArray = new byte[HeaderLength];
     int counter = 0;
     byteArray[counter++] = (byte)(a16bitField >> 8);
     byteArray[counter++] = (byte)(a16bitField >> 0);
     byteArray[counter++] = (byte)(a32bitField >> 24);
     byteArray[counter++] = (byte)(a32bitField >> 16);
     byteArray[counter++] = (byte)(a32bitField >> 8);
     byteArray[counter++] = (byte)(a32bitField >> 0);
     return byteArray;
}

// Java bit shifting
private static final int HEADER_LENGTH = Short.BYTES + Integer.BYTES;
public byte[] toByteArray(Short a16bitField, Integer a32bitField) {
     byte[] byteArray = new byte[HEADER_LENGTH];
     int counter = 0;
     byteArray[counter++] = (byte)(a16bitField >> 8);
     byteArray[counter++] = (byte)(a16bitField >> 0);
     byteArray[counter++] = (byte)(a32bitField >> 24);
     byteArray[counter++] = (byte)(a32bitField >> 16);
     byteArray[counter++] = (byte)(a32bitField >> 8);
     byteArray[counter++] = (byte)(a32bitField >> 0);
     return byteArray;
}

// Java ByteBuffer
private static final int HEADER_LENGTH = Short.BYTES + Integer.BYTES;
public byte[] toByteArray(Short a16bitField, Integer a32bitField) {
     ByteBuffer byteBuffer = ByteBuffer.allocate(HEADER_LENGTH);
     byteBuffer.putShort(a16bitField);
     byteBuffer.putInt(a32bitField);
     return byteBuffer.array();
}
```

Figur 4.6. *Tre metoder som returnerar ett bytefält från en header av två fält.*

De tre metoderna i figur 4.6 är ekvivalenta för scenariot då man har en header bestående av ett 16-bitarsfält följt av ett 32-bitarsfält. Metoderna tar in värden för de två fälten och returnerar bytefält-representation.

4.1.3 Fortsatt utveckling enligt protokoll

Skillnader i C#- och Java-implementationerna blev så många att det blev svårare att fortsätta implementera med liknande tankesätt. Istället användes nätverksprotokoll (FFFIS TCC-OCS) som enda referens för vidare utveckling i Java, vilket lett till fler avvikelser mellan de två implementationerna.

4.2 Testning av kommunikationsmodul

4.2.1 OCS-simulator

För att testa kommunikationen från OCS till COC i en mer verklighetstrogen miljö används en simulation av OCS bestående av två fristående delar, en CCU-del samt en OC-simulator, med ett separat grafiskt gränssnitt (OCSimulatorGUI). OC-simulatorn simulerar flera OC som är anslutna till olika objekt som finns längs spår. OCS-simulatorn kan skicka och ta emot meddelanden från kommunikationsmodulen och visa konsekvenserna av ordrar, vilket blev det mest praktiska sättet att testa modulens funktionalitet under tidig utvecklingsfas.

4.2.2 Testfall

Vissa testfall skrevs i testklasser för att försäkra att implementerade FFFIS TCC-OCS-meddelanden och procedurer uppförde sig korrekt. Då projektet syftade på att ta fram en fungerande prototyp prioriterades inte testning för olika typer av täckning (*code coverage*). Kärnfunktionalitet testades mer än specialiserade saker, såsom typer av paket som inte än hade användning.

4.3 Integrering av COC med CTC

Kommunikationsmodulen skapades till en början väldigt fristående, utan att på något sätt integreras med CTC. Inte förrän modulen var nästintill färdig, enligt specifikationer i FFFIS TCC-OCS, undersöktes olika möjligheter att integrera den i CTC.

Avsikten från projektets början var att helt ersätta kommunikationslagret i CTC. När kommunikationslagret togs bort visade sig mycket av CTC-programmet bero på dess kod vilket gjorde åtgärden mindre passande. Istället analyserades projektet för att hitta lämpliga delar som kunde använda COCs gränssnitt. CTC-kommunikationens ändpunkt modifierades för att använda sig av COC:s gränssnitt och skapade därmed koppling mellan COC och CTC.

De existerande nätverksfunktionerna stängdes av i CTC genom att modifiera dess tillståndsmaskin. Detta stoppade programmet från att försöka koppla upp till en TCC, samt stängde av viss funktionalitet som berodde på uppkopplingen till TCC.

4.3.1 Utgående trafik

Den utgående datatrafiken löstes genom att i existerande klass där XML-meddelanden skickades ut (till TCC), översätta FFFIS CTC XML-formatet till dataformatet enligt FFFIS TCC-OCS. För detta användes Java DOM parser för att ta ut relevanta noder från XML-meddelandet och beroende på innehåll, bygga FFFIS TCC-OCS paket och meddelanden enligt de i applikationsskiktet.

Därefter hittas namn på mottagare för att kunna skicka meddelandet via COC-modulen över nätverk till OCS.

4.3.2 Inkommande trafik

XML-meddelandens struktur identifierades med hjälp av FFFIS CTC-protokollet, samt genom att analysera den kod som läser av data i inkommande XML-meddelanden. Med denna information kunde existerande gränssnitt i CTC-projektet för behandling av XML användas för att implementera skapandet av XML-meddelanden baserat på inkommande data.

För inkommande trafik skulle statusuppdateringar visas upp på användargränssnittet för att ge operatören feedback. Då statusuppdateringar är de enda inkommande meddelandena som behöver behandlas av CTC:s logik, används FFFIS CTC:s indikationsmeddelanden (*indication messages)*.

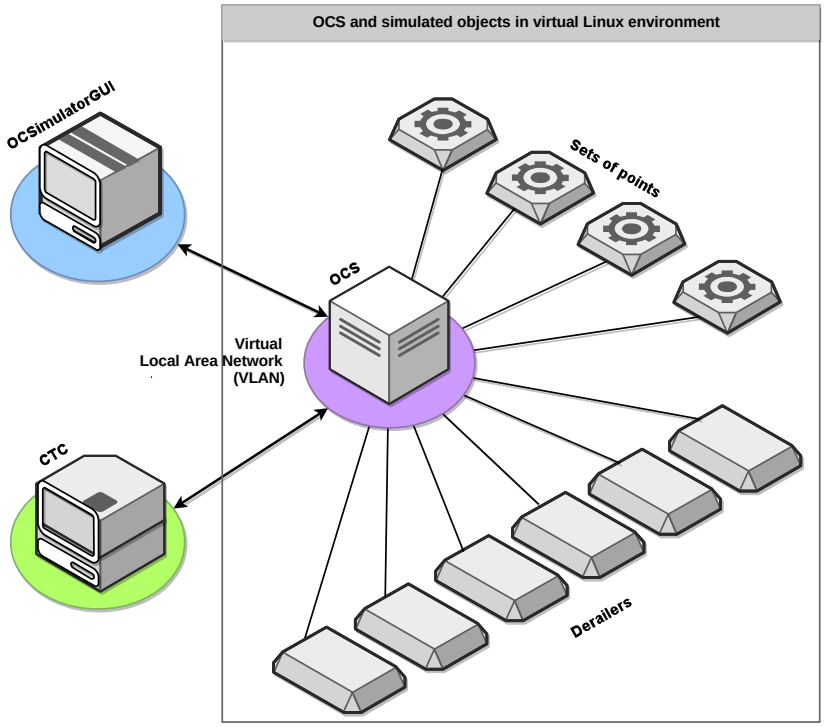
4.4 Testning av modifierad CTC

CTC-OCS-kommunikationen testades genom att använda CTC med simulationen av OCS. För att försäkra att CTC påverkade korrekt objekt enligt valt kommando, användes verktyget OCSimulatorGUI för att kunna se status på alla uppkopplade OC:s. Vyn i OCSimulatorGUI kunde jämföras med det som visades på CTC. CTC förväntades kunna styra objekt genom att en operatör klickar på lämpliga kommandon i GUI:t. Dessutom förväntades CTC indikera statusuppdateringar för anläggningens objekt periodvis, samt på tillståndsförändringar.

Följande steg utförs för att sätta igång testmiljön:

1. Starta OC Simulator. Sätter i gång simulation av Object Controllers.
2. Starta OCSimulatorGUI och anslut till IP-adress och specifik port på maskinen som kör OC-simulatorn. GUI:t visar översikt av alla OC.
3. Starta CTC, som nu väntar på klient (CCU) att ansluta.
4. Starta CCU. CCU ansluter sig till IP-adress inställd i en konfigurationsfil på en specifik port. IP-adressen skall vara CTC-maskinens.

Efter dessa steg bör CCU vara ansluten till CTC inom några sekunder. När CCU anslutit till trygghetsskiktet och därefter applikationsskiktet, bör den periodvis skicka statusuppdateringar, vilket skall synas på CTC genom att symboler under objekten på användargränssnittet försvinner.



Figur 4.7. *Datorkonfiguration vid testning.*

Anläggningen med samtliga objekt och områden specifieras i en XML-konfigurationsfil. För testcenariot användes en konfigurationsfil med tio OC:s under en OCS. De tio OC:s ansvarade var för sig ett objekt. Objekten i testkonfigurationen hade sex växlar och fyra spårspärrar, utöver spår och logikregler.

Vid första inläsning av ny konfigurationsfil finns ingen layout för samtliga objekt. CTC lägger ut de objekt den kan, men för att kunna flytta de grafiska symbolerna kan ett redigeringsläge aktiveras, i vilket de grafiska objekten omplaceras för att få önskad layout.

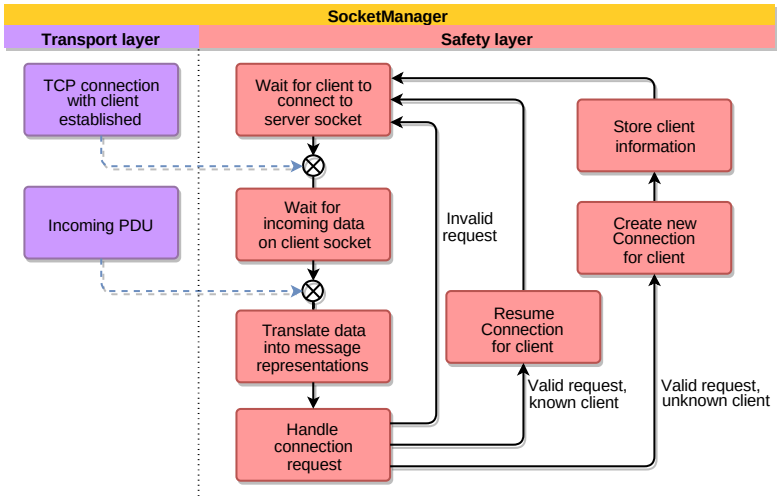
5 Resultat

5.1 Kommunikationsmodulen

Kommunikationsmodulen, COC, är en Java-implementation av FFFIS TCC-OCS. All funktionalitet för att upprätthålla anslutningar är implementerad. Implementationen har hårdkodat vissa värden (som versionsnummer och namn) för att lyckas med initial anslutningsprocedur. Modulen har testats direkt mot CCU:n i OCS med simulerade OCs, vilket visat sig fungera. Modulen kan upprätthålla anslutningar, ta emot meddelanden, och processa meddelanden till representationer.

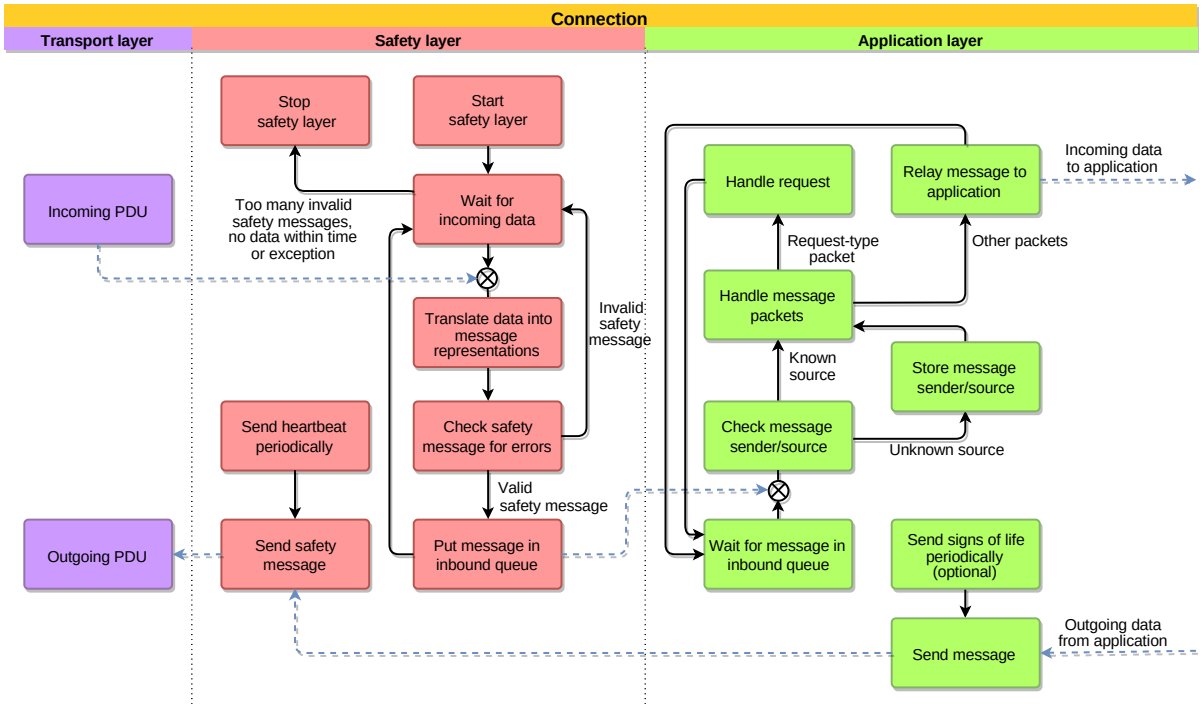
Kommunikationsmodulen arbetar på flera skikt. För transportskiktet används sockets från Java.net API:et. Trygghetsskiktet är implementerat efter specifikationerna i FFFIS TCC-OCS för behandling av meddelanden. Detta skikt ser till att meddelanden håller korrekt form och giltiga värden som kontrolleras för att godkänna användning. Logiken för trygghetsskiktet är fördelad mellan klasserna SocketManager och Connection. FFFIS TCC-OCS har även krav i applikationsskiktet för att upprätthålla anslutningar till andra enheter på applikationsskiktet vilket Connection ansvarar för. Översättningen av OCS-meddelanden till CTC-meddelanden har däremot placerats utanför kommunikationsmodulen.

Inkommande förfrågningar mottas i klassen SocketManager, där de bedöms för vidare kommunikation. Vid giltiga klientanslutningar skapas nya Connection-instanser för varje klient. Figur 5.1 förklarar flödet för klassen.



Figur 5.1. *Översiktligt flöde i klassen SocketManager, som hanterar nya klienter.*

Figur 5.2 förklarar logikflödet i klassen som representerar en anslutning. Klassen Connection ansvarar för allt på trygghetsskiktet, förutom den initiala anslutningsproceduren. På applikationsskiktet ansvarar Connection för den initiala anslutningsproceduren, behandling av särskilda typer av meddelanden och förmedling till applikation för resterande meddelande typer. Notera att en anslutningsprocedur sker på samtliga tre skikt.

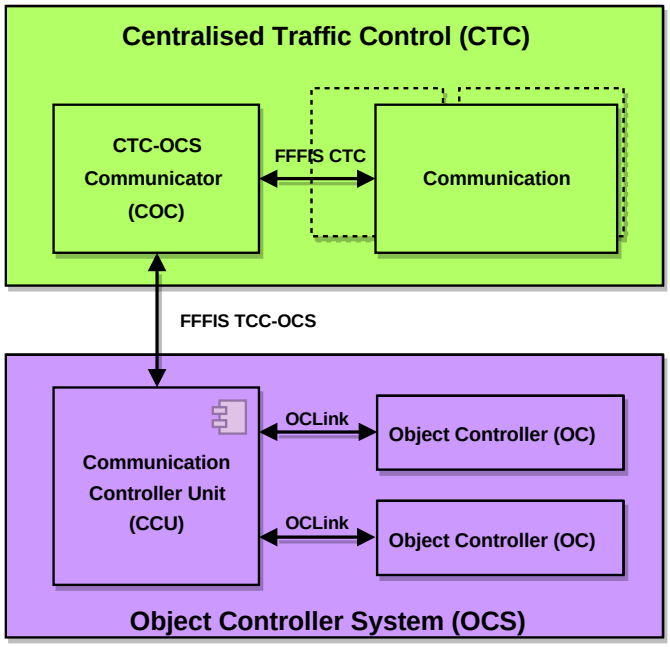


Figur 5.2. *Översiktligt flöde i klassen Connection, som innehåller kärnfunktionaliteten av COC.*

5.2 Kommunikationen mellan delsystem

För att integrera kommunikationsmodulen i CTC anpassades den existerande kommunikationen. Ändringar gjordes för att omdirigera all in- och utdata via COC. Anslutningsförsök till TCC i CTC stängdes av, för att istället låta COC ta emot anslutningar d.v.s. agera server. Kommunikationsmodulen fick ansvaret för att ta emot och uppehålla anslutningar.

Översättning av FFFIS TCC-OCS-meddelanden till FFFIS CTC-meddelanden och vice versa blev nödvändig för att få ihop informationsutbyte. Detta sker i ändpunkten av kommunikationslagret i CTC, som tidigare ansvarade för kontakt med TCC. De objekt som stöds är begränsade till ett fåtal typer, men kan enkelt utökas med fler.



Figur 5.3. *Förbindelser i och mellan CTC och OCS med integrerad kommunikationsmodul.*

6 Slutsats och diskussion

6.1 Användning av två nätverksprotokoll

Borttagning av CTC:s kommunikationslager föreslogs för arbetet, men fullföljdes inte eftersom det krävde högre förståelse av CTC:s struktur. Detta visade sig tidskrävande på grund av dess abstrakta uppbyggnad och brist på dokumentation. Samtidigt skulle även en kommunikationsmodul (COC) utvecklas, vilket hade högre prioritet under arbetets gång.

Skulle CTC:s kommunikationslager helt tagits bort, skulle som följd kommunikationen i prototypen enbart använda sig av FFFIS TCC-OCS-specifikationen. COC skulle då behöva använda sig av CTC:s logiklagers gränssnitt, istället för kommunikationslagrets som i prototypen. Fördelen med detta skulle innebära en mer anpassad och effektiv lösning för kommunikation med OCS. Nackdelen är att majoriteten av CTC:s funktionalitet utesluts.

I prototypen behålls kommunikationslagret i CTC vilket nu kräver två protokoll (FFFIS TCC-OCS och FFFIS CTC) för att fullborda dataflödet. FFFIS CTC-specifikationen innehåller kommunikation för mer än bara objekt längs järnvägsspår. Detta innebär att stora delar av protokollet (och som följd CTC-programmets funktionalitet) för tillfället inte har någon användning utan TCC.

6.2 Vidareutveckling

De flesta CTC-meddelanden översätts inte ännu till motsvarande OCS-meddelanden och paket. Åt motsatt håll översätts inte heller alla OCS-meddelanden till CTC-meddelanden. Dessa översättningar behöver implementeras beroende på vilken funktionalitet systemet skall ha med.

Kommunikationen mellan CTC och OCS har enbart testats med modifierad OCS med styrlogik implementerad i CCU:n. Inga värden modifierades i OCS för att anpassas till kommunikationen mot CTC, vilket krävt att CTC använder existerande värden menade för kommunikation med TCC. I COC innebär detta att en del fält som används i headers hårdkodats då det inte finns konkreta värden att tillgå. För en mer robust lösning behöver nya, lämpliga värden bestämmas med OCS-änden, vilket även skulle tillåta en mer allmän lösning för CTC-änden, till skillnad från den tvingade lösningen i denna prototyp.

Trygghetsskiktet är beroende av korrekta tider för att kunna uppehålla anslutningar. För COC har tidsberäkningen inte visat sig få samma resultat som från OCS-sidan. För att synkronisera klockorna krävs troligen implementering av TCP/IP-stackens Network Time Protocol (NTP).

Det finns ytterligare sätt att förbättra kodstrukturen. Följande är några punkter som kan åtgärdas:

- Använda concurrency API:ets Executors för bättre trådhantering.
- Använda Observer-mönstret för bättre inkapsling och resursanvändning.
- Hantering av fler exceptions.
- Förbättrad, mer konsekvent användning av Factory-mönstret för fler modell-objekt.
- Flytta viss logik till lämpligare klasser, d.v.s. refaktorisera.

6.3 Hållbar utveckling

Signalsystemet detta arbete skapar har potential att bidra till ökad automatisering av kontroll av järnvägsspår. Ökad automatisering leder till minskad manuellt arbete som istället kan beordras utifrån det fysiska området och utföras av datorstyrna maskiner. Säkerheten ökar för arbetare, till exempel genom att färre växlar behöver slås om för hand i riskfyllda spårområden.

6.4 Framsteg

Verifiering av kommunikationen har genomförts genom att skicka instruktioner från GUI:t i CTC, vilka mottas i en uppkopplad CCU, vilket i sin tur kommunicerar med simulerade Object Controllers och visats styra objekten som tänkt. Svar och statusuppdateringar mottas i COC och förmedlas genom logiken i CTC som uppdaterar det grafiska gränssnittet.

Med denna funktionalitet har målet för detta arbetes avgränsningar uppfyllts. Målet för arbetet var att skapa kommunikation mellan Centralised Traffic Control (CTC) och Object Controller System (OCS). Med detta skulle det vara möjligt att styra objekt längs järnvägsspår utan mellanliggande TCC. Detta har möjliggjorts med kommunikationsmodulen COC som kommunicerar med OCS, samt modifiering av CTC för att använda sig av COC.

Kommunikationen har testats mot det parallella arbetets version av CCU6 med styrlogik. Där har logiken visat sig kunna begränsa ordrar för att till exempel hindra ändring av växlar som kan leda till kollision. Dessutom har logiken visat sig automatiskt utföra ordrar som konsekvens av en annan, som att automatiskt växla en spårväxel längre fram vid manuell växling av en spårväxel. Med denna styrlogik har säkerhetsfunktionalitet som TCC ansvarade för ersatts.

Med fungerande CTC-OCS-kommunikation och styrlogik i OCS har målet för även hela projektet uppfyllts. En prototyp av ett signalsystem, utan TCC, som tillåter lokal kontroll av objekt längs järnvägsspår har utvecklats.

Referenser

[1] *Bombardier in Sweden - Gothenburg, Sweden,* Bombardier.
URL: http://se.bombardier.com/se/site_details_gothenburg.htm
[Hämtad 2016-05-31]

[2] *INTERFLO 150 – KIRUNA MINE Optimising the movement of iron ore*, Bombardier Transportation, 2013.
URL: http://www.infrasig.net/media/86275/08393_interflo_150_signalling_cs_lores.pdf
[Hämtad 2016-06-09]

[3] L. Leiulfsrud, J. Sjöberg, *Styrprogram med konfigurerbar logik för spårtrafik*, Chalmers tekniska högskola, 2016.

[4] K. Hedberg, F. Elestedt, S*afety-critical Communication Controllers for Railway Signalling in Public Networks*, Chalmers tekniska högskola, 2008.
URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.154.5123&rep=rep1&type=pdf [Hämtad 2016-06-06]

[5] F. Elestedt, *FFFIS TCC-OCS ERTMS-R*, Bombardier Transportation, 2010.
ID: 3NSS006735D0065.

[6] J. Reimers, *FFFIS CTC Interflo 150*, Bombardier Transportation, 2013.
ID: 3NSS010609D0112.

[7] Gosling et al., *The Java® language specification*, Oracle, 2015.
URL: https://docs.oracle.com/javase/specs/jls/se8/html/index.html

[8] *Java™ Platform, Standard Edition 8 API Specification*, Oracle, 2016.
URL: https://docs.oracle.com/javase/8/docs/api/

[9] *.NET Framework Class Library,* Microsoft, 2016.
URL: https://msdn.microsoft.com/en-us/library/gg145045(v=vs.110).aspx

[10] J. Bloch, *Effective Java 2nd Edition*, Addison-Wesley, 2008.
ISBN-13: 978-0-321-35668-0, ISBN-10: 0-321-35668-3.