CHALMERS

OPC UA Stack och SDK i Delphi riktat mot färdigt miljöredovisningssystem

Examensarbete i Data- och Informationsteknik

WILLIAM BJÖRKLUND
VICTOR DAHLBERG

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2017

EXAMENSARBETE

OPC UA Stack och SDK i Delphi riktat mot färdigt miljöredovisningssystem

WILLIAM BJÖRKLUND
VICTOR DAHLBERG

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET

Göteborg, Sverige 2017

OPC UA Stack och SDK i Delphi riktat mot färdigt miljöredovisningssystem
WILLIAM BJÖRKLUND
VICTOR DAHLBERG

© WILLIAM BJÖRKLUND, VICTOR DAHLBERG, 2017

Examinator: Peter Lundin

Institutionen för Data- och Informationsteknik
Chalmers tekniska högskola
SE-412 96 Göteborg
Sverige
Telefon: +46 (0)31-772 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.

Instutionen för Data- och Informationsteknik
Göteborg, Sverige 2017

Sammanfattning

Projektet behandlar användningsområden för kommunikationsstandarden OPC UA, samt implementeringen av en kommunikationsstack och klient enligt denna standard i programmeringsspråket Delphi. OPC UA är en standard från *OPC Foundation* som är modernare än den tidigare utgivna OPC Classic som är i brett industriellt användande i dagsläget. Denna nyare standard ger större framtida flexibilitet och kompatibilitet gentemot varierande operativsystem och framtida säkerhetsalgoritmer. Klienten som är utvecklad inom ramen för detta projekt kan upptäcka och kommunicera med servrar lokalt eller över internet via meddelandestrukturen för *opc.tcp*. Kontakt- och strömförlust hanteras och klienten är designad för att hantera läsning av konfigurationsfiler, varpå möjligheten att snabbt prenumerera på förutbestämda värden från en server erbjuds.

Nyckelord: OPC, OPC UA, Delphi

Abstract

This project concerns the necessity of the *OPC UA* standard for communication as well as the implementation of a OPC UA communication stack and client in the Delphi programming language. *OPC UA* is the newer standard for communication developed by the *OPC Foundation* to ensure greater future flexibility and compatibility with different operating systems and security algorithms. The client application may discover and communicate with different servers locally or via the internet by means of messages utilising the standard's predefined message structures for *opc.tcp*. Loss of connectivity and power by the client is handled and options for prepared configurations are in place to ensure the viability of quick subscription to data values on the desired server.

Förord

Ett stort tack går ut till företaget Entric för denna möjlighet, den trevliga atmosfären samt hjälpsamhet igenom hela projektet. Fredrik Jonson ges en särskild eloge för sin hand i struktureringen av arbetet samt kontinuerlig kvalitetskontroll och hjälp med problemområden. Stor uppskattning även till Roger Johansson för hans roll som akademisk handledare.

Terminologi

Adressrymd Alla noder en OPC UA-Server innehåller.
OPC Object Linking and Embedding for Process Control.
OPC UA OPC Unified Architecture.
SDK Ett Software Development Kit är ett paket av utvecklingsverktyg som simplifierar utvecklingen av en mjukvara för ett visst syfte.
PLC Programmerbar logisk kontrollenhet. En industriell dator som kontrollerar en eller fler hårdvaruenheter.
Socket En ändpunkt för kommunikation bestående av en ip - adress och port.
opc.tcp Ett transportprotokoll för OPC UA designat specifikt av OPC Foundation.
Dödband Ett värdesintervall som bestämmer om ett värde är önskat eller ej.
Bug Fel i källkoden vilket gör att den inte agerar enligt avsikt.
TBytes Datatyp som beskriver ett fält av bytes.
Int32 Datatyp som beskriver ett 32-bitars heltal.

Innehåll

Sammanfattning **i**

Abstract **ii**

Förord **iii**

Terminologi **v**

Innehåll **vii**

1 Inledning **1**
1.1 Bakgrund 1
1.2 Syfte 1
1.3 Precisering av Frågeställningen 1
1.3.1 Kravspecifikation 2
1.4 Avgränsningar 2

2 Teknisk Bakgrund **3**
2.1 Delphi 3
2.2 OPC Classic 3
2.3 OPC UA 4
2.4 OPC UA Kommunikation 5
2.4.1 OPC UA Meddelandestruktur 5
2.4.2 Säkerhetskanal *(Secure Channel)* 6
2.4.3 Tjänst för att upptäcka OPC-Servrar *(Discovery Service)* 6
2.4.4 Sessionstjänsen *(Session Service)* 7

3 Metodbeskrivning **8**
3.1 Versionshantering 8
3.2 Paketsniffning 9
3.3 Att jämföra bytes med hjälp av paketsniffning 10
3.3.1 Definitionen av *Null*-värden 10
3.3.2 Strukturparametrar 10
3.3.3 Jämförelse av bytes 11
3.4 Felsökning 11
3.4.1 Praktiska begränsningar 11
3.5 Minnesläckor 12

4 Konstruktion **14**
4.1 Interaktion med *socketen* 14
4.1.1 Användning av *callback*-procedurer 14
4.1.2 Bufferttråden 15
4.1.3 Designfel i bufferttrådens huvudloop 16
4.1.4 Delegering av exekvering av *callback*-procedurer 17

4.2 Prenumeration på värden 17
4.2.1 Prenumerationstråden 18
4.2.2 Överskrivning av data i notifikationsförfrågningar 19
4.2.3 Samling av prenumerationstrådar 20
4.3 Bläddra i OPC-serverns *addressrymd* 21
4.3.1 Nod-ID från sökväg med hjälp av *Folder*-objektet 22
4.4 Hantering av kontaktförlust eller anslutningsfel 23
4.5 Agerande vid kontakt- och strömförlust 24
4.5.1 Kontaktförlust 25
4.5.2 Strömförlust 25
4.5.3 Agerande 25

5 Resultat **27**
5.1 Klienten 27
5.1.1 Startfönstret 27
5.1.2 Server- och ändpunkts-val 28
5.1.3 Huvudfönster 28
5.1.4 Anslutningsfönster 30
5.1.5 Fönster för att lägga till enskilda datakällor i en prenumeration 30
5.1.6 Prenumerationsfönster 31
5.1.7 Fönster menat för produktionskörning 31
5.2 Återkoppling till kravspecifikationen 32

6 Diskussion **34**
6.1 Hållbar utveckling 34

Referenser **35**

1 Inledning

OPC UA är inte en ny standard. Inom industrin har den dock fortfarande inte tagit över för den föregående arkitekturen. Anledningar till detta kan ses inom både utvecklingskostnad samt oro för att tillförlitligheten inte blir densamma som ett existerande, beprövat system. Båda nya och etablerade företag har dock tagit steget och fler lär troligen göra så under kommande år. Med detta i åtanke detaljeras *Entrics* situation och intresse för denna nya standard.

1.1 Bakgrund

Inom Sverige och EU som helhet existerar i dagsläget omfattande lagstiftning angående industriellt utsläpp av luftburna gaser[1, 2, 3]. Emissionsnivåer för olika gaser måste kontinuerligt mätas då överskridning av den tillåtna gränsen är grund för totalstopp av verksamheten samt potentiell laglig påföljd. Företag har således ett behov av konstant övervakning av sina utsläpp samt tydlig notifiering då kvoterna för dessa når relevanta tröskelnivåer.

Entric är ett svenskt företag som idag främst arbetar med support och uppgradering av deras tidigare utvecklade miljöredovisningssystem, *MRS*. *MRS* samlar upp mätdata från relevanta rökgaskanaler och redovisar den för att kunna påvisa att den svenska och europeiska lagstiftningen följs.

MRS har stöd för flera olika kommunikationsprotokoll och kommunikationsstandarder för att kunna anpassa sig efter kundens behov och eventuella krav. En sådan kommunikationsstandard som *MRS* stödjer är OPC, även benämnt som *OPC Classic*. OPC-specifikationen är baserad på Microsofts *COM/DCOM*-teknologi och är därför Windows-beroende. Nyare styrsystem levereras allt oftare idag med OPC UA, *OPC Unified Architecture*, en plattformsoberoende vidareutveckling av OPC Classic inbäddad. *Entric* vill därför kunna erbjuda OPC UA och inte riskera att förlora kunder.

1.2 Syfte

Syftet är att med hjälp av den tekniska specifikationen av *OPC UA* utveckla en *OPC UA*-klient som kan implementeras i *MRS* och slutligen erbjudas som en kommunikationslösning till *Entrics* kunder.

1.3 Precisering av Frågeställningen

I programmeringsspråket och utvecklingsmiljön *Delphi* skall det utvecklas en mjukvara sådan att den kan implementeras i *MRS* och vara ansvarig för kommunikation med en OPC UA-Server.

Ytterligare skall det utvecklas en minimal OPC UA Stack samt ett minimalt OPC UA *SDK*, *Software Development Kit*, i *Delphi*. Detta för att kunna kommunicera med en OPC UA-Server. UA Stacken och *SDK*:t skall endast implementera funktioner som underlättar eller krävs för att implementera kraven enligt kravspecifikationen beskrivna i 1.3.1.

1.3.1 Kravspecifikation

Kraven mjukvaran har för att anses fullständig för *Entrics* bruk är följande:

- Etablera, behålla samt avsluta kontakt med en OPC UA-Server.
- Möjligheten att upptäcka OPC UA-Servrar och dess ändpunkter.
- Navigera sig i en OPC UA-Servers *adressrymd*.
- Läsa samt prenumerera på värden.
- Hantering av oförutsedda fel som innebär kontaktförlust, t.ex strömavbrott.
- Etablera kontakt samt återskapa prenumerationer från en konfigurationsfil.
- Skall utvecklas i *Delphi*.

OPC UA är uppbyggt av noder. Varje variabel, datatyp, referenstyp, objekt etcetera är en nod. Med *adressrymd* syftas på alla noder en OPC UA-Server innehåller.

1.4 Avgränsningar

En fullständig kommunikationsstack eller ett fullständigt *SDK* ska ej utvecklats. Endast funktionalitet som krävs för att uppnå kraven definierade i 1.3.1 ska implementeras. Det är alltså inte lämpligt att återanvända kommunikationsstacken utvecklad här för att utveckla en OPC Server. OPC UA stödjer flera tjänster, men dessa betraktras ej då de inte är efterfrågade till OPC UA-Klient inom projektet. OPC UA erbjuder stöd för tre olika transportprotokoll varav endast det binära OPC TCP implementeras i denna lösning, detta då övriga protokoll ej kommer att användas i miljön denna mjukvara är avsedd för.

Att implementera applikationen i *MRS* ligger utanför detta projektets omfattning. Applikationen bör dock vara i ett sådant stadie att den, förhållandevis smärtfritt, kan implementeras i *MRS*.

2 Teknisk Bakgrund

I detta kapitel kommer centrala begrepp och teknologi för rapporten att redovisas och förklaras. Fokus kommer att läggas på OPC Standarden. Först ges dock en kort bakgrund av programmeringsspråket som har använts för utveckling.

2.1 Delphi

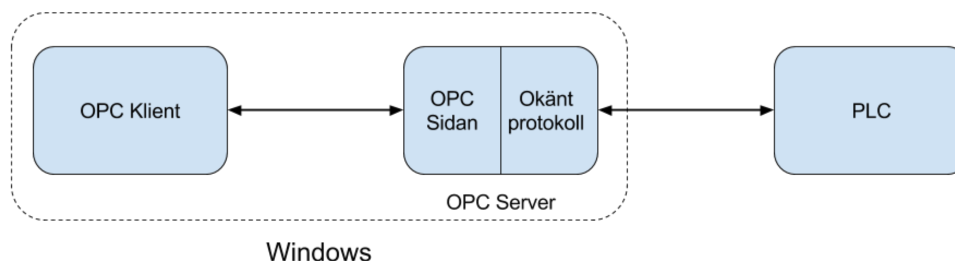
Utveckling av mjukvaran har skett i *Delphi*. Delphi är en vidareutvecklad variant av *Object Pascal*[4]. *Object Pascal* är en objektorienterad version av programspråket *Pascal*. Delphi är ett språk likt Java där det är någorlunda plattformsoberoende. Det är olikt Java och mer likt C/C++ då det inte har någon skräphantering av minne. Delphi exekveras inte i någon virtuell maskin som jämförelsevis Java görs i JVM, *Java Virtual Machine*. '

Delphi är inte gratis och för detta uppdrag har *Entric* införskaffat akademiska licenser. Företaget *embarcadero* distribuerar *Delphi*-licenser[1].

2.2 OPC Classic

OPC Classic är en kommunikationsstandard för datorer och inbyggda system specificerad av OPC Foundation. Specifikationen för OPC Classic släpptes 1996 och dess syfte var att abstrahera *PLC*-specifika protokoll som Modbus. PLC står för programmerbar logisk kontrollenhet och är en industriell dator som kontrollerar en eller fler hårdvaruenheter. Abstraheringen ger upphov till ett standardiserat gränssnitt, OPC-gränssnittet, som översätter diverse förfrågningar den tar emot till enhetsspecifika förfrågningar. Med OPC-gränssnittet kunde användare välja produkter samt komponenter bäst lämpade för deras ändamål. Genom ett OPC-gränssnitt kunde dessa komponenter kommunicera med varandra och utbyta data, vilket innan varit problematiskt[5].

OPC Classic lät användare läsa bland annat realtidsdata och äldre data. Det var möjligt att prenumerera på händelser samt data- och status-förändringar[5]. OPC Classic-specifikationen utvecklades så att Microsofts COM/DCOM-teknologi fick ansvara för säkerheten. Lösningen var därför bunden till Windows[5].



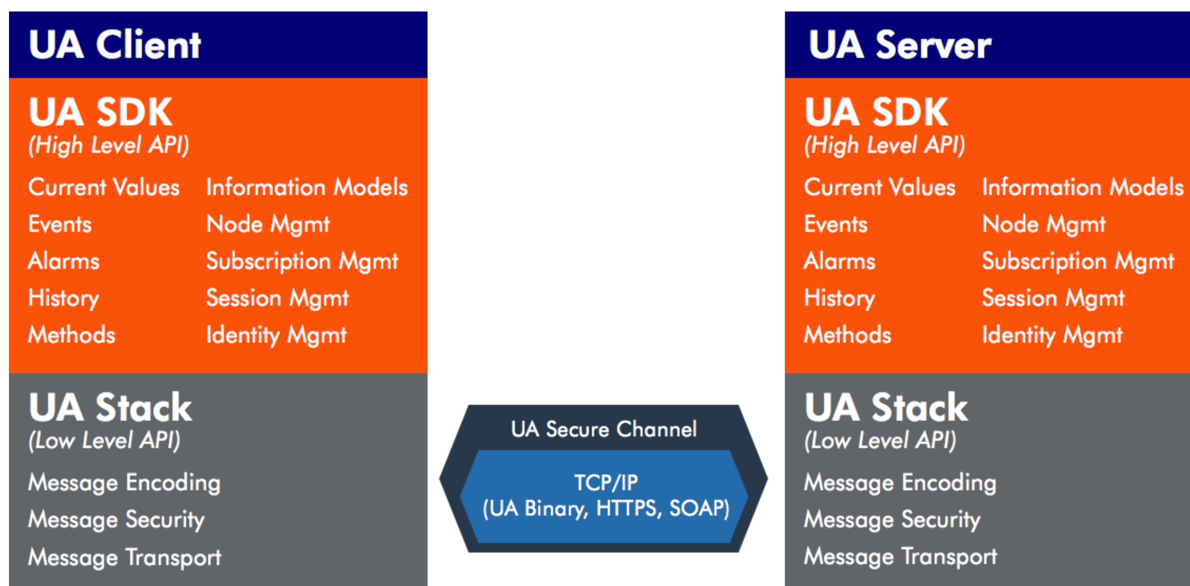
Figur 2.1: *OPC Klient, Server och PLC-enhet.*

[1]Mer om embarcadero på: *https://www.embarcadero.com/*

OPC-gränssnittet är separerat i två delar, en OPC-klient samt en OPC-server. Detta visas i förhållande till en *PLC* i figur 2.1. Vanligtvis interagerar en användare med en OPC-klient som i sin tur kommunicerar med en OPC-Server. OPC-Servern är den delen av systemet som sedan översätter de generiska OPC-förfrågningarna till specifika förfrågningar beroende på enhet[5]. OPC-servern är alltså delen av OPC-gränssnittet som har kontakt med en datakälla.

2.3 OPC UA

OPC UA är den modernare standarden som kom efter OPC Classic. OPC UA-standarden definieras i en grupp av specifikationer distribuerade av standardens ansvariga organisation, OPC Foundation[2]. Grundtanken var att åstadkomma oberoende från specifika operativsystem, algoritmer och övriga val av implementation. Den föregående standarden var bunden till operativsystemet *Windows* då dess implementation baserades på Microsofts DCOM-teknologi[5]. Initialt var detta tämligen negligerbart vid tiden då standarden utformades, men det har i ökande grad kommit att bli en begränsning. OPC UA baserar inte sin säkerhet på denna teknologi och är därför applicerbar för en större utsträckning av hårdvaror och operativsystem.



Figur 2.2: *Hur OPC UA kommunikationen är uppbyggd[6].*

En OPC UA Klient eller Server är uppbyggd av två lager. En kommunikationsstack och ett *SDK*. Kommunikationsstacken implementerar säkerhet, kodning samt transport av OPC UA meddelanden. *SDK*:t definierar funktioner för de olika tjänsterna och är också det *API*, *Application Programming Interface*, som används flitigast för implementering av en UA Klient eller UA Server. Figur 2.2 från *Prosys OPC* visar en OPC klient samt Server som kommunicerar via en säkerhetskanal(se 2.4.2).

[2]OPC Foundations hemsida kan nås på www.opcfoundation.com

OPC-UA erbjuder likvärdig funktionalitet gentemot OPC Classic, men lägger även till ytterligare funktioner innefattande:

- Servrar har möjligheten att erbjuda en tjänst för att upptäcka OPC-UA Servrar, *Discovery Service* (Se 2.4.3), vilken klienter kan nyttja för att hitta servrar över valt nätverk.
- Organisering av data i hierarkisk struktur för förbättrad användarvänlighet.
- Serverdefinierade program eller metoder kan exekveras av klienten.
- Varierbar säkerhet enligt egenvald specifikation.
- Möjlighet för utökning med önskade datatyper och algoritmer.

2.4 OPC UA Kommunikation

OPC UA standarden erbjuder ett flertal tjänster med standardiserad kommunikation mellan klient och server, samt möjligheten för en server att erbjuda egendefinierade tjänster. I detta avsnitt beskrivs kommunikationen som specifikt används i den utvecklade applikationen.

2.4.1 OPC UA Meddelandestruktur

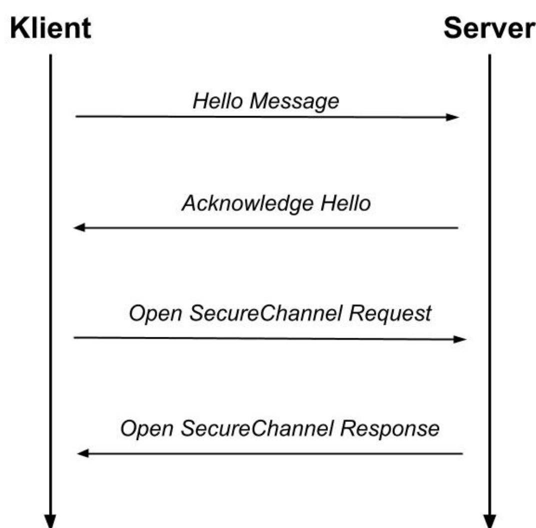
All OPC UA-kommunikation sker i form av meddelanden. Alla meddelanden antas ha en viss struktur som finns beskriven i specifikationen för OPC UA, se figur 2.3. Varje meddelande har ett så kallat meddelandehuvud. Meddelandehuvudet beskriver hur stort meddelandet är, om det är uppdelat i flera stycken, samt en bytesträng som beskriver vilket typ av meddelande det är. Meddelandehuvudet är inte avancerat och beskriver endast simpla typer. Efter meddelandehuvudet följer ett säkerhetshuvud. I säkerhetshuvudet beskrivs vilken säkerhet som används samt även identifierare för användaren. Sekvenshuvudet följer efter säkerhetshuvudet och sekvenshuvudet beskriver vilket meddelande i följden det är. Till slut återfinns kroppen av meddelandet.

Figur 2.3: *Meddelande struktur för OPC UA.*

I kroppen finns oftast något som beskriver en tjänst, till exempel att läsa ett värde. Efter kroppen kan det finnas meddelandefot samt eventuella signaturer. De lagren på meddelandet läggs på av de olika lagren i till exempel en OPC Klient. *SDK:t* beskriver meddelandekroppen, säkerhetskanalen kodar meddelandekroppen och lägger till säkerhetshuvudet, sekvenshuvudet samt meddelandehuvud.

2.4.2 Säkerhetskanal *(Secure Channel)*

Alla kommunikation mellan en klient och server måste ske över en på transportprotokoll-nivå säkrad kanal. Undantaget till detta är de meddelanden som upprättar själva kanalen. Dessa består av en initial hälsning *Hello Message* till servern som svarar med godkännande till kommunikation, *Acknowledge*. Efter detta följer en *OpenSecureChannel*-förfrågan, se figur 2.4. Önskade parametrar specificeras och svarsmeddelandet skickas från serverns sida med nödvändig information för att kommunicera över den nya kanalen.

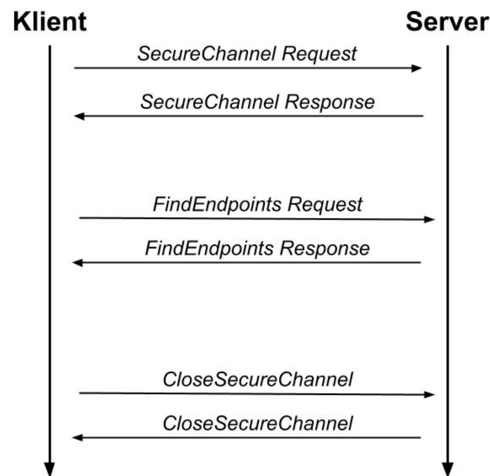


Figur 2.4: *Initial kontakt med en OPC Server för att öppna en säkerhetskanal (Secure Channel).*

2.4.3 Tjänst för att upptäcka OPC-Servrar *(Discovery Service)*

Discovery-tjänster kommunicerar med en *discovery*-server som potentiellt kan annonsera ett flertal servrar. Denna server är inte nödvändigtvis samma som den slutgiltiga kommunikationspunkten som eftertraktas.

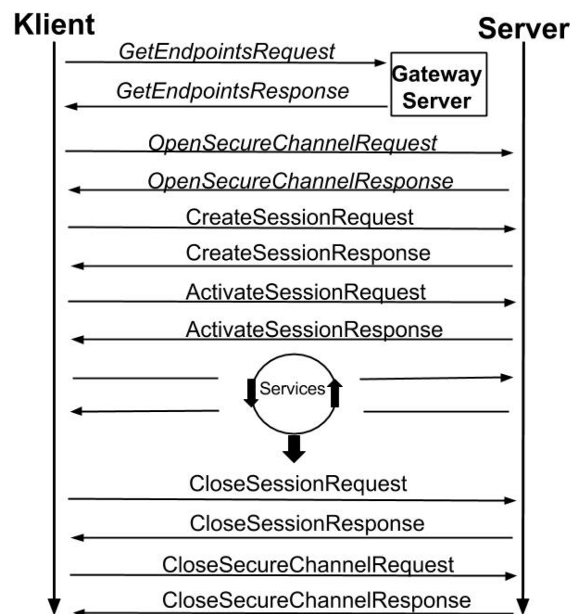
Denna tjänst består av två primära funktioner, *FindServers* och *FindEndpoints*. *FindServers* är tjänsten för att hitta potentiella servrar på vald enhet. *FindEndpoints* syfte är att returnera de olika anslutningsmöjligheterna, ändpunkterna, som är associerade med servern. I figur 2.5 visas de anrop som krävs för att ta reda på en OPC UA-servers ändpunkter.



Figur 2.5: *De anrop som krävs för att hitta en servers ändpunkter.*

2.4.4 Sessionstjänsen *(Session Service)*

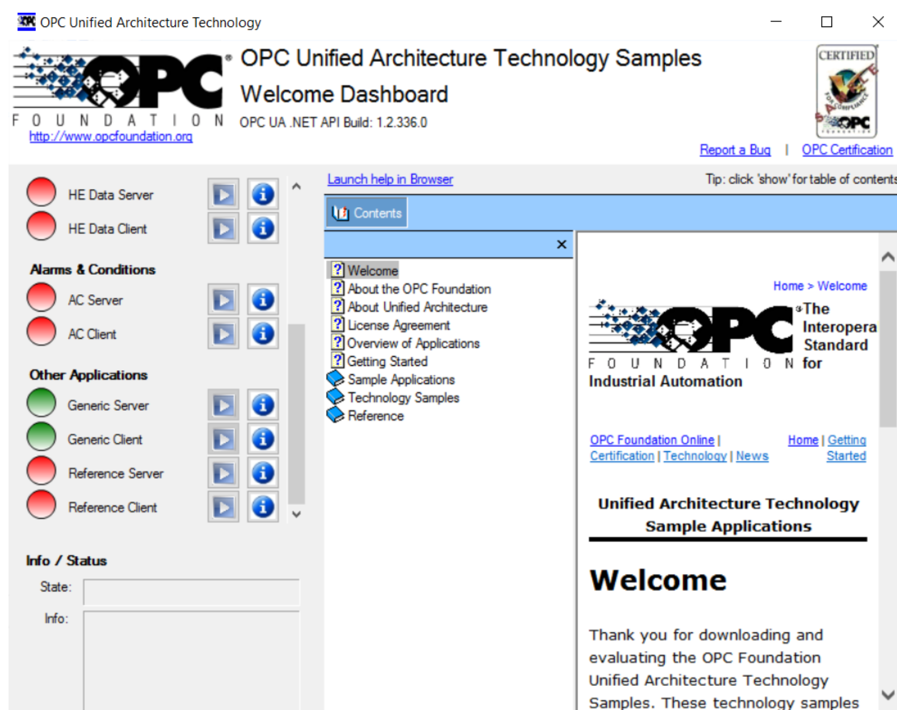
Syftet med en session är möjligheten att lägga till önskad säkerhet och övriga parametrar ovanpå vad som implementeras på transportprotokoll-nivån av säkerhetskanalen. Tjänsten består av tre funktioner: *CreateSession*, *ActivateSession* och *CloseSession*. Alla tjänster utöver *FindEndpoints*, *FindServers* samt säkerhetskanalen kräver att en session är etablerad. En samlad bild av hur ett anrop för att etablera en långvarig kontakt mellan en OPC UA klient samt server återfinns illustrerad i figur 2.6.



Figur 2.6: *Anropsföljd av förfrågningar för att skapa samt avsluta en session.*

3 Metodbeskrivning

För att bygga en OPC UA kommunikationsstack eller ett SDK är det nödvändigt att förstå exakt vad kommunikationsstacken eller SDK:t skall innehålla, hur alla delar håller ihop och hur OPC UA enheter kommunicerar med varandra. Allt detta förklaras i specifikationerna för OPC UA. Specifikationerna är uppdelade i flera delar och alla delar är inte nödvändiga att läsa beroende på syftet. OPC UA Specifikationerna är distribuerade av OPC Foundation och för att få tillgång till dem behövs ett medlemskap i deras organisation. Med detta medlemskapet får man även tillgång till färdigskrivna OPC UA-applikationer[1] man kan använda för testning samt verifiering. Figur 3.1 visar de olika standardapplikationerna OPC Foundationer tillhandahåller.



Figur 3.1: *OPC UA Applikationer samt OPC Dashboard, vilket OPC Foundation tillhandahåller vid ett medlemskap.*

Källkoden för dessa applikationer är delvis tillgängliga[2]. Koden är dock så pass objektorienterad att det är svårt att få någon överblicksbild av vad det är som händer vid vilka tillfällen. Utöver de strikt nödvändiga servrarna som erbjuds av OPC Foundation, finns det även användning för den generiska klienten vilket tas upp i detalj i 3.3.

3.1 Versionshantering

När man delar upp arbetsuppgifter mellan flera personer där en central del av arbetet är att skriva kod för någon slags funktion är det väldigt fördelaktigt att kunna arbeta parallellt. Diverse versionshanteringsprogram möjliggör detta. Versionshantering är också bra för att kunna

[1] https://opcfoundation.org/developer-tools/developer-kits-unified-architecture/sample-applications

[2] På OPC Foundations github, https://github.com/OPCFoundation, finns källkoden i olika språk för en OPC UA Stack och SDK

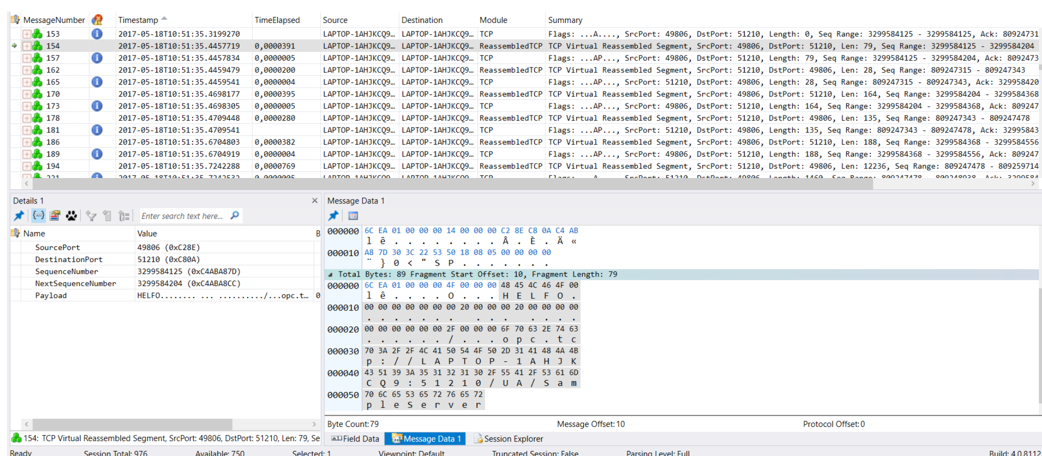
gå tillbaka till en tidigare version av källkoden. Detta kan vara önskat av olika anledningar.

I detta projekt har versionshanteringsprogrammet *Mercurial*, som kör på *Entrics* server använts. För att kommunicera med versionshanteringsservern har gratisprogramvaran *TortoiseHg* använts[3].

3.2 Paketsniffning

Som senare nämns i 3.3 finns det anledning att analysera meddelanden skickade mellan en korrekt implementerad server och klient för att verifiera det korrekta kommunikationsprotokollet för diverse meddelanden. Då merparten av trafiken under testning är förlagd på en och samma dator innebär detta att meddelanden aldrig passerar ett verkligt nätverksinterface och man behöver alltså fånga och läsa, även kallat sniffa, paket på *loopback*-adressen.

Det existerar många olika tredjepartslösningar för paketsniffning. Ett av dessa med möjlighet att observera trafik över *loopback*-adressen är *Windows Message Analyzer*. *Windows Message Analyzer* visas i figur 3.2. Det välkända programmet för paketsniffning, *Wireshark*, erbjuder inte möjligheten att sniffa efter paket på *loopback*[4]-gränssnittet i Windows och därför är Microsofts *Windows Message Analyzer* ett lämpligt val[7]. *Windows Message Analyzer* involverar diverse funktionalitet, men för observerande av trafiken över en specificerad port erbjuds ett fönster med utgående och inkommande meddelanden i tidsordning. Analys av dessa meddelanden möjliggör identifiering av vilka meddelande de är, vilka fält som ingår samt vilka värden som valts i en fungerande lösning för att få ett korrekt svar från servern.



Figur 3.2: *Ett, med hjälp av Windows Message Analyzer, sniffat 'Hello'-meddelande från en OPC UA-klient till en OPC UA-server.*

[3]Mer om TortoiseHg och Mercurial på: *https://www.mercurial-scm.org/wiki/TortoiseHg* respektive *https://www.mercurial-scm.org/*

[4]Vad en *loopback*-adress är beskrivs i *http://www.pcmag.com/encyclopedia/term/57812/loopback-address* (hämtad 2017-06-15)

3.3 Att jämföra bytes med hjälp av paketsniffning

Att kommunicera enligt en vedertagen kommunikationsstandard innebär att alla meddelanden som skickas och tas emot är strikt definierade. Den exakta kompositionen av olika meddelanden står att finna i den officiella OPC UA dokumentationen. Det står dock klart att dokumentationen är bristfällig trots multipla iterationer och korrigeringar av den. Tre primära fel orsakar problematik vid konstruktionen av ingående tjänster och de meddelanden som är relaterade till dem.

3.3.1 Definitionen av *Null*-värden

Ett vanligt förfarande i olika meddelanden är att diverse fält ej är obligatoriska beroende på vad som önskas uppnås. I denna situation skickas ett väldefinierat värde som korresponderar mot avsaknaden av ett värde, ett null-värde. Inom OPC UA standarden är detta särskilt från ett tomt värde. För klarifiering kan alltså exempelvis en fältkomponent av ett meddelande skickas med indexerade värden, noll värden eller med ett null-värde. Signifikansen av noll-värden och ett null-värde varierar beroende på meddelandetyp. För somliga har de samma innebörd, men i andra fall kan de innebära olika förfrågningar eller svar.

Problematiken runt detta är tvådelad. Dokumentationen är missvisande angående vissa meddelande, varpå fält som deklarerats som nödvändiga inte används i praktiken. I de fall där ett värde ska deklareras null ligger problemet i hur ett sådant värde bör representeras i olika meddelanden. Dokumentationen specificerar enbart att ett null-värde är godtagbart med null-värdes definitionen lämnad som underförstådd. Det finns även fall då felaktig information om hur ett null-värde bör representeras är specificerad för ett meddelande. I båda dessa fall är det centrala problemet att det finns tre olika sätt att deklarera värdet null, se tabell 3.1.

Längd Noll	Null	Blank
[00_{16}, 00_{16}, 00_{16}, 00_{16}]	[FF_{16}, FF_{16}, FF_{16}, FF_{16}]	[ ]

Tabell 3.1: *Potentiella null-värden.*

3.3.2 Strukturparametrar

Dokumentationen av OPC UA använder sig genomgående av terminologin struktur för att beskriva logiska grupperingar av värden, *RequestHeader* och *NodeId* är exempel på strukturparametrar. Ordet struktur har dock även en definierad programmatisk innebörd inom OPC UA. Definitionen av en struktur är att den kräver inledande beskrivande bitar i form av sitt korresponderande *nodid*. Ett nodid består av en kodningsbyte, namnrymd samt identifierare. Ett exempel på ett nodid finns illustrerat i tabell 3.2.

Kodningsbyte	Namnrymd	Identifierare
[01_{16}]	[00_{16}]	[BE_{16}, 01_{16}]

Tabell 3.2: *Nodid för OpenSecureChannelRequest-Encoding-DefaultBinary.*

Detta ger möjligheten för egendefinierade strukturer då de kan särskiljas från officiella OPC UA definierade strukturer på namnrymdsbyten. Denna dubbla betydelse av order struktur innebär dock att det i definitionen av vissa meddelanden är omöjligt att säkerställa huruvida beskrivande bitar krävs eller ej.

3.3.3 Jämförelse av bytes

Ett förfarande för att verifiera vilka val och krav som faktiskt gäller vid de oklarheter som beskrivits i 3.3.1 och 3.3.2 är att ta nytta av en färdig OPC UA-klient. OPC Foundation tillhandahåller ett utbud av generella klienter och servrar tillgängliga för testning i lokal miljö. Möjligheten finns då att skicka olika meddelanden från klient till server för att fånga upp de bytes som skickas mellan applikationerna. Då konstruktionen av dessa meddelanden är känd från dokumentationen är det möjligt att analysera dessa uppfångade meddelanden och verifiera vilka fält som faktiskt skickas, samt deras representation av *null*-värden och användning av strukturparametrar.

3.4 Felsökning

Ett kontinuerligt förfarande genom konstruktionsprocessen av applikationen är upptäckande, verifiering och hantering av buggar. Den faktiska korrigeringen av kod som genererar oönskat resultat är nödvändigtvis gjord manuellt genom alla stadier av projektet.

Att upptäcka buggar och verifiera den exakta felande logiken kan göras på flera vis. I tidiga skeden av applikationsutveckligen är brytpunkter ett enkelt sätt att analysera och verifiera troliga felområden då det i detta stadie typiskt är förhållandevis lätt att nå insikt angående felet. Detta blir dock i ökande grad ohållbart med utökning av koden och i synnerhet då trådar utöver huvudtråden implementeras. En enkel form av felsökning i detta stadiet är utskrift av strängar som korresponderar till olika sektioner av kod. Detta ger en bättre insikt i vilka trådar och funktioner vars exekvering är oväntad. Analysering av exekveringsstacken i *Delphi* vid ett fel ger även stor insikt vid felsökning.

3.4.1 Praktiska begränsningar

Testfall för korrekt funktionalitet av enklare funktioner såsom translatering av en sträng till byte-form är enkelt genomförbart, se algoritm 1 samt 2, och erbjuder en nivå av kvalitetskontroll och regressionstestning.

Utöver denna typ av hjälpfunktioner innefattar dock applikationen ett stort antal komplicerade procedurer med olika komponenter som gör testning väsentligt mer komplicerad. En komplicerande faktor är kommunikationen mellan klient och server som är en grundsten i OPC UA. Denna kommunikation innebär att ett fel som uppstår i klienten inte nödvändigtvis beror på en bugg i koden. Utomliggande nätverksproblem och felaktig hantering från serverns sida är möjliga felorsaker. Applikationen är även baserad på ett flertal trådar som exekverar pseudoparallellt. Synkroniseringfel mellan dessa trådar är högst komplicerat att skriva testfall för, och med tanke på den obestämda körtiden svårt att ge rimlig garanterad säkerhet för utan

Algoritm 1 Translatering av sträng till byte-form

```
1: function EncodeString(val:TBytes)
2:     TempByteArray ← TEncoding.UTF8.GetBytes(val)    ▷ TEncoding.UTF8.GetBytes är en standardfunktion i Delphi som returnerar varje bytes korresponderande ASCI-värde i en sträng. T.ex: 'abc' blir [97, 98, 99]
3:
4:     if val = '-1' then    ▷ En sträng som har värdet '-1' tolkas som en null-sträng.
5:         Result ←Int32(−1).Encode
6:         return Result : TBytes
7:     end if
8:
9:     SetLength(Result, LENGTH(TempByteArray)+4)
10:    EvenMoreTempByteArray ←Int32ToTbytes(LENGTH(TempByteArray))
11:    move(EvenMoreTempByteArray[0], Result[0], 4)
12:    move(TempByteArray[0], Result[4], LENGTH(TempByteArray))
13:
14:    return Result : TBytes
15: end function
```

Algoritm 2 Test för translatering av sträng till byte-form

```
1: function testEncodeString
2:     Result ←True;
3:     Result ←Result AND ( encodeString("-1") AND [FF16, FF16, FF16, FF16] )
4:     Result ←Result AND ( encodeString("") AND [0, 0, 0, 0] )
5:     Result ←Result AND ( encodeString("Test") AND [0416, 0, 0, 0, 5416, 6516, 7316, 7416] )
6:     return Result : Boolean
7: end function
```

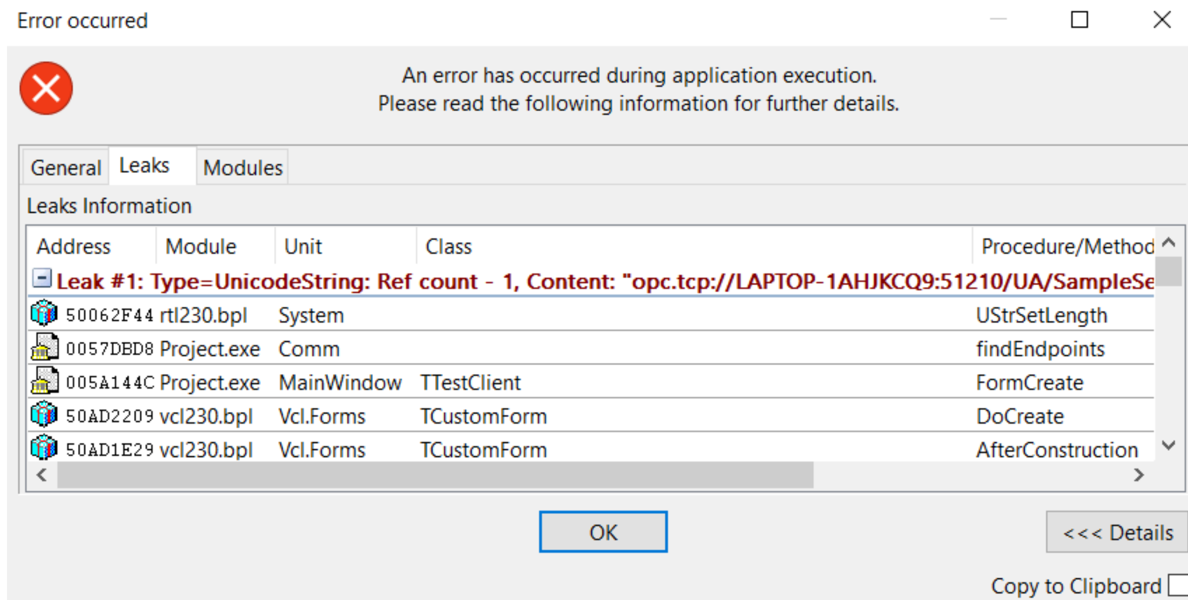
matematiska belägg angående exekveringsslingan.

Med tidsaspekten av projektet i åtanke hanteras felsökning på enkelt vis, med testfall begränsade till lämpliga hjälpfunktioner.

3.5 Minnesläckor

Minnesläckor uppstår när minne har allokerats, och sedan inte frigjorts innan applikationen terminerat. Ett exempel på minnesallokering är när objekt skapas. Objekt tar upp arbetsminne, vilket innebär att minne ligger bundet till ett program som redan har stängts. Detta problem är principiellt lätt att undvika då man enbart kontinuerligt behöver frigöra alla objekt man skapar och allt annat minne man allokerar under utvecklandet av applikationen. Försvårande faktorer som multipla trådar och passande av objekt eller passande av pekare till objekt som parametrar kan dock göra det oklart om objekt frigörs korrekt. Detta tillsammans med den mänskliga faktorn gör att minnesläckor är högst troliga att uppkomma under utvecklingsprocessen.

Ett tredjepartsprogram som automatiserar upptäckten av existerande minnesläckor är *Eureka logs*[5]. Vid stängning av applikationen ger *Eureka logs* respons angående förekomsten av minnesläckor efter programavslut. Figur 3.3 visar en minnesläcka rapporterad av *Eureka Logs*.



Figur 3.3: *Minnesläcka rapporterad av Eureka Logs.*

[5]För mer information om eureka logs: https://www.eurekalog.com/

4 Konstruktion

Den designade applikationen byggdes upp enligt en kravspecifikationen med övrig potentiell funktionalitet enligt OPC-UA standarden utelämnad. Funktionell interaktion mellan server och klient kan uppnås på ett flertal vis, och med olika styrkor och svagheter baserat på detta. Detta avsnitt detaljerar konstruktionsval och problemområden relaterade till dem. I denna del av rapporten används algoritmer för beskriva vissa delar av källkoden. Dessa algoritmer använder sig av vissa datatyper. Datatypen *TBytes* innebär ett fält av bytes. Andra datatyper såsom till exempel *Int32* innebär en 32-bitars heltalstyp.

4.1 Interaktion med *socketen*

När flera olika uppgifter, till exempel prenumerationer, läsning och förnyelse av säkerhetskanalen, ska dela på en och samma *socket* uppstår komplikationer. Programmet exekverar inte sekventiellt utan beror på externa händelser och event, vanligtvis från en användare eller resultatet av en användares beteende. Det är viktigt att varje förfrågan till servern får den responsen den förväntar sig och att detta sker asynkront så att anropande tråd kan arbeta med annat i väntan på svar.

4.1.1 Användning av *callback*-procedurer

Att hantera svaren när dem ankommer löstes med så kallade *callback*-procedurer. Detta innebär att en funktion som ska anropas sparas i en variabel. När svaret sen kommer anropas denna procedur med data från servern som inparameter. Algoritm 3 visar hur *callback*-funktionaliteten fungerar i en läsning av data på servern. Om *ReadRequest*, som anropas i algoritm 3, skriver till *socketen* och sen väntar på svar är anropet fortfarande inte asynkront. Detta eftersom *ReadRequest* då inte kan returnera förrän den fått svar och *OnReadButtonClick* kan i sin tur inte returnera förrän *ReadRequest* är färdig. Detta kan lösas genom att ha ett trådat objekt som ansvarar för alla läsningar samt skrivningar av respektive till *socketen*. Resultatet är *Buffertråden* som beskrivs i 4.1.2.

Algoritm 3 Exempel på *callback*-funktionalitet vid en Läs-förfrågning.

```
 1: procedure OnReadButtonClick
 2:     NodeId ← Valt elements Nod-Id
 3:
 4:     CallBackFunc ←(
 5:     procedure OnReceive(Data)
 6:         Visa Data på GUI.
 7:     end procedure
 8:     )
 9:
10:     ReadRequest(NodeId, CallBackFunc)    ▷ ReadRequest() bygger upp anropet som
    OPC-servern förväntar sig det och anropar sedan CallBackFunc när respons ankommer.
11: end procedure
```

4.1.2 Bufferträden

Bufferträdens huvudsakliga uppgift är att ansvara över all interaktion som behövs med *socketen*. Detta innebär att inget annat objekt kan varken skriva till respektive läsa från *socketen* utan att gå genom Bufferträden. En av de centrala delarna av Bufferträden är att skriva till *socketen*. Skrivning till *socketen* sker genom ett proceduranrop. Proceduren beskrivs i algoritm 4. Proceduren förväntar sig tre inparametrar, datan som ska skrivas, vilket *RequestID* som datan har samt en referens till en *callback*-procedur som ska anropas när just denna förfrågan får ett svar. *RequestID*:t är unikt över en session och serverns respons innehåller alltid detta unika *RequestID*. Ett *RequestID* har ingen definierad funktion mer än att det skall vara unikt och tolkas som ett 32-bitars heltal. Implementation av ett *RequestID* valdes då som en räknare som återställs då heltalet börjar närma sig det maximala värdet av ett 32-bitars heltal, det vill säga $2^{31} - 1 = 2,147,483,647$. Detta talet är så pass stort att det tar lång tid innan det återställs och kan då igen anses som unikt.

Algoritm 4 Bufferträdens procedur som anropas när något ska skrivas till socketen.

Require: Data, RequestID, OnMsgRcv = Referens till en callback-procedur
1: **if** Socketen är öppen **then**
2:     Spara $RequestID$ och $OnMsgRcv$ som ett par i en datastruktur.
3:     Skriv *Data* till socketen
4: **end if**

RequestID:t och referensen till *callback*-proceduren sparas som ett par i en datastruktur. Datastrukturen som valdes var en *hashmap* eftersom en *hashmap* har konstant tidskomplexitet för alla grundläggande operationer[1].

Algoritm 5 BufferTrådens huvudloop.

1: **while** not Terminated **do**
2:     *Data* ← ReadInputBuffer
3:     **if** *Data* is not empty **then**
4:         $ToInvoke$ ← GetFunc($Data.RequestID$)   ▷ *GetFunc* är en klass-specifik funktion för BufferTråden som returnerar rätt callback-funktion beroende på RequestID:t.
5:         **if** $ToInvoke \neq$ null **then**
6:             Remove(Data.RequestID)   ▷ Ta bort paret från datastruktureren eftersom meddalandet nu har mottagits.
7:             ToInvoke(Data)
8:         **end if**
9:     **end if**
10:     Sleep(C)   ▷ C = En konstant.
11: **end while**

Bufferträden ärver, som namngivningen indikerar, från trådbasklassen och är ett trådat objekt. Det är den trådade delen av objektet som ansvarar för läsningen av *socketen* och anropet av rätt *callback*-procedur. Bufferträdens huvudloop, som också läser från *socketen*,

[1]Med grundläggande operationer syftas här på insättning, sökning, hämtning samt borttagning av element.

beskrivs i algoritm 5. Det första som sker vid varje iteration av loopen är att data läses från *socketens inputbuffer*. Efter det anropas rätt *callback*-procedur, hämtad från datastrukturen som algoritm 4 skriver till.

4.1.3 Designfel i buffertrådens huvudloop

När programmet avslutar skickas en *CloseSession*-förfrågan och OPC-Servern ger ett *CloseSession*-svar. *CloseSession* är en klass-specifik procedur som tillhör klassen *Session*. *CloseSession* beskrivs i algoritm 6.

Algoritm 6 Hur anropet av CloseSession ser ut som används när programmet avslutas.

1: Bygg Msg enligt OPC UA-Specifikationen.
2:
3: $F \leftarrow($
4: **procedure** OnRcv($Data : TBytes$)
5:     Terminera Bufferttråden.
6:     Frigör Bufferttråden.
7: **end procedure**
8:)
9:
10: BufferThread.WriteMsg($Msg, Msg.RequestID, F$).

Bufferttråden skickar en *CloseSession*-förfrågan till OPC-servern och får inom en snar framtid ett *CloseSession*-svar i respons. Bufferttråden hämtar vilken procedur som ska anropas, vilket i detta fall är F i algoritm 6, och anropar sagd procedur. Denna proceduren säger att Bufferttråden ska terminera och sedan frigöras. När Bufferttråden är avslutad finns det inte längre något kvar som interagerar med *socketen* och *socketen* kan stängas vilket senare medför att programmet kan avslutas.

När man i *Delphi* försöker terminera en tråd så sätter man bara *terminated-flaggan* till att vara sann. För att slippa minnesläckor måste man även frigöra skapade objekt. En tråd kan inte frigöras förrän tråden har avslutat sitt *Execute*-anrop. En *deadlock*-situation uppstår när bufferttråden anropar F i algoritm 6 och försöker frigöra bufferttråden, det vill säga sig själv. Bufferttråden kan dock inte frigöras eftersom den fortfarande exekveras. Resultatet blir att bufferttråden väntar på frigöra sig själv tills den är färdig med sitt *Execute*-anrop, men för att blir färdig med sitt *Execute*-anrop måste den frigöra sig själv.

Anledning till ovan beskrivit *deadlock*-fel grundar sig i ett designfel i buffertrådens huvudloop. Enligt algoritm 5 får inte bufferttråden endast ansvaret att sköta all interaktion med *socketen*, utan även att anropa och exekvera alla *callback*-procedurer. Det finns alltså även ett potentiellt överbelastningsfel där bufferttråden är upptagen med att exekvera *callback*-procedurer istället för att läsa från *socketen*, som kan innehålla avgörande information för användaren eller en annan del av programmet.

4.1.4 Delegering av exekvering av *callback*-procedurer

Som beskrivet i 4.1.3 återfinns ett designfel i buffertrådens huvudloop. Felet leder till att buffertråden arbetar med uppdrag den inte var menad att hantera. En *deadlock*-situation uppkommer även när programmet ska avslutas. För att lösa dessa problem delegeras istället alla anrop och exekveringar av *callback*-procedurer till en temporärt skapad tråd vars enda uppgift är att exekvera en procedur och sedan avslutas. Buffertrådens huvudloop har ett nytt utseende vilket visas i algoritm 7 och den kortlivade temporära trådens mycket enkla *Execute*-procedur beskrivs i algoritm 8.

Algoritm 7 BufferTrådens huvudloop med delegering till temporärt skapad tråd.

```
 1: while not Terminated do
 2:     Data ← ReadInputBuffer
 3:     if Data is not empty then
 4:         ToInvoke ← GetFunc(Data.RequestID)
 5:         if ToInvoke ≠ null then
 6:             Worker ← WorkerThread.Create
 7:             Worker.ToInvoke ← ToInvoke
 8:             Worker.Data ← Data
 9:             Worker.FreeOnTerminate ← True
10:             Worker.Execute
11:             Sleep(C)
12:         end if
13:     end if
14: end while
```

Algoritm 8 WorkerTrådens Execute-procedur

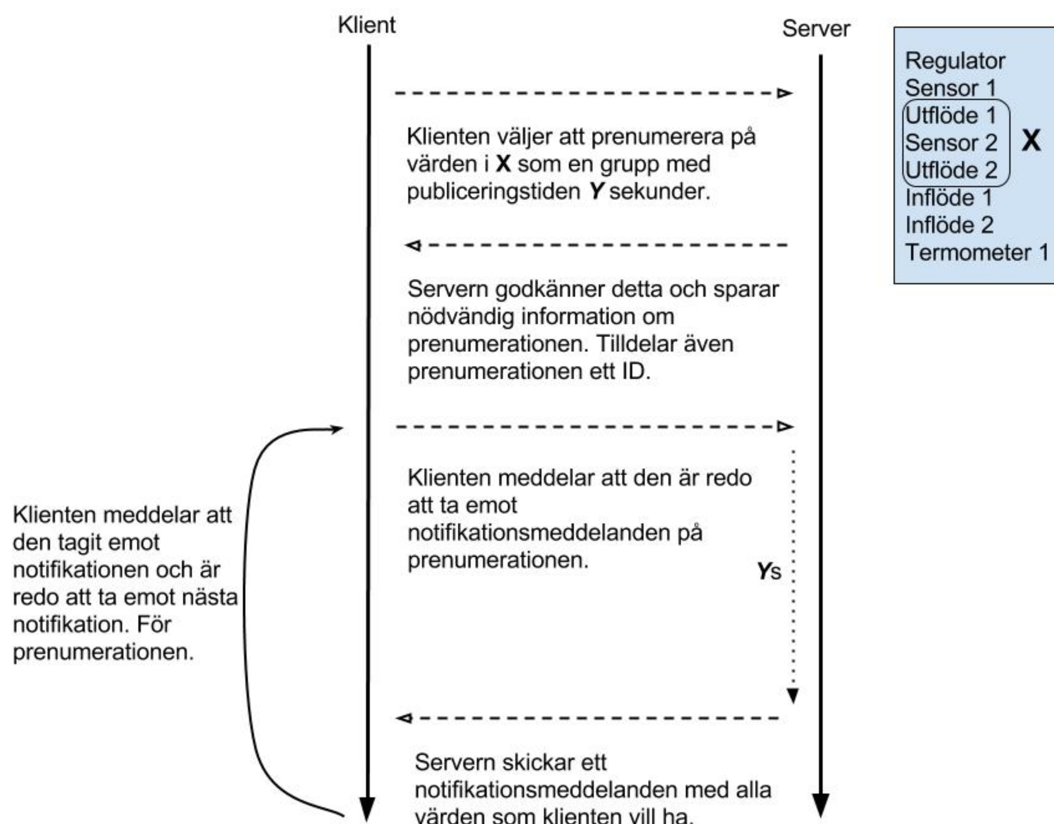
```
1: ToInvoke(Data)
2: Terminate
```

Med dessa ändringar kan man se att inga okända procedurer eller procedurer som riskerar att aldrig returnera anropas i buffertrådens huvudloop, vilket löser problemet beskrivna i 4.1.3.

4.2 Prenumeration på värden

Möjligheten att prenumerera på värden beskrivs i OPC UA-specifikationen och är en av de mest centrala delarna *Entric* begär att deras OPC UA klient har stöd för. OPC UA-specifikationen beskriver två olika typer av prenumerationer. Prenumerationer på värden samt prenumerationer på händelser. OPC UA-klienten som utvecklas med denna rapporten har endast stöd för prenumerationer på värden. Detta för att, istället för att prenumerera på händelser, har det valts att prenumerera på standardvärden varje server måste ha. I dessa ingår bland annat en prenumeration på serverns statusvärden.

I OPC UA (även OPC DA) så prenumererar man oftast på en grupp värden. Varje grupp består av en egen prenumeration där varje prenumeration har en specificerad tid som bestämmer inom vilket intervall klienten vill ha notifikationsmeddelande från servern. Figur 4.1 visar ett flödesschema mellan en OPC UA-klient och OPC UA-server för en prenumeration.



Figur 4.1: *Flödesschema över en prenumeration.*

När man skapar en prenumeration gör man det på en eller flera datakällor. Man kan definiera vissa egenskaper på denna datakälla, till exempel att man bara vill ha värdet om den går utanför att värdesintervall. Ett värdesntervall som bestämmer om ett värde ska skickas eller ej kallas för ett *dödband*. Det är även möjligt att säga till servern att läsa värdet på ett annat intervall än prenumerationens publiceringstid. Vill man läsa datakällan varje $250ms$ och den ligger i en prenumeration som har ett publiceringsintervall på två sekunder kommer det i varje notifikationsmeddelande finnas åtta värden för sagd datakälla.

4.2.1 Prenumerationstråden

Enligt OPC UA specifikationen och vad som visas i figur 4.1 så ska klienten vid varje respons av notifikationsmeddelande meddela servern att den har tagit emot notifikationen och är redo att ta emot en ny. Detta problem löses enklast genom användning av en tråd för varje prenumeration. Denna tråd ansvarar för att hantera varje mottagen notifikation för sin prenumeration och att meddela att den vill ta emot nästa notifikation. Dessa meddelanden kallas notifikations-

förfrågningar.Varje notifikationsförfrågning är generaliserad genom metoden *Publish* i klassen *Session*, definierad i algoritm 9. En prenumerationtråds huvudloop är definierad i algoritm 10.

Algoritm 9 Sessionens Publish-procedur.

Require: SubId, SequenceNumbersToAck : Int32, ToInvoke = Referens till *Callback*-procedur

1:
2: Bygg *ToSendData* m.h.a *SubId* och *SequenceNumbersToAck* enligt OPC UA-specifikationen.
3:
4: $F \leftarrow$
5: (
6: **procedure** OnMsgRcv(*AnswerData* : TBytes)
7:     ToInvoke(*AnswerData*)
8: **end procedure**
9:)
10:
11: SendMsg(*ToSendData*, *F*)

Algoritm 10 En prenumerationtråds huvudloop.

1: **while** not Terminated **do**
2:
3:     **if** *HasRcvdMsg* **then**
4:         *HasRcvdMsg* ←False
5:         Session.Publish(*MySubId*, *NextSeqNum*, *OnRcv*) ▷ *OnRcv* är en metod definierad av objektet som skapat prenumerationen. *OnRcv* skulle t.ex. kunna vara att rita ut värdena i GUI:t. *OnRcv* sätter också *HasRcvdMsg* till *True*.
6:     **end if**
7:     Låt tråden sova tills ungefär innan nästa notifikation bör mottas.
8:     **while** not *HasRcvdMsg* **do**
9:         Sleep(C) ▷ C = En konstant.
10:     **end while**
11: **end while**

4.2.2 Överskrivning av data i notifikationsförfrågningar

Ett problem visar sig dock när man har flera prenumerationer igång. Problemet redovisas bäst i form av en händelsetabell, tabell 4.1. Som tabell 4.1 visar så i händelse *3*, anropas inte *P1.ToInvoke*. Detta beror på att senaste anropade prenumeration, *P2*, satt den lokala variabeln, eller referensen, till *P2.ToInvoke*. Lösningen är alltså att låta *Publish*-procedurens egna *callback*-procedur hämta prenumerations-id:t från svaret. Sedan hämta rätt referens till en procedur att anropa från en datastruktur och anropa den. På det sättet frånkommer man felet när flera trådar anropar samma procedur där lokala variabler kan ersättas. Denna lösning som också används i implementationen är beskriven i algoritm 11. Prenumerationstråden är oförändrad med avvikelsen att *OnRcv* inte används som parameter.

Numrering	Händelse	Publish.ToInvoke
1	P1 Anropar Session.Publish med P1.ToInvoke	*P1.ToInvoke*
2	P2 Anropar Session.Publish med P2.ToInvoke	*P2.ToInvoke*
3	P1:s Svar mottags men P2.ToInvoke anropas.	*P2.ToInvoke*
4	P2 Anropar Session.Publish med P2.ToInvoke för att ta emot nästa notifikation.	*P2.ToInvoke*
5	P2:s Svar mottags och P2.ToInvoke anropas.	*P2.ToInvoke*

Tabell 4.1: *Händelseförlopp för två prenumerationer med Publish-procedurens lokala referens till callback-proceduren ToInvoke.*

Algoritm 11 Sessionens Publish-procedur som löser felet som beskrivs i tabell 4.1.

Require: SubId, SequenceNumbersToAck : Int32

1:
2: Bygg *ToSendData* m.h.a *SubId* och *SequenceNumbersToAck* enligt OPC UA-specifikationen.
3:
4: $F \leftarrow$
5: (
6: **procedure** OnMsgRcv(*AnswerData* : *TBytes*)
7:     *ToInvoke* ←HashMap.Get(AnswerData.SubscriptionID) ▷ HashMappen tillhör *Session*-objektet som fyller datastrukturen för varje ny prenumeration som skapas.
8:     **if** ToInvoke ≠ null **then**
9:         ToInvoke(AnswerData)
10:     **end if**
11: **end procedure**
12:)
13:
14: SendMsg(*ToSendData*, *F*)

En annan tänkbar lösning är att behandla *Publish*-proceduren som en kritisk region. En kritisk region är ett kodblock där endast en tråd får befinna sig i taget. Detta kallas även ett atomiskt kodblock. Denna typ av lösning är dock inte bra i detta fallet då anropande tråd måste vänta i det atomiska kodblocket tills svar har mottagits. Detta hindrar andra trådar från att ta emot eller skicka notifikationsmeddelanden respektive notifikationsförfrågningar.

4.2.3 Samling av prenumerationstrådar

Om en användare skapar flera prenumerationer kommer varje prenumeration ha sin egen tråd. Att ha flera trådar som anropar samma procedur eller ändrar på samma variabel innebär att alla dessa operationer måste vara atomiska. Att göra dessa operationer atomiska är inte nödvändigtvis svårt. Den mänskliga faktorn medför dock att fel kan göras där trådarna hamnar i diverse låsningar, som kan vara svåra att upptäcka i efterhand.

Eftersom prenumerationstråden sover den största delen av tiden finns det inte heller ett

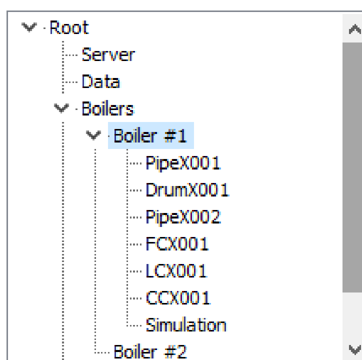
tydligt argument för varför varje prenumeration bör vara en egen tråd. Man kan istället lösa problemet med endast en tråd som läser vilka prenumerationer som godkänner nästa notifikationsmeddelande från en kö där läs- och skriv-operationer är atomiska. Prenumerationsobjekten skriver till denna kön när objektet är redo att mottaga nästa notifikationsmeddelande. Prenumerationstråden, som samlar alla prenumerationers kommunikation, är beskriven i algoritm 12. Denna algoritm används i nuvarande version av applikationen.

Algoritm 12 Tråden som samlar alla prenumerationers kommunikation i en tråd.

```
 1: while not Terminated do
 2:     if QueueLock.TryEnter then
 3:         if not Queue.isEmpty then
 4:             Session.Publish(Queue.Dequeue)
 5:         end if
 6:         QueueLock.Leave
 7:     end if
 8:     if QueueLock.IsEmpty then Sleep(C)            ▷ C = En konstant.
 9:     end if
10: end while
```

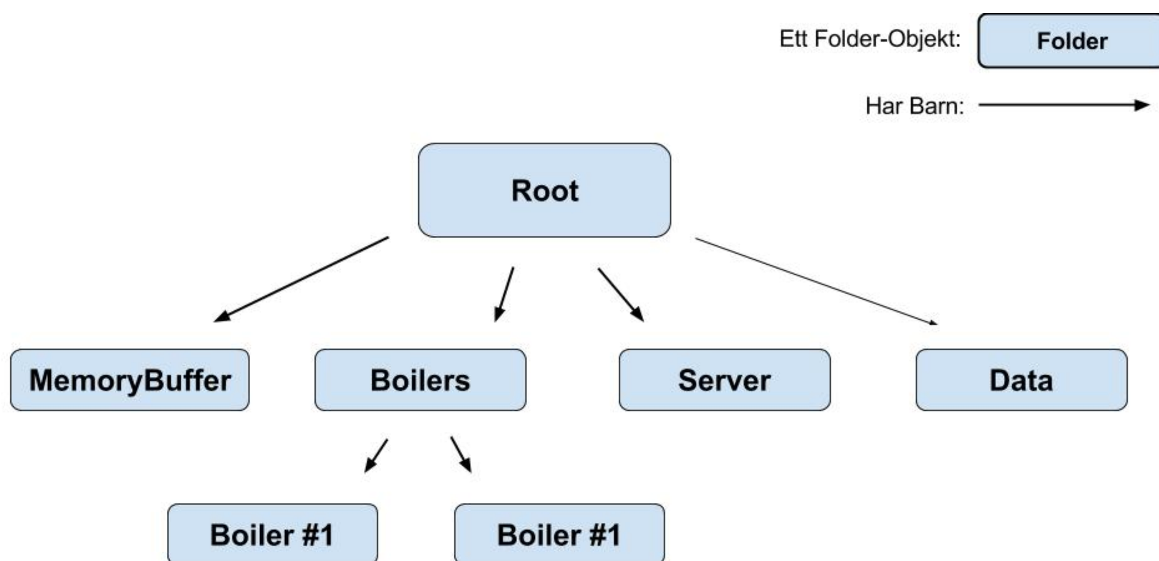
4.3 Bläddra i OPC-serverns *addressrymd*

En OPC-Servers *addressrymd* är strukturerad likt en filhierarki. Noder har olika typer av referenser till andra noder. *Browse*-tjänsten ger en OPC-klient möjligheten att på ett intuitivt vis navigera genom en OPC-servers *addressrymd*. Med denna tjänst kan man navigera sig fram till värden man till exempel vill prenumerera på. Prenumerationer och andra tjänster använder sig av en nods identifierare när den utför operationer med eller på noden. Denna identifierare säger inte mycket för en vanlig användare utan istället brukar man vilja använda ett mer beskrivande namn. Varje nod har också ett så kallat bläddringsnamn(*BrowseName*), vilket är ett mer beskrivande namn för en nod. Eftersom en OPC-servers *addressrymd* följer en hierarkisk modell representeras bläddringen bäst i form av trädstruktur. Hur bläddringen i den utvecklade mjukvaran set ut visas i figur 4.2.



Figur 4.2: *Bläddringsfönstret. Används för att navigera i serverns addressrymd.*

För att minska antalet bläddringsförfrågningar till servern och därmed belastningen på servern byggs successivt ett objekt, *Folder*, som representerar trädstrukturen upp. Vid varje händelse i bläddringsfönstret, mer specifikt en dubbelklickning på ett element skickar en bläddringsförfrågan till servern om det är som så att elementet inte redan har blivit bläddrat vid ett tillfälle. Har elementet blivit bläddrat en gång hämtas istället informationen från objektet *Folder*. Hur *Folder*-objektet kan se ut efter ett visst antal händelser illustreras i figur 4.3. Varje *Folder*-objekt har referenser till sina barn som i sin tur är *Folder*-objekt. Detta designmönster kallas *Composite Pattern*[8]. Ett *Folder*-objekts barn är okända tills en bläddringsförfrågan har skickats för vald *Folders* nod. I figur 4.3 är till exempel *Boilers* och *Root* bläddrat men inte *Boiler 1, Boiler 2, MemoryBuffer, Server* eller *Data*.



Figur 4.3: *Hur strukturen av Folder-objekt kan se ut.*

4.3.1 Nod-ID från sökväg med hjälp av *Folder*-objektet

Genom *Folder*-objektet löses även problemet med att från en konfigurationsfil, läsa sökvägar för bläddringsnamn och prenumerera på dem. *Folder*-objektet har nämligen en procedur som tar en sökväg som inparameter och returnerar information om målet, proceduren är beskriven i algoritm 13[2]. Är noderna på vägen till målet inte bläddrade blir dem det och bläddringen sparas, som beskrivit, i *Folder*-objektet.

OPC UA-tjänsten *View* definierar en förfrågan, *TransleBrowsePathToNodeIds*, som översätter en sökväg till ett nod-id. Denna fördefinierade förfrågan har valt att inte användas då eventuella fel hellre hanteras lokalt än att delegera felhanteringen till en OPC server. Genom att bygga ett *Folder*-objekt på detta vis vet man redan innan man skickar en förfrågan till servern om en nod med ett visst bläddringsnamn existerar eller ej.

[2]Säkerhetsåtgärder så som att t.ex. kolla att bläddringsnamnet existerade är medvetet utlämnat från algoritmen för att spara plats.

Algoritm 13

1: **function** FOLDER.NODEIFFROMPATH($ASession : Session$, $Path : String$)
2: $SavedFolder = Self$
3: $Elements$ ←Splitta $Path$ så att varje BrowseName finns i elements.
4: **for all** E in $Elements$ **do**
5: **for all** $CFolder$ in $Self.Children$ **do**
6: **if** $E = CFolder.BrowseName$ **then**
7: Om $CFolder$ inte är bläddrat, bläddra det m.h.a $ASession.Browse$.
8: $Self$ ←$CFolder$
9: Break inner loop.
10: **end if**
11: **end for**
12: **end for**
13: $Result$ ←$Self$
14: $Self$ ←$SavedFolder$
15: **return** $Result.NodeId$
16: **end function**

4.4 Hantering av kontaktförlust eller anslutningsfel

Externa händelser som strömavbrott eller nätverksfel innebär att OPC-Klienten tappar kontakt med OPC-Servern. Denna typen av fel måste meddelas till användaren samt hanteras på korrekt vis. Biblioteket som används för att etablera en *socket* med en server och därefter läsa respektive skriva till *socketen* kallas Indy[3]. Vid ett misslyckat kommunikationsförsök över *socketen* kastar Indy ett undantag som bör hanteras av applikationen.

Säkerhetskanalen samt bufferträden är de enda objekt som anropar någon typ av Indy-procedur eller Indy-funktion. Dessa objekt måste alltså fånga eventuella undantag och meddela användaren angående typ av fel. Varken bufferträden eller säkerhetskanalen bör ha kunskap om GUI komponenten eller användaren, varpå det objektorienterade observatörmönstret anammades. Observatörmönstret finns illustrerad i figur 4.4. *GUI*:t implementerar ett observerbart-gränssnitt, se algoritm 14, och objekten det observerar implementerar ett observatör-gränssnitt, se algoritm 15.

Algoritm 14 Observerbart-gränssnitt

1: $FObserveable = interface$
2: **procedure** ADDOBSERVER($o : FObserver$)
3: **end procedure**
4: **procedure** REMOVEOBSERVER($o : FObserver$)
5: **end procedure**
6: **procedure** NOTIFYOBSERVERS($Code : Int32$)
7: **end procedure**

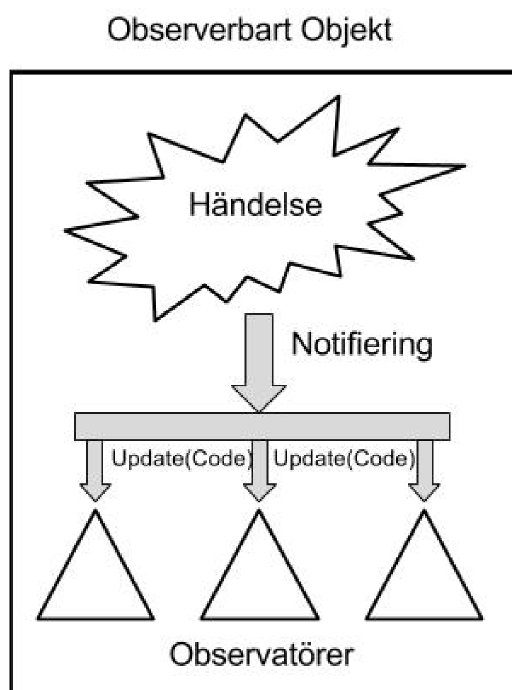
[3]Mer om Indy-projektet kan läsas på dess hemsida, http://www.indyproject.org/index.en.aspx.

Algoritm 15 Observatör-gränssnitt

1: $FObserveable = interface$
2: **procedure** UPDATE($Code : Int32$)
3: **end procedure**

Kärnan av observatörmönstret ligger i att ett observerbart objekt kan hålla en lista över observatörer som manipuleras genom metoder för tillägg och bortagning av objekt. Ett observatör objekt måste implementera uppdateringsmetoden specificerad i observatörs-gränssnittet som därför garanterat kan kallas via en notifieringsmetod. Ingen ytterligare information delas varpå den önskade, minimala delningen av information uppnås.

När ett observerbart objekt fångar ett undantag vidarebefodras denna händelse via en statuskod till alla observatörer. Detta sker via en notifierings-metod som delegeras till en temporär tråd. Metodens uppgift är att anropa och exekvera alla observatörers uppdateringsmetoder, varpå varje objekt kan hantera händelsen på ett unikt lämpligt vis. Händelseförloppet illustreras i figur 4.4.



Figur 4.4: *Observeratörmönstret*

4.5 Agerande vid kontakt- och strömförlust

Kontakt- och strömförlust är två viktiga, grundläggande verkligheter som måste hanteras på lämpligt vis när de inträffar. Kontaktförlust innebär inte terminering av applikationen vilket ger en möjlighet att hantera nuvarande information samt att förlägga den nuvarande instansen till ett säkert läge, vare sig detta är att stänga ner programmet eller att tillåta återanslutning.

Strömförlust innebär p.g.a. sin natur att programmet termineras utan möjlighet för reaktion till händelsen. Dessa skillnader innebär att det finns olika möjligheter att behandla respektive problematik. I denna implementation har dock behandlingen principiellt hållits uniform.

4.5.1 Kontaktförlust

I implementationen är huvudfönstret ensam observatör till buffertråden och säkerhetskanelen, och därmed bestämmer implementeringen av detta objekts uppdateringsfunktion hur kontaktförlust hanteras. Om servern inte kan nås finns det många olika tillvägagångssätt för att hantera detta. Universellt kan det dock anses lämpligt att ej terminera applikationen, varpå återanslutning kan erbjudas. Återanslutningen kan vara en automatiskt process som kontinuerligt försöker ansluta, eller vara en manuell möjlighet för användaren att exekvera.

En klient kan potentiellt ha flera olika pågående kontakter mot servern när kontaktförlust inträffar. Utöver en etablerad säkerhetskanal och session kan klienten även ha pågående prenumerationer samt andra typer av kontinuerligt meddelandeutbyte. Det kan vara önskvärt att inte förlora dessa pågående kommunikationer, och det ges stöd av OPC UA standarden för återetablering av kontakt. En problematik med detta är dock att det meddelande som behöver skickas till servern för att återuppta en tidigare session kräver att de aktuella sekvensnumret och andra variabler finns tillgängliga för att följa korrekt kommunikationsprotokoll. Detta skulle minimalt kräva att nuvarande sekvensnummer och övriga viktiga variabler sparas ner för användning vid återanslutning. Att skriva dessa uppgifter för lagring är dock ej nödvändigtvis säkert, vilket ytterligare tas upp i 4.5.3.

4.5.2 Strömförlust

Som nämnts erbjuder inte denna feltyp möjligheten att hantera felet, utan applikationen dör oväntat under fri del av exekveringen. Därmed finns ingen möjlighet att spara ner nödvändig information för återetablering av kontakt med servern baserad på sessionen vid tiden för strömförlust.

4.5.3 Agerande

Då industriellt användande av applikationen innebär kontinuerlig körning av samma prenumerationer kan en konfigurationsfil skapas och lagras för senare läsning, och automatisk etablering av kontakt med server med valda prenumerationer aktiva. Denna fil kan antingen skrivas för hand eller skriva ut av applikationen. Skrivning av applikationen till hårddisken är principiellt inte önskvärt då det är en av de mest tidskrävande operationer en dator kan utföra. Med detta i åtanke sparas nuvarande information inte ner vid kontaktförlust.

Användaren har möjligheten att spara ner en konfigurationsfil för snabb etablering av en tidigare definierad prenumerationsuppsättning. Detta är en manuell process varpå hänsyn till tiden operationen tar. Ett undantag är att applikationen sparar nödvändig information om servern den senast var i kontakt med. Eftersom applikationen sällan kommer byta server i produktion ses detta som acceptabelt.

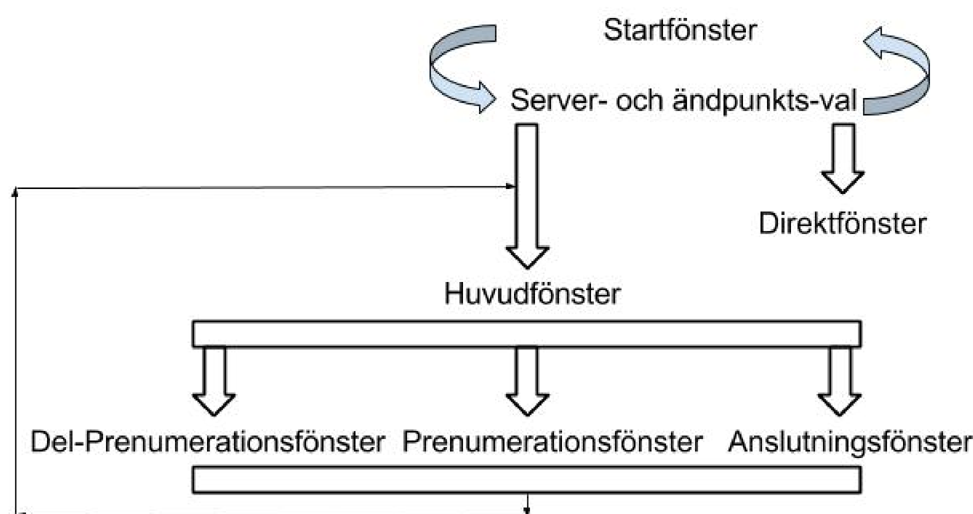
Vid kontaktförlust nollställs programmet till huvudfönstret efter rensning av nuvarande kontaktuppgifter. Utöver detta är hanteringen uniform mellan kontaktförlust och strömförlust. Konfigurationsfiler finns tillgängliga för att snabbt etablera en ny kontakt mot servern, men ingen information om tidigare pågående sessioner sparas.

5 Resultat

I detta kapitel återfinns först en beskrivning av applikationen som utvecklats. Därefter följer en återkoppling till kravspecifikationen med den utvecklade applikationen som utgångspunkt. Källkoden för applikationen beskriven i denna sektion finns tillgänglig på *GitHub*[1].

5.1 Klienten

Klienten är uppbyggd av ett antal grafiska fönster ihopkopplade via knappar som överlåter kontroll från ett fönster till nästa. Applikationens flödesdiagram visualiseras i figur 5.1.

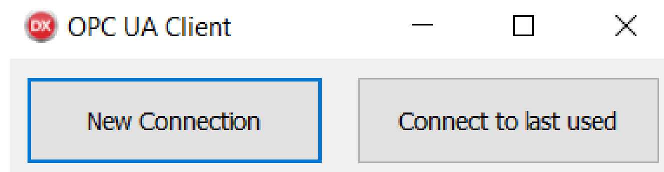


Figur 5.1: *Applikationens flödesdiagram.*

5.1.1 Startfönstret

Startfönstret ger valet att antingen starta en ny anslutning mot en server där värden kan granskas och prenumereras på eller att ansluta mot en tidigare server med hjälp av en konfigurationsfil, startfönstret visas i figur 5.2. Anslutning via konfigurationsfil är det primära användarläget då det är troligt att samma värden lär kontinuerligt önskas bevakas vid industriell användning. Det första läget är menat för en första anslutning och för att undersöka en server och dess innehåll. Val av ny anslutning ger kontroll över till fönstret för val av server- och ändpunkts(5.1.2). Val av anslutning via konfigurationsfil ger kontroll över till direktfönstret(5.1.7).

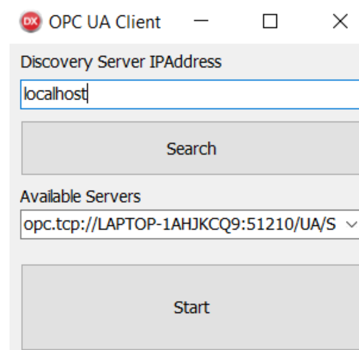
[1]https://github.com/vdahlberg/LMTX38



Figur 5.2: *Det första fönstret som visas när applikationen startar.*

5.1.2 Server- och ändpunkts-val

Detta fönster hanterar tjänsten för att upptäcka OPC UA servrar på en *IP*. Användaren skriver in en IP- Address till en *discovery-server*, varpå en efterfrågan skickas till denna angående vilka dataservrar som finns tillgängliga att ansluta till. Valet av dataservrar presenteras i en rullgardinsmeny, och anslutning mot den valda skapar en säker kanal. Vid anslutning ger fönstret kontroll till huvudfönstret (5.1.3). Om fönstret stängs återgår kontroll till startfönstret (5.1.1). Fönstret visas i figur 5.3.



Figur 5.3: *Server- och ändpunkts-val för anslutning.*

5.1.3 Huvudfönster

Huvudfönstret är det primära användargränsnittet för att utföra operationer gentemot en server. Huvudfönstret visas i figur 5.4. Det innefattar följande komponenter och vyer:

1. *Anslutningsval:* Rullgardinslista med alla tillgängliga anslutningsmöjligheter för att skapa en session ovanpå den säkra kanalen.

2. *Anslutningsknapp:* Knapp för att ansluta. Ger kontrollen till 5.1.4.

3. *Sessionsnamn:* Textfält för val av sessionsnamn.

4. *Bläddringsfönster:* Fönster för visuell representation av en servers noder och data i en trädstruktur.

5. *Värdesfönster:* Presentation av tillgängliga värden på den aktuella nivån i navigeringsfönstret. Dessa utgör möjliga datakällor att prenumerera på.

6. *Lägg till datakälla i prenumeration:* Ger kontrollen till fönstret beskriven i (5.1.5) för parameterval till vald datakälla.

7. *Värdeinformationsfönster:* Anger ytterligare information angående det valda värdet i värdesfönstret.

8. *Prenumerationsfönster:* Återger de aktuella datakällor som valts att prenumerera på.

9. *Prenumerationsknapp:* Ger kontrollen till prenumerationsfönstret(5.1.6) för parameterval till prenumerationen.

10. *Ta bort datakälla:* Tar bort vald datakälla från prenumerationsfönstret, varpå den inte längre ingår i den potentiella prenumerationen.

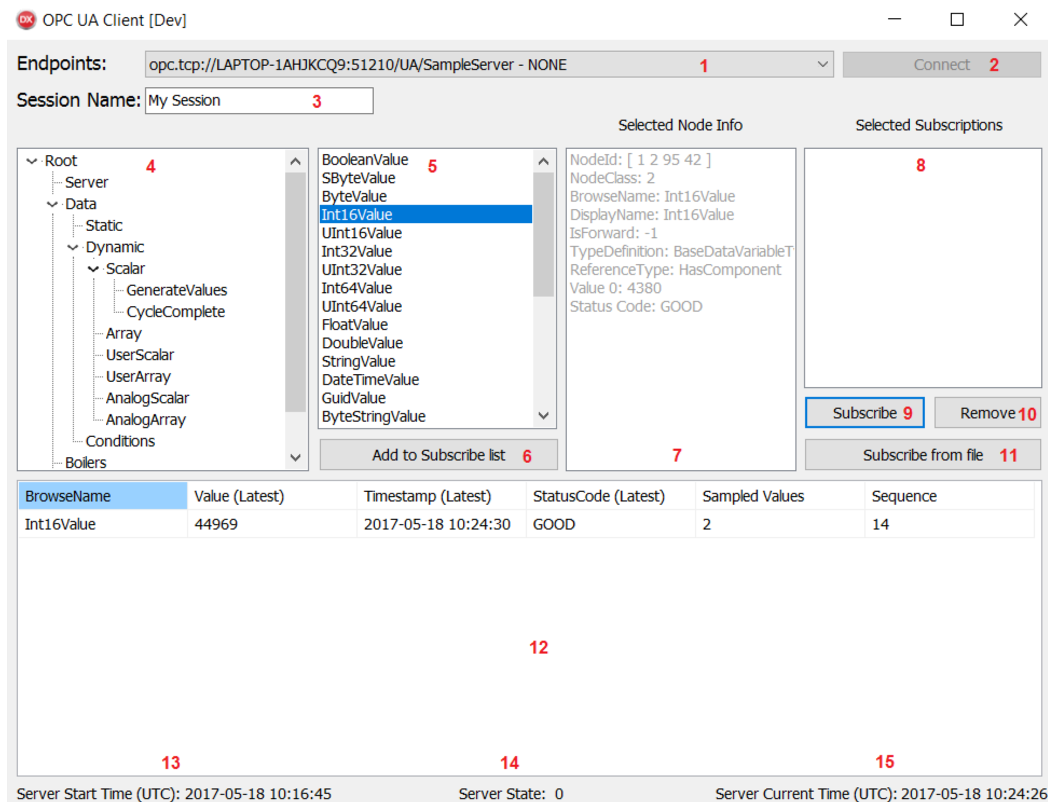
11. *Prenumerera m.ha. konfigurationsfil:* Öppnar upp en bläddringsvy där en konfigurationsfil kan väljas. En prenumeration startas baserad på vad som återges i den valda filen.

12. *Prenumerationsvärdesfönster:* Denna vy detaljerar den information som servern återger för aktiva prenumerationer.

13. *Serverns starttid:* Återger datum och tid då servern startades.

14. *Aktuell serverstatus:* Återger serverns nuvarande status, det vill säga om problem finns på serverns sida för närvarande eller ej.

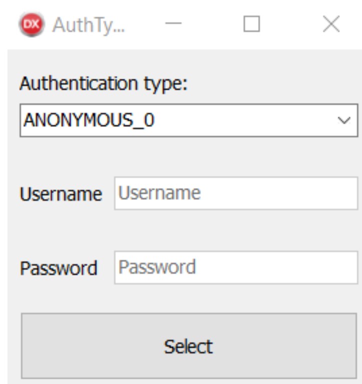
15. *Aktuell servertid:* Återger nuvarande datum och tid vid servern.



Figur 5.4: *Applikationens huvudfönster.*

5.1.4 Anslutningsfönster

Användaren kan här välja metod för att verifiera användares identitet. Flera alternativ finns men i denna applikation implementeras ingen verifieringsmetod, det vill säga att användaren är anonym. Detta fönster visas i figur 5.5.



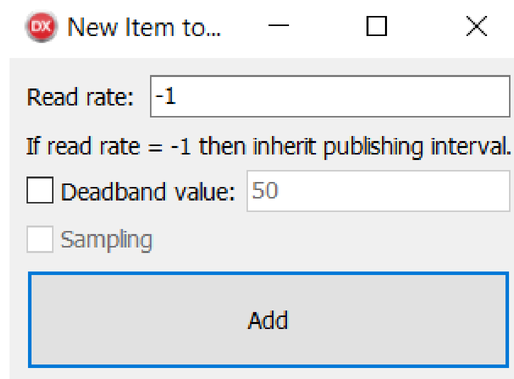
Figur 5.5: *Fönster för att välja autentiseringsmetod till en anslutning.*

5.1.5 Fönster för att lägga till enskilda datakällor i en prenumeration

Här bestäms relevanta parametrar för enskilda datakällor som ingår i en prenumeration. Dessa parametrar är:

- *Läsfrekvensen:* Hur ofta servern ska läsa av ett värde för datakällan.
- *Dödband:* Om dödband önskas, och vilket absolutvärde som ska gälla för det isåfall.
- *Rapporteringsläge:* Om rapporteringsläget skall vara rapportering eller provtagning.

Användning av lägga-till-knappen för in den nya datakällan som del i en prenumeration. Datakällan visualiseras i huvudfönstret och kontroll överges till sagt fönster(5.1.3).Fönstret för att lägga till enskilda datakällor visas i figur 5.6.



Figur 5.6: *Fönster för att lägga till enskilda datakällor i en prenumeration.*

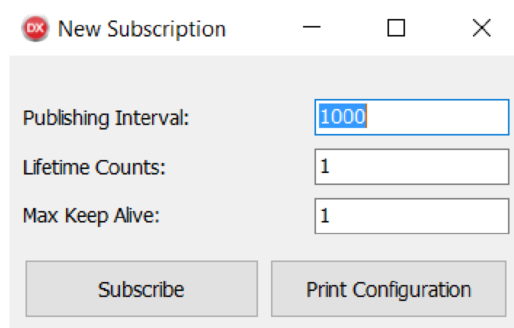
5.1.6 Prenumerationsfönster

I detta fönster anger användaren parametrar som behövs för den aktuella prenumerationen. Parametrarna är:

- *Publiceringsintervall:* Hur ofta servern efterfrågas att publicera värden på alla ingående prenumerationsobjekt. Värdet ges i millisekunder.
- *Livstid / Max Livstid:* Dessa parametrar avgör hur lång tid efter att en prenumeration slutat skicka prenumerationsvärden som prenumerationen skall vara vid liv.

Användning av prenumerationsknappen skickar en förfrågan att starta en prenumeration på alla prenumerationsobjekt som ingår i prenumerationen till servern. De värden som kontinuerligt tas emot enligt publiceringsintervallet visas i huvudfönstret. Kontroll ges tillbaka till huvudfönstret (5.1.3).

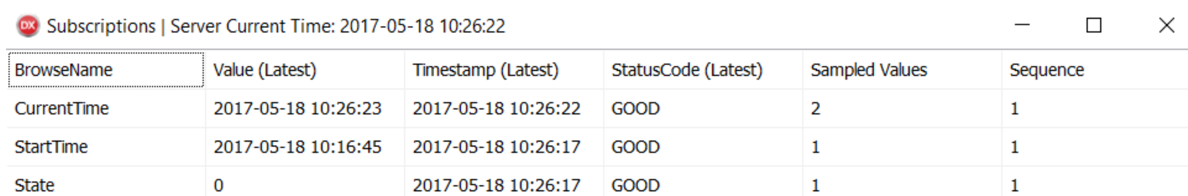
Knappen *'Print Configuration'* (se figur 5.7) skapar en konfigurationsfil med all information som behövs för att återskapa denna prenumeration vid ett senare tillfälle. Användning av denna knapp lämnar inte fönstret. Detta för att tillåta användaren att välja att starta prenumerationen nu, eller att stänga fönstret och återgå till huvudfönstret. Fönstret för att skapa en prenumeration visas i 5.7.



Figur 5.7: *Prenumerationsfönster*

5.1.7 Fönster menat för produktionskörning

Uppstartandet av detta fönster går igenom samma processer som används för att starta upp prenumerationer från en konfigurationsfil via huvudfönstret. Skillnaden är att det sker automatiskt för att åstadkomma en snabb, praktisk användarupplevelse när samma värden kontinuerligt övervakas.



BrowseName	Value (Latest)	Timestamp (Latest)	StatusCode (Latest)	Sampled Values	Sequence
CurrentTime	2017-05-18 10:26:23	2017-05-18 10:26:22	GOOD	2	1
StartTime	2017-05-18 10:16:45	2017-05-18 10:26:17	GOOD	1	1
State	0	2017-05-18 10:26:17	GOOD	1	1

Figur 5.8: *Fönster menat att användas i produktion.*

Själva fönstret är en förstoring av det som syns i huvudfönstret för att visualisera prenumerationer. Stängning av fönstret avslutar applikationen. Detta fönster visas i figur 5.8.

5.2 Återkoppling till kravspecifikationen

Kravspecifikationen som är beskriven i början av rapporten återges här tillsammans med kommentarer som beskriver om kravet är uppfyllt eller ej.

- **Etablera, behålla samt avsluta kontakt med en OPC UA-Server**
  Applikationen etablerar och avslutar kontakt gentemot en server på korrekt vis. Kontakten upprätthålls enligt körtest på två timmar.

- **Möjligheten att upptäcka OPC UA-Servrar och dess ändpunkter**
  OPC UA *discovery*-tjänsterna för att upptäcka servrar och ändpunkterna till vald server är implementerade.

- **Navigera sig i en OPC UA-Servers *adressrymd***
  Navigering genom en servers adressrymd är implementerat. En lokal representation av alla noder som har besökts byggs även upp för att minimera antalet meddelanden som behöver skickas till servern.

- **Läsa samt prenumerera på värden**
  Läsning av värden är implementerad som en automatisk förfrågan när ett datavärde markeras under navigering av adressrymden. Möjligheten att prenumerera på värden är implementerad. *Dödband* för prenumerationer är endast implementerat enligt skillnad i absolutvärde, det vill säga att nya värden jämförs mot det föregående och endast skickas från servern om det skiljer med minst det valda dödbandsvärdet. *Entric* var endast intresserade av denna typ av *dödband*, varpå denna extra avgränsning gjordes.

  Värt att notera är att applikationen är testad för två tillagda prenumerationer och körningstid på två timmar. Vid flera prenumerationer agerar inte applikationen som avsett och diverse fel relaterade till minneskorruption uppstår.

- **Hantering av oförutsedda fel som innebär kontaktförlust, t.ex strömavbrott**
  Strömavbrott hanteras genom möjligheten att skapa konfigurationsfiler för att snabbt ansluta mot en server och skapa önskad prenumeration. Detta erbjuder snabb återkoppling efter avbrottet. Nätverksförlust hanteras genom att avsluta nuvarande kommunikation och att återställa programmet till ett läge där det kan stängas säkert, eller försök till återanslutning göras.

- **Etablera kontakt samt återskapa prenumerationer från en konfigurationsfil**
  Prenumerationer kan skapas från en konfigurationsfil, och denna filen kan skapas av själva applikationen under körning. Ett alternativt körläge för applikationen har utvecklats där kontakt etableras mot servern som applikationen senast var ansluten mot. Användaren väljer en konfigurationsfil för att automatiskt etablera en prenumeration baserad på denna.

- **Skall utvecklas i *Delphi***
  Applikationen är utvecklad i *Delphi* utan inblandning av andra programmeringsspråk och onödiga tredjepartsbibliotek.

Applikationen skulle även preferabelt vara av en sådan nivå att den kunde implementeras i *MRS*. I dagsläget är detta målet ej uppfyllt.

6 Diskussion

När man tar sig an en ny uppgift, där man aldrig löst en liknande uppgift innan, blir tidsplaneringen mer en lös uppskattning av deluppgifternas tidsmässiga krav. Det är även enkelt att underskatta tiden det tar att lära sig ett nytt programmeringsspråk så pass väl att man obehindrat kan lösa ett problem orelaterat till programmering. Detta har bidragit till att applikationen inte är så utförligt testad som kunde önskas för att kunna implementera den i *MRS*. Den undermåliga testningen medför även att applikationen inte är fri från *buggar*.

En stor del av OPC UA-specifikationerna definierar OPC UA:s *addressrymd* samt statiska noder. I projektet las tid på att definiera alla dessa standard-noder för det fanns anledning att tro att detta skulle användas. Eftersom kommunikation mot servern inte hade utvecklats ännu kunde implementeringen av noderna inte heller testas. Det visades sig att denna kod var onödig och endast komplicerade implementationen. Hade fokus istället först varit på att implementera en kommunikationslösning hade det insetts att implementationen av alla standard-noder för OPC UA:s *addressrymd* var onödig. Det ska också noteras att hade dokumentationen lästs mer noggrant och varit bättre förstådd innan utvecklingen började hade den inte inletts med att utveckla *addressrymden*.

I en applikation som kräver multitrådning är noggrann planering av kritiska regioner och varje tråds ansvarsområde av största vikt. Att i efterhand finna alla potentiella regioner där trådar kan tänkas bearbeta samma data är betydligt svårare. Eftersom denna typ av fel inte är kompileringsfel, utan logiska fel, är det även mycket svårare att identifiera felkällan.

Delphi gör det väldigt enkelt att lägga till och hantera GUI-komponenter i en applikation. Det krävs dock att man testar applikationen ur en användares perspektiv och inte en utvecklares perspektiv. Det är enkelt som utvecklare att inte identifiera att en viss ordningsföljd av händelser kan leda till att applikationen befinner sig i ett oavsiktligt stadie, exempelvis i någon typ av låsning. Detta på grund av att just denna ordningsföljden inte är realistisk för en utvecklare, men mycket väl kan vara det för en användare.

Applikationen bör inte implementeras i *MRS* i dagsläget. Detta på grund av felen som uppstår när en användare skapar och kör flera prenumerationer. Det är ännu inte garanterat att dessa fel inte potentiellt skulle kunna inträffa även med en enda aktiv prenumeration om den lämnas aktiv under en längre tid än vad som hittills har testats. Innan applikationen implementeras i *MRS* måste antingen koden som orsakar dessa fel identifieras och åtgärdas, eller så måste det påvisas att fel inte uppstår när endast en prenumeration är aktiv. I det senare fallet bör applikationen rimligtvis inte tillåta en användare att skapa multipla prenumerationer.

6.1 Hållbar utveckling

Projektet förhåller sig till hållbar utveckling på ett påtagligt sätt då det är menat som en ytterligare kommunikationslösning till *Entrics* miljöredovisningssystem. Utsläpp från industrin som övervakas av *MRS* är högst relevanta för både den lokala och globala miljön, varpå projektet indirekt bidrar till att gällande miljölagar följs.

Referenser

[1] *Naturvårdsverkets föreskrifter om mätutrustning för bestämmande av miljöavgift på utsläpp av kväveoxider vid energiproduktion.* NFS2016:13. Naturvårdsverkets Författningssamling. 2016.

[2] *Förordning om förbränning av avfall.* SFS13:253. Svensk Författningssamling. 2013.

[3] *Förordning om stora förbränningsanläggningar.* SFS13:252. Svensk Författningssamling. 2013.

[4] *Delphi Overview.* Embarcadero. URL: `https://www.embarcadero.com/products/delphi` (hämtad 2017-05-23).

[5] *What is OPC?* OPC Foundation. URL: `https://opcfoundation.org/about/what-is-opc/` (hämtad 2017-05-11).

[6] *What is OPC and OPC UA?* Prosys OPC. URL: `https://www.prosysopc.com/opc/` (hämtad 2017-05-15).

[7] *Loopback Capture Setup.* Wireshark. URL: `https://wiki.wireshark.org/CaptureSetup/Loopback` (hämtad 2017-05-23).

[8] *Composite Design Pattern.* sourcemaking. URL: `https://sourcemaking.com/design_patterns/composite` (hämtad 2017-06-15).