Datavetenskap

Linnea Hjalmarsson
Johan Kärnell

Systemövervakningstjänst

System monitoring service

Datavetenskap
C -uppsats

Datum:	2011-06-09
Handledare:	Katarina Asplund
Examinator:	Donald F. Ross
Löpnummer:	C2011:02

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

Systemövervakningstjänst

Johan Kärnell, Linnea Hjalmarsson

Signatursida

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen respektive högskoleingenjörsexamen i datavetenskap. Allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Johan Kärnell

Linnea Hjalmarsson

Godkänd, 2011-06-09

Handledare: Katarina Asplund

Examinator: Donald F. Ross

Sammanfattning

Övervakningstjänster används inom industrin för att bland annat övervaka hälsan hos servers och brandväggar. Ninetechs driftcentral har länge haft ett behov av att skaffa sig en bättre övervakningstjänst då deras nuvarande övervakningssystem inte ger en snabb överblick av hälsotillståndet hos de källor den övervakar. Den här rapporten beskriver hur vi skapade en övervakningstjänst som uppfyller dessa krav. Övervakningstjänsten lagrar inrapporterad data från olika system samt distribuerar densamma till intresserade klienter. Exempel på klienter kan vara PC, Android eller Iphone. Övervakningstjänsten ser också till att radera inaktuell data för att kunna presentera den eftersökta ögonblicksbilden.

Abstract

Monitoring services are used in industry to monitor the health of servers and firewalls. Ninetech's service desk has long had a need to acquire a better monitoring service since their current services do not give a quick overview of the monitored system's health. This report describes how we created a monitoring service that meets these demands. The monitoring service stores reported data from various systems and distributes it to interested clients such as PC, Android or IPhone. The monitoring service also removes outdated data in order to present the requested snapshot.

Förord

Detta examensarbete har omfattat 15 högskolepoäng på C-nivå och genomförts inom fakulteten för datavetenskap på Karlstads universitet (KaU).
Uppdragsgivare var företaget Ninetech och vår kontakt samt handledare på företaget har varit Henrik Bäck. Katarina Asplund var vår handledare från Karlstad universitet och vår examinator var Donald F. Ross. Examensarbetet har utförts på Ninetechs kontor i Karlstad.

Tack

Vi vill först och främst tacka Henrik Bäck, vår handledare på Nintech för att han varit hjälpsam och alltid funnits till hands när vi stött på problem. Vi vill också tacka Ninetechs Service Desk för ett intressant examensarbete samt Mattias Berglund och övriga på Ninetech för att vi fått göra vårt examensarbete hos er och den hjälp vi fått. Dessutom vill vi tacka Katarina Asplund för att du hjälpt oss i din roll som handledare från skolan.

Innehållsförteckning

Figurförteckning

Tabellförteckning

1 Inledning

Ninetech är ett konsultbolag med ett brett åtagande som både utvecklar och förvaltar applikationer och tjänster åt sina kunder. För att Ninetech ska kunna ha detta breda åtagande mot sina kunder behöver de kontinuerligt kunna följa upp statusen på de driftsmiljöer där kundernas system körs.

Ninetech har idag en rad olika system för att övervaka sina driftmiljöer. Driftmiljöerna består av hårdvara, mjukvara och nätverk. Dessa övervakningssystem kan ge ut mycket detaljerad information och historik, men det finns ingen möjlighet att få en tydlig och snabb ögonblicksbild över statusen hos de olika systemen i driftmiljön.

Anledningen till att Ninetech är intresserade av detta examensarbete är att de inte anser att nuvarande övervakning är tillräcklig på grund av den tid det tar att lokalisera var ett fel har uppstått och vad som har hänt. Dessutom är det svårt att få en klar uppfattning av hur statusen för hela driftsmiljön är vid ett givet ögonblick.

Examensarbetet har gjorts med Ninetechs interna Service Desk som beställare.

1.1 Syfte & Mål

Syftet med detta examensarbete var att skapa en tjänst som samlar in och lagrar data från olika övervakningssystem och sedan vidarebefordrar den till en uppkopplad klient så att data kan presenteras i ett grafiskt gränssnitt. Exempel på uppkopplade klienter kan vara en Windowsapplikation, en Androidapplikation eller en Iphoneapplikation. Tjänsten skulle göras så modulär som möjligt då det skulle vara enkelt att bygga ut tjänsten och ansluta nya systemkällor.

Målen var indelade i primära och sekundära mål där de primära målen var de som bedömdes rimliga att hinna med under den utsatta tiden (kursens längd) samt de som stod högst upp på önskelistan hos beställaren. De sekundära målen skulle endast genomföras i mån av tid om de primära målen visade sig ta mindre tid än väntat.

1.1.1 Primära mål

De primära målen för projektet var att skapa en plattform till en övervakningstjänst som kan lagra data som hämtas/skickas från övervakade system och som kan distribuera ut lagrad data till klienter.

I första hand skulle övervakningstjänsten kunna ta emot data från en källa och lagra statusinformation om denna i form av responstid och genomströmningshastighet. Dessa data skulle lagras i en databas av typen SQL Server 2008 Express. För att hantera databasen från övervakningstjänsten skulle så kallade lagrade procedurer (se avsnitt 2.3.2) anropas. Övervakningstjänsten skulle vara en Windowstjänst (se avsnitt 2.4.2) som skulle ha i uppgift att sköta all databashantering i form av lagring och hämtning av data. Utöver databashantering skulle övervakningstjänsten kunna kommunicera med anslutande klienter och distribuera ut data.

1.1.2 Sekundära mål

Det fanns flera sekundära mål som beroende på hur mycket tid som återstod när de primära målen är klara skulle behandlas och prioriteras utifrån rimlighet och kundens önskemål. Dessa mål var:

- Göra programmet så stabilt som möjligt, bland annat testa olika tänkbara scenarion som kan uppstå vid körning och felaktiga avslutningar
- Implementera insamlingsmoduler för alla brandväggar
- Implementera inhämtning av felinformation ifrån övervakade system som exempelvis:
    - Typ av fel (ex. varning, kritiskt)
    - Sammanfattande felmeddelande
    - Berörd server
    - Fysisk miljö som berörs
    - Kund som berörs
- Skapa en enkel Windows-klient som visar informationen som samlats in.

1.2 Resultat

Vi har uppnått de primära målen och skapat en övervakningstjänst som kan lagra data med hjälp av inladdade moduler som i sin tur kommunicerar med övervakningssystem och hämtar ut önskad data. Övervakningstjänsten kan också, i enlighet med de primära målen, kommunicera med olika typer av klienter.

Vi har även hunnit med att uppfylla det sekundära kravet om att möjliggöra lagring av felmeddelanden som genererats av övervakade system. Dessutom har vi i så hög utsträckning som möjligt tagit hänsyn till det sekundära kravet om att göra övervakningstjänsten stabil.

I Ninetechs nuvarande övervakningstjänst finns en lista i vilken alla fel från ett av de övervakade systemen presenteras. Problemet med denna lista är att det inte räcker att åtgärda felet för att det ska försvinna ur listan utan det ligger kvar, men ändrar status från ”obehandlad” till ”behandlad”. I vårt program visas endast en ögonblicksbild över de aktuella felen och därmed utesluts s.k. ”icke fel”. Detta innebär att när fel åtgärdats tas de automatiskt bort ur listan.

1.3 Rapportöversikt

I avsnitt 2 introduceras läsaren till bakgrundsfakta som krävs för att förstå resterande delar av rapporten. Avsnitt 3 beskriver den övergripande konstruktionslösningen medan avsnitt 4 går djupare och beskriver de tekniska detaljerna i vår implementering. Avsnitt 5 redovisar det resultat vi uppnått. I avsnitt 6 berättar vi om de slutsatser vi dragit samt vilka erfarenheter vi plockat på oss efter att ha genomfört projektet.

2 Bakgrund

I detta avsnitt beskrivs olika bakgrundsfakta och begrepp som är viktiga för att läsaren ska kunna tillgodoräkna sig innehållet i resterande delar av denna rapport.

I avsnitt 2.1 listar vi de förutsättningar och krav som ställdes på projektet. Avsnitt 2.2 beskriver de verktyg som använts under projektet och i avsnitt 2.3 beskrivs databasrelaterad information. Vidare beskriver avsnitt 2.4 olika Windowskomponenter som använts under projektet medan avsnitt 2.5 förklarar ett antal begrepp som används längre fram i rapporten.

2.1 Förutsättningar och krav

Från Ninetechs sida fanns önskemål om att vi skulle utveckla programmet i språket C# och därför ansågs Visual Studio lämpligast som utvecklingsmiljö. Ninetech rekommenderade oss också att använda delegat för kommunikation mellan källmoduler och övervakningstjänst, att de klasser som sköter databashantering skulle använda sig av designmönstret singleton, att lagrade procedurer skulle användas för att ställa frågor mot databasens tabeller samt att kommunikation mellan klienter och övervakningstjänst skulle ske med WCF.

Ninetech krävde att databasen skulle vara av typen SQL -Server Express 2008. Det var dessutom ett krav att alla bibliotek övervakningstjänsten använder sig av som inte ingår i .NET -ramverket från början skulle vara gratis.

2.2 Verktyg

Under avsnitt 2.2.1 beskrivs utvecklingsmiljön Visual Studio, under avsnitt 2.2.2 förklaras vad .NET ramverket är och under avsnitt 2.2.3 står att läsa om språket C#, som övervakningstjänsten är skriven i. Vidare berättas under avsnitt 2.2.4 om SQL-Server 2008 Express och i avsnitt 2.2.5 står att läsa om SQL-Server 2008 Management Studio Express.

2.2.1 Visual Studio 2010

Visual Studio är en utvecklingsmiljö framtagen av Microsoft för att utveckla allt ifrån konsolapplikationer till grafiska användargränssnitt och webbsidor. Visual Studio stöder utveckling i flera olika språk däribland C#, C, C++, Visual Basic, J# och F#. Vi har skrivit övervakningstjänsten i språket C# (se avsnitt 2.2.3) [1]. Visual Studio 2010 stöder .NET ramverket (se avsnitt 2.2.2) vilket är ett ramverk framtaget av Microsoft [2]. Utöver tidigare nämnda språk kan man även utveckla interaktiva webbsidor med hjälp av HTML, CSS och ASP i Visual Studio 2010.

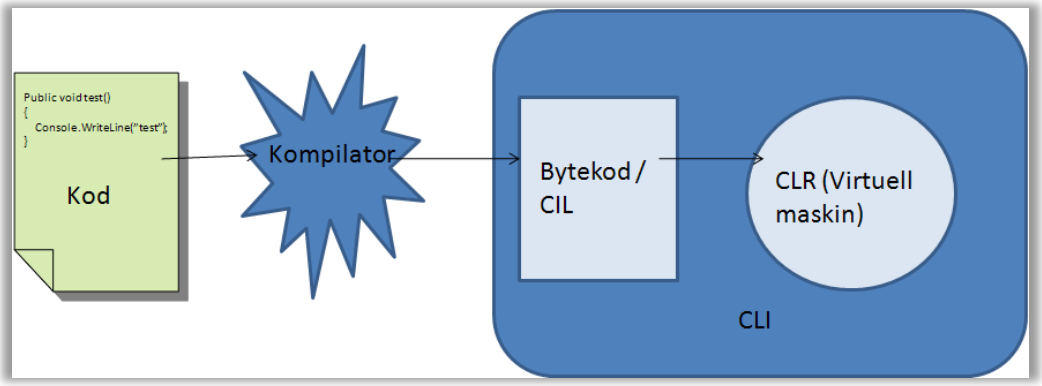
2.2.2 .NET ramverket

.NET-ramverket är ett ramverk för Windows-operativsystem som innehåller ett stort klassbibliotek med lösningar på olika problem [3]. .NET -ramverket stöder ett flertal programmeringsspråk vilka alla är driftskompatibla, dvs. kod skrivet i ett av språken kan anropas av kod skrivet i ett av de andra språken som också stöds av ramverket [2]. Ramverkets klassbibliotek underlättar systemutvecklingen för programmerare då de kan kombinera sin egen kod med den kod som redan finns i ramverkets klassbibliotek. Exempel på färdiga lösningar som ofta används ur klassbiblioteket är kod för att skapa gränssnitt, databashantering och kryptering [2].

2.2.3 C#

C# är ett objektorienterat högnivåspråk som bygger på C++ och det liknar syntaxmässigt språk som C, C++ och Java [4]. Vid kompilering av C# -kod omvandlas koden till en sorts bytekod vid namn CIL (Common Intermediate Language) som sedan en virtuell maskin vid namn CLR, (Common Language Runtime) kör [5]. Bytekod är kompilerad programkod som till skillnad från maskinkod inte är bunden till en specifik datorarkitektur [6] .

En nackdel med C# är att det är långsammare än program skrivna i lågnivåspråk som t.ex. C eller Assembler. En av anledningarna till detta är att C# kompileras precis innan programmet körs. Denna typ av kompilering kallas för JIT (Just In Time). JIT bidrar till att start av programmet tar längre tid då den mer generella CIL- koden måste tolkas av den virtuella maskinen CLR och göras om till maskinkod som kan läsas av datorarkitekturen (se Figur 2.1). CLI (Common Language Infrastructure) är ett samlingsnamn för CIL och CLR [5].



Figur 2.1: CLI

2.2.4 SQL-Server 2008 Express

SQL-server 2008 Express är en gratisversion av SQL-Server:s databashanteringssystem och det är utvecklat av Microsoft [7]. SQL-Server Express 2008 stöder lagrade procedurer (se avsnitt 2.3.2) och är dessutom integrerat i Visual Studio.

2.2.5 SQL-Server 2008 Management Studio Express

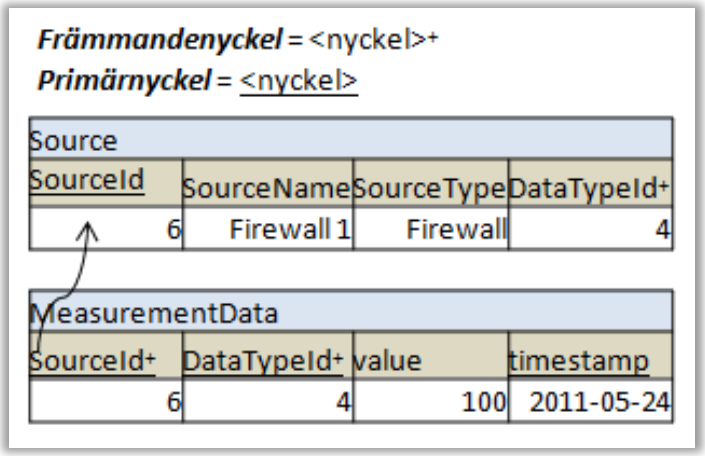
SQL-Server 2008 Management Studio Express (SSMSE) är ett gratis databashanteringsverktyg framtaget av Microsoft som arbetar mot SQL-Server 2008 Express miljön [8]. SSMSE är en integrerad miljö för att ge tillgång till, konfigurera, hantera, administrera och utveckla alla komponenter som finns tillgängliga i SQL-server 2008. SSMSE tillhandahåller såväl grafiska verktyg som editorer att skapa skript med [9]. SSMSE och den kompletta versionen SQL-Server Management Studio (SSMS) skiljer sig bl.a. åt genom att SSMSE inte kan hantera analyseringstjänster, integreringstjänster och rapporteringstjänster [10]. SSMSE stödjer heller inte schemaläggning av administrativa uppgifter genom att använda SQL-Server Agent [10], till skillnad från SSMS.

2.3 Databaskomponenter

Under avsnitt 2.3.1 tas teorin bakom relationsdatabaser upp och under avsnitt 2.3.2 beskrivs lagrade procedurer.

2.3.1 Relationsdatabaser

En relationsdatabas är en samling av tabeller som kopplas till varandra genom att använda gemensamma egenskaper som finns hos tabellerna [11]. Relationer mellan tabeller definieras genom att använda sig av främmandenycklar. En främmandenyckel i en tabell är primärnyckel i tabellen den relaterar till. Ett exempel på detta är relationen som finns mellan tabellerna Source och MeasurementData i vår databas (se Figur 2.2). För att skapa en relation mellan tabellen Source och tabellen MeasurementData används Source primärnyckel som främmandenyckel i MeasurementData. Främmandenycklar kan ses som en av de komponenter i databasen som gör databasen till en relationsdatabas.



Figur 2.2: Relation mellan tabeller

2.3.2 Lagrade procedurer

Lagrade procedurer erbjuder ett sätt att återanvända SQL -kod. Om utvecklare märker att samma SQL -fråga skrivs gång på gång är det aktuellt att göra SQL-frågan till en lagrad procedur istället. För att exekvera SQL -koden i en lagrad procedur räcker det att anropa proceduren. Det är även möjligt att skicka in parametrar till procedurer [12].

Lagrade procedurer minskar också datatrafiken i nätverket då de lagras hos databasen och endast funktionsanrop till en procedur behöver sändas istället för att skicka en hel SQL -fråga. Även den generella prestandan påverkas positivt av de lagrade procedurerna då SQL -frågor

inte behöver kompileras varje gång de ska exekveras. Det räcker att SQL -servern kompilerar en procedur första gången den exekveras vilket minskar belastningen på servern [13].

Ännu en aspekt som förbättras genom att använda lagrade procedurer istället för vanliga SQL -anrop från C# -koden är säkerheten. Genom att använda lagrade procedurer behöver inte användare få tillgång till underliggande dataobjekt utan får endast exekveringsrättigheter till procedurerna [13].

2.4 Windowskomponenter

I detta avsnitt beskrivs olika Windowskomponenter som använts under projektet. Dessa komponenter är delegat som finns att läsa om i avsnitt 2.4.1, Windowstjänster som beskrivs i avsnitt 2.4.2 och Windows kommunikationsfundament som beskrivs i avsnitt 2.4.3.

2.4.1 Delegat

Delegat är C#:s motsvarighet till de funktionspekare som finns i t.ex. C och C++ [14]. Ett delegat är en datatyp som refererar till en metod och kan användas precis som en vanlig metod med parametrar och returvärden [15]. Delegaters primära användningsområden är händelsehantering och återanrop vilket innebär att en metod uppmanas att anropa en annan metod [14].

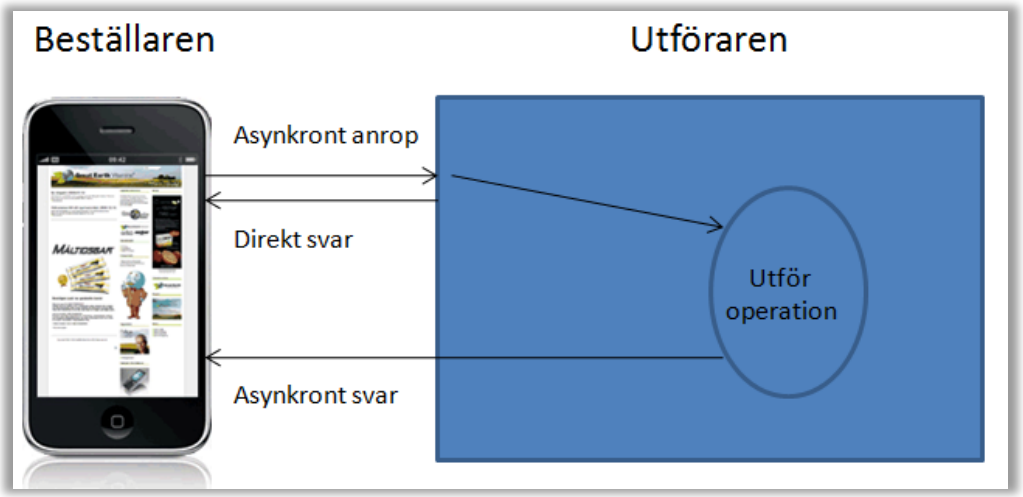
2.4.2 Windowstjänst

En Windowstjänst möjliggör skapandet av långkörande applikationer som körs i egna Windowssessioner. En fördel med dessa program är att de kan startas automatiskt när datorn startas. En Windowstjänst innehåller funktionerna OnStart, OnStop, OnPause och OnContinue. Funktionsnamnen är i och för sig talande men för att tydliggöra så är OnStart den funktion som anropas då tjänsten startas och OnStop -funktionen anropas då man vill att tjänsten skall sluta köra. OnPause används om tjänsten skall pausas under en tid och OnContinue anropas när tjänsten skall startas igen.
En Windowstjänstapplikation har inget användargränssnitt och är speciellt användbart inom servermiljöer där långkörande applikationer används [16].

2.4.3 Windows kommunikationsfundament (WCF)

”WCF är ett ramverk som används för att bygga serviceorienterade applikationer” [17]. Den främsta anledningen till att vi har använt WCF är att detta ramverk implementerar funktionalitet för att göra asynkrona funktionsanrop men också för att vilken klient som helst ska kunna koppla upp sig mot övervakningstjänsten. Det enda som krävs av klienterna är att de följer det kontrakt som WCF angivit.

Figur 2.3 visar hur ett asynkront anrop enligt fråga/svara mönstret ser ut, vilket är det vanligaste mönstret för asynkrona funktionsanrop. Iphoneapplikationen som syns i figuren skickar ett asynkront anrop och får ett direkt svar tillbaka som indikerar att anropet kommit fram och Iphoneapplikationen behöver därmed inte stå och vänta på det slutgiltiga svaret utan kan fortsätta med annat. Anropet i sig triggar igång en operation hos utföraren vars svar returneras till Iphoneklienten när operationen är klar. [18]



Figur 2.3: Asynkront anrop (fråga/svara mönstret)

Ett annat mönster är envägsmeddelandemönstret där en beställare skickar ett anrop men inte förväntar sig svar tillbaka innan den fortsätter med andra processer. Det finns även ett tvåvägsmönster där två ändpunkter utbyter data likt ett snabbmeddelandeprogram [17].

2.5 Begrepp

Under detta avsnitt står att läsa kortfattat om olika begrepp som återkommer längre fram i rapporten. I avsnitt 2.5.1 beskrivs DLL -filer och i avsnitt 2.5.2 står att läsa om vad windowshändelseloggar kan innehålla och avslutningsvis i avsnitt 2.5.3 beskrivs singeltonmönstret.

2.5.1 DLL (Dynamic Link Library)-filer

DLL -filer är färdigkompilerade kodavsnitt som kan användas likt ett kodbibliotek. DLL -filen behöver inte kompileras när en ändring sker i en av de andra modulerna. På detta vis separeras funktionalitet i DLL -filen från övriga moduler.

DLL -filer minskar också minnesanvändningen när flera program använder samma funktionalitet samtidigt. Anledningen till detta är att varje anrop av DLL -filen får en egen kopia av DLL -data medan de samtidigt delar på koden inuti DLL -filen [19].

2.5.2 Händelselogg

I Windows finns inbyggda händelseloggar där mjukvara och hårdvara skriver om olika händelser som inträffat [20]. I loggarna anges också vilken typ av information som skrivits i loggen, ex. fel, varning eller information [21].

2.5.3 Singletonmönstret

Singleton är ett designmönster som säkerställer att det finns som mest en instans av en klass och istället tillhandahåller en global accesspunkt till instansen som alla intressenter kan använda sig av [22].

2.6 Kapitelsammanfattning

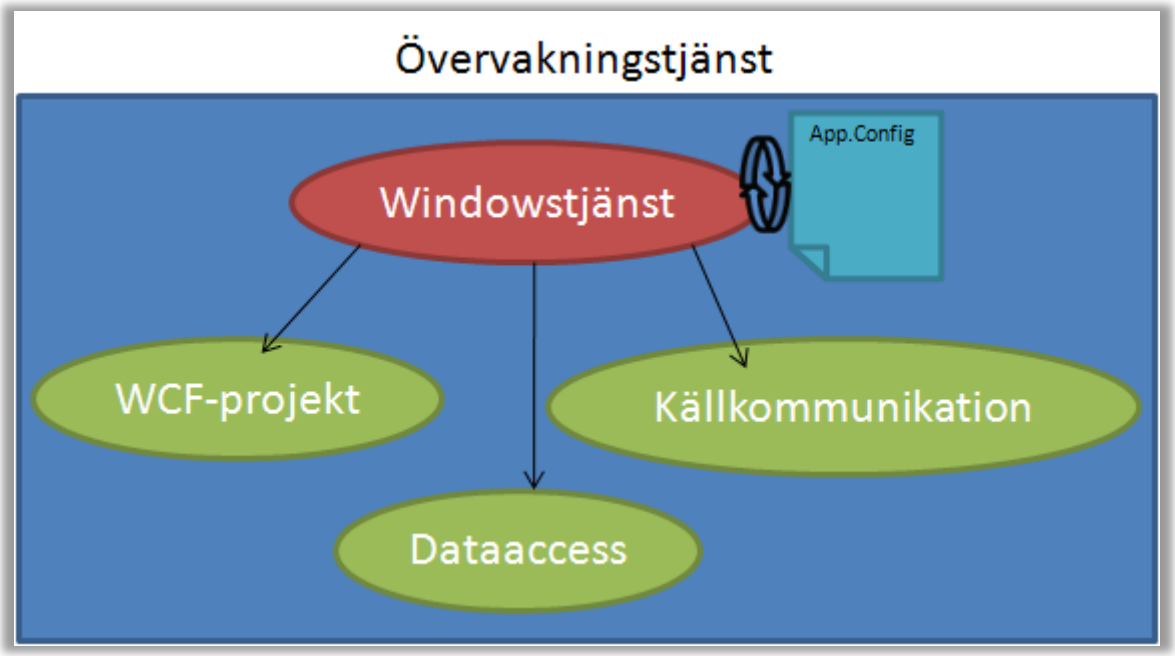
Detta kapitel har handlat om bakgrundsfakta vi anser att läsaren kommer behöva för att tillgodogöra sig fortsatta delar av rapporten. Vi har bland annat beskrivit vad windowstjänster, WCF -projekt och .NET –ramverket är.

3 Beskrivning av konstruktionslösning

Detta avsnitt beskriver konstruktionslösningen för den övervakningstjänst vi skapat och hur den är uppbyggd, det vill säga hur vi löst uppgiften. Avsnitt 3.1 ger en översiktlig beskrivning över övervakningstjänsten och avsnitt 3.2 beskriver dess olika delar. Avsnitt 0 beskriver dataaccessmodulen samt lagrade procedurer medan Avsnitt 3.4 ger en beskrivning över databasstrukturen. Avsnitt 3.5 är en sammanfattning av hela avsnitt 3.

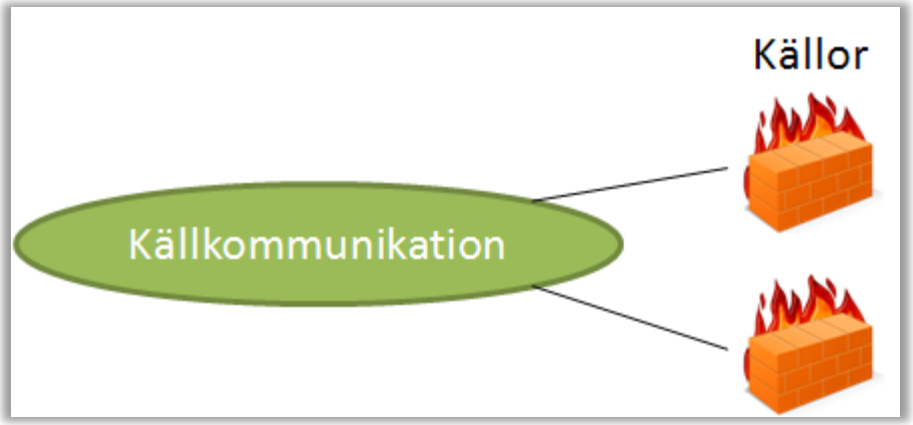
3.1 Översikt

Figur 3.1 visar en övergripande modell av övervakningstjänsten medan Figur 3.2 och Figur 3.3 illustrerar källornas respektive klienternas kommunikation med övervakningstjänsten. Figur 3.4 visar databasens kommunikation med övervakningstjänsten.



Figur 3.1: Övergripande modell av övervakningstjänst

Övervakningstjänsten består av en Windowstjänst som laddar in de källor den ska övervaka från en konfigurationsfil. Därefter öppnas kopplingen mot de inladdade källorna vilket gör att övervakningstjänsten kan börja lagra data som kommer från källorna (se Figur 3.2).

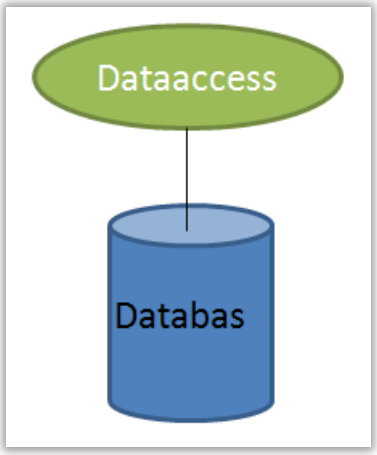


Figur 3.2: Källornas koppling mot övervakningstjänsten

När källor är inladdade och kan börja rapportera data startar windowstjänsten ett WCF -projekt vilket sköter kommunikationen med klienter (se Figur 3.3).

Figur 3.3: Klienternas koppling mot övervakningstjänsten

Därefter kan både klienter och källor kommunicera med övervakningstjänsten. Data som anländer till övervakningstjänsten från olika källor läggs in i en databas genom att använda sig av ett antal klasser som vi valt att se på som en modul kallat dataaccess som i sin tur anropar ett antal lagrade procedurer i databasen (se Figur 3.4).



Figur 3.4: Databasens koppling mot övervakningstjänsten

3.2 Övervakningstjänsten

Detta avsnitt beskriver på ett ingående sätt hur de olika komponenterna i övervakningstjänsten fungerar. I avsnitt 3.2.1 beskrivs windowstjänsten som startar tjänsten och i avsnitt 3.2.2 beskrivs den konfigurationsfil från vilken de olika källmodulerna läses in. Vidare beskriver avsnitt 3.2.3 kommunikation mellan övervakningstjänst och källmoduler medans avsnitt 3.2.4 beskriver kommunikation mellan övervakningstjänsten och de klienter som vill ta del av lagrad data.

3.2.1 Windowstjänst

Navet i vår övervakningstjänst utgörs av en windowstjänstapplikation (se avsnitt 2.4.2). Anledningen till att vi valde att använda oss av en windowstjänst är att övervakningstjänsten ska vara ett långkörande program som kan väljas att startas i samband med att datorn den är installerad på startas.

Windowstjänsten innehåller två funktioner: OnStart och OnStop. Vi har valt bort OnPause och OnContinue då vi inte ser någon användning av att kunna pausa programmet. Funktionen OnStart startar programmet och ser till att källor laddas in i övervakningstjänsten. Detta sker genom att läsa sökvägar m.m. från en konfigurationsfil (se avsnitt 3.2.2). Dessutom

ser OnStart till att klienter och källor kan börja kommunicera med övervakningstjänsten. Klientkommunikation möjliggörs genom att starta igång vårt WCF-projekt (se avsnitt 3.2.4). OnStop har precis som dess namn antyder i uppgift att stoppa programmet. Detta sker genom att anropa stoppfunktioner hos källorna som gör att de slutar sända data. När alla aktiva trådar körts klart stängs tjänsten ner.

3.2.2 Konfigurationsfil

När övervakningstjänsten startas börjar den ladda in de källor som står angivna i konfigurationsfilen (se Figur 3.5).

Figur 3.5: Inläsning från konfigurationsfil

I konfigurationsfilen har vi angett ett mönster som den som lägger till en källa måste följa för att inladdningen skall kunna ske korrekt. Exakt hur detta ser ut står att läsa om i avsnitt 4.2.

Då all inladdning sker från en konfigurationsfil är det lätt att editera källorna, t.ex. ta bort en källa eller ändra hur länge dess mätdata skall sparas i databasen.

3.2.3 Uppkoppling mot källor

För att säkerställa att all inmatning från källorna sker på ett korrekt sätt har vi skapat ett API som de anslutande källorna måste implementera. Figur 3.6 visar två källor som kopplar upp sig mot övervakningstjänsten genom att implementera API:et.

Figur 3.6: API källor

API:et definierar vilka metoder som källor måste implementera. Vi har valt att ha en metod för att initialisera källor samt en metod för att stoppa rapportering av data. Det finns dessutom en tredje metod som i framtiden ska starta rapporteringen, en uppgift som i nuläget sköts av initialiseringsmetoden.

Vi har dessutom valt att använda delegat (se avsnitt 2.4.1) vid anrop från källorna då vi inte vill att källor skall ha kännedom om underliggande funktionalitet och struktur hos övervakningstjänsten. Anledningen till detta är att säkerheten förbättras då källor inte tillåts modifiera värden på egen hand utan måste gå genom delegatfunktioner för att kommunicera med övervakningstjänsten. Det blir också enkelt för de som väljer att programmera en anslutande källa då de inte behöver veta något om övervakningstjänstens inneboende struktur. Dessutom behöver implementeringen inte ändras om det sker någon förändring i övervakningstjänsten så länge som källorna lever upp till kontraktet de tvingas att använda.

För en mer teknisk beskrivning över källkommunikationen se avsnitt 4.3.

3.2.4 Klientkommunikation

Klientkommunikationen med övervakningstjänsten sker genom ett WCF -projekt (se avsnitt 2.4.3). Detta har vi valt eftersom vi vill att klientkommunikation skall ske med asynkrona anrop vilket finns inbyggt i WCF. Ytterligare en fördel med WCF är att det finns en inbyggd testklient där man enkelt kan testa sina asynkrona funktionsanrop.
Vidare så har vi skapat ett API som WCF -projektet implementerar (se Figur 3.7).

Figur 3.7: Modell över koppling från klienter till tjänst

Detta API innehåller följande funktionalitet:

- Lista alla källor som är inlagda i databasen
- Hämta aktiva felmeddelanden från databasen
- Hämta mätdata mellan två inmatade tidpunkter för en specifik källa
- Lista alla källor och vilka typer av data de lagrar i databasen
- Vid given datatyp skall korrekt mätenhet returneras

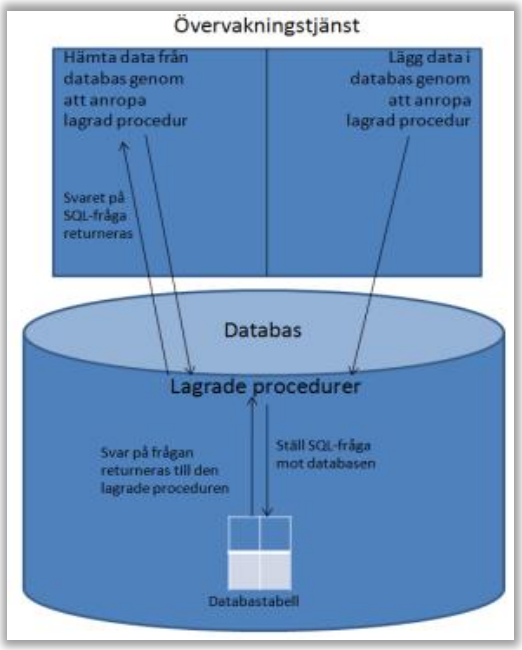
För en mer detaljerad beskrivning om klientkommunikationen se avsnitt 4.4.

3.3 Dataaccess och lagrade procedurer

Vi har ett antal klasser som ingår i det vi kallar dataaccessmodulen i vår tjänst. I dessa klasser finns den kod som anropar de lagrade procedurer (se avsnitt 4.5) som finns i databasen för hämtning och lagring av data. Vid implementering av dessa klasser har vi valt att använda singeltonmönstret (se avsnitt 2.5.3). Anledningen till detta är att vi vill undvika konflikter och dödlägen i databasen då flera klienter eller källor vill läsa eller skriva samtidigt till samma tabell.

Som sagt har vi, för att hantera och modifiera data i databasen, använt lagrade procedurer. Vi har skapat lagrade procedurer för insättning, uthämtning, sökning efter och borttagning av data. Procedurerna anropas från övervakningstjänsten och ställer SQL-frågor mot databasen. Detta är bättre än att låta övervakningstjänsten skicka in SQL-frågor direkt eftersom det belastar nätverket mer och dessutom försämrar säkerheten.

Vi har separerat insättning och hämtning av data från varandra i övervakningstjänsten och skapat en klass vardera som anropar lagrade procedurer, vilka utför operationer mot databasens tabeller (se Figur 3.8). På samma vis har vi separerat sökning efter och borttagning av data från varandra i separata klasser.



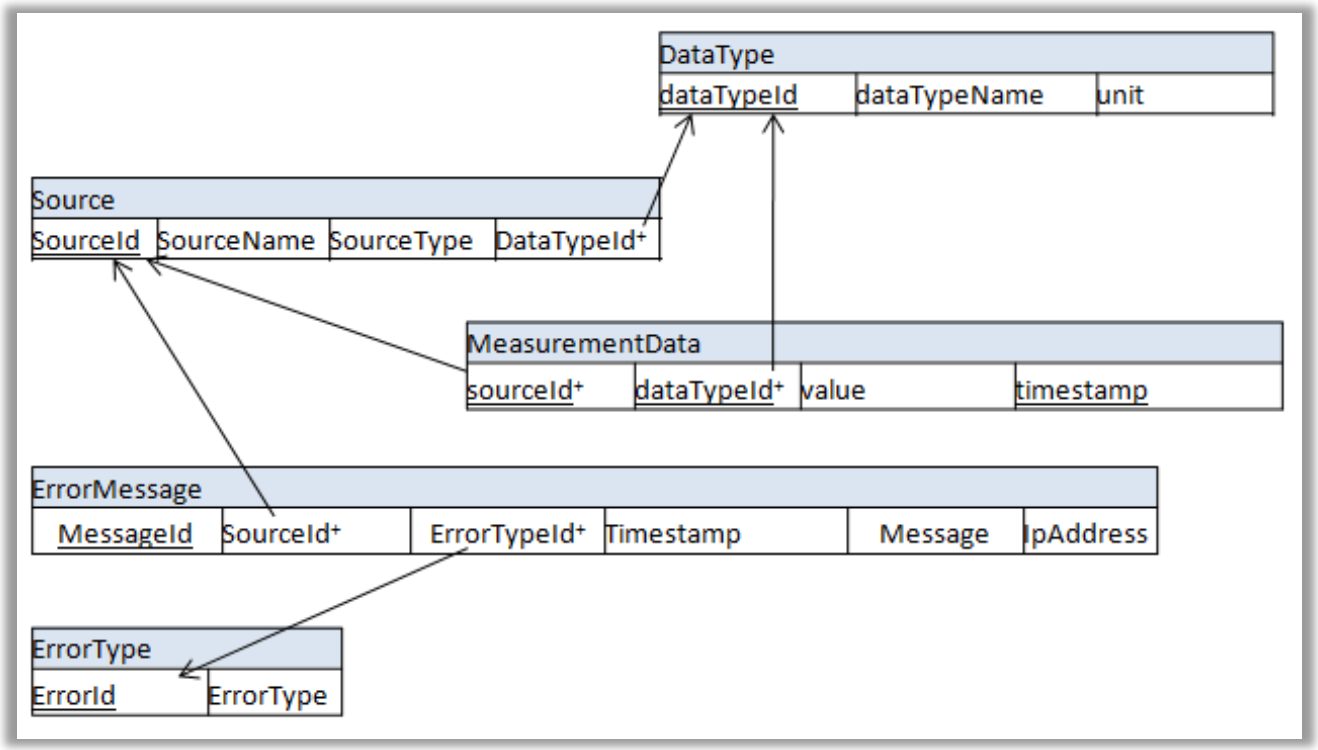
Figur 3.8: Insättning/Hämtning av data

För en mer detaljerad beskrivning över de lagrade procedurer som finns i vår databas se avsnitt 4.5.

3.4 Databasstruktur

Databasen har implementerats i SQL Server 2008 Express och innehåller tabellerna Source, DataType, MeasurementData, ErrorType och ErrorMessage (se Figur 3.9). Figuren visar också på relationerna mellan tabeller i databasen.

De attribut som har ett ”+” efter namnet är främmandenycklar i en relaterad tabell medan understrykning av ett attribut betyder att de är primärnycklar i den tabell de står i. Är flera attribut understrukna betyder det att de tillsammans bildar en sammansatt primärnyckel.



Figur 3.9: Översiktlig databasstruktur

När vi utformat databasen har vi tagit hänsyn till vilken typ av data som skulle lagras. Data som skulle lagras var mätdata och felmeddelanden som genereras av övervakade system. Utifrån detta skapade vi en rad tabeller som krävdes för att kunna identifiera vilken källa, feltyp och datatyp en mätdatapost eller ett felmeddelande har. Källor kan rapportera både mätdata och felmeddelanden. Nedan beskrivs hur de olika tabellerna är uppbyggda.

Tabell 3.1: Source har i uppgift att lagra data om de källor som övervakningstjänsten övervakar och den har attributet SourceId som primärnyckel. Dessutom innehåller Source attributen SourceName, SourceType och DataTypeId. SourceName innehåller precis som namnet antyder källans namn, SourceType innehåller vilken typ källan är av t.ex. om det är en brandvägg och DataTypeId är en främmandenyckel till tabellen DataType vars uppgift är att kartlägga vilken datatyp källan sänder.

Tabellnamn	**Source**			
beskrivning	lagrar data om de källor som övervakningstjänsten övervakar			
Attribut	**SourceId**	**SourceName**	**SourceType**	**DataTypeId**
beskrivning	Källans unika id	Källans namn	Källans typ	Datatypid
datatyp	int	varchar(50)	varchar(50)	int
verifieringsuttryck	>0	!=null	!=null	>0
obligatorisk	ja	ja	nej	ja
Nycklar				
kandidatnyckel	ja			
primärnyckel	ja			
främmandenyckel				ja

Tabell 3.1: Source

Tabell 3.2: DataType har i uppgift att lagra data om olika datatyper och vilken enhet de är av. DataType har attributet DataTypeId som primärnyckel men innehåller också attributen DataTypeName och Unit. DataTypeName innehåller datatypens namn och Unit innehåller vilket enhet datatypen är av t.ex. Kbit/s etc.

Tabellnamn	**DataType**		
beskrivning	lagrar data om olika datatyper och vilken enhet de har		
Attribut	**dataTypeId**	**dataTypeName**	**unit**
beskrivning	Datatypens unik id	Datatypens namn	Enhet datatypen är utav
datatyp	int	varchar(50)	varchar(50)
verifieringsuttryck	>0	!=null	!=null
obligatorisk	ja	ja	ja
Nycklar			
kandidatnyckel	ja		
primärnyckel	ja		
främmandenyckel			

Tabell 3.2: DataType

Tabell 3.3: ErrorType har i uppgift att lagra olika typer av fel t.ex. ”Warning” eller ”Critical” som används för att beskriva hur hög prioritet ett fel har. ErrorType har attributet ErrorTypeId som primärnyckel men också attributet ErrorType som innehåller namnet på feltypen.

Tabellnamn	**ErrorType**	
beskrivning	Lagrar olika typer av fel t.ex. ”Warning” eller ”Critical”	
Attribut	**ErrorId**	**ErrorType**
beskrivning	Unikt feltypsid	Namnet på feltypen, t.ex. "varning" eller "kritisk"
datatyp	int	varchar(50)
verifieringsuttryck	>0	!=null
obligatorisk	ja	ja
Nycklar		
kandidatnyckel	ja	ja
primärnyckel	ja	
främmandenyckel		

Tabell 3.3: ErrorType

Tabell 3.4: MeasurementData används för att lagra data rörande källors mätbara belastning. Den lagrar t.ex. data rörande brandväggars belastning i form av genomströmningshastighet och responstid. MeasurementData har tabellen Source primärnyckel och tabellen DataTypes primärnyckel som främmandenycklar som tillsammans med attributet timestamp utgör dess primärnyckel. Dessa behövs eftersom alla poster i MeasurementData har en källa och en datatyp. Utöver dessa attribut innehåller tabellen även attributen value och timestamp. Value innehåller ett värde av något slag som representeras av en float medan timestamp innehåller vilken tidpunkt posten skickades in från övervakningstjänsten till databasen.

Tabellnamn	**MeasurementData**			
beskrivning	används för att lagra data rörande källors mätbara belastning			
Attribut	**sourceId**	**dataTypeId**	**value**	**Timestamp**
beskrivning	källans unika	datatypens unika id	värde på mätdata	Tidsstämpel inrapportering skedde
datatyp	int	int	double	DateTime(6)
verifieringsuttryck	>0	>0	!=null	> 0000-00-00:000000
obligatorisk	ja	ja	ja	Ja
Nycklar				
kandidatnyckel				
primärnyckel	ja	ja		Ja
främmandenyckel	ja	ja		

Tabell 3.4: MeasurementData

Tabell 3.5: ErrorMessage har i uppgift att lagra felmeddelanden som genererats av de källor som övervakningstjänsten är kopplad mot. Tabellen innehåller ett attribut som är en främmandenyckel till tabellen Source attribut SourceId. Det finns även en främmandenyckel från attributet ErrorTypeId till tabellen ErrorTypes attribut ErrorId. Utöver dessa attribut innehåller ErrorMessage attributen MessageId, timestamp, message och IpAddress. MessageId är en unik identifierare för en rad i tabellen, timestamp är en tidsstämpel för tidpunkten då felet uppstod, message innehåller själva meddelandet och IpAddress innehåller IP-adressen till källan som rapporterade in felet.

Tabellnamn	**ErrorMessage**					
beskrivning	Lagrar felmeddelanden som genererats av de källor som övervakningstjänsten övervakar					
Attribut	**MessageId**	**SourceId**	**ErrorTypeId**	**Timestamp**	**Message**	**IpAddress**
beskrivning	Meddelandets unika id	källans unika id	feltypens unika id	Tid fel uppstod	Felmeddelandet	IP-adress till källa
datatyp	int	int	int	DateTime(6)	varchar(50)	IpAddress
verifieringsuttryck	>0	>0	>0	> 0000-00-00:000000	!=null	!=null
obligatorisk	ja	ja	ja	ja	nej	nej
Nycklar						
kandidatnyckel	ja					
primärnyckel	ja					
främmandenyckel		ja	ja			

Tabell 3.5: ErrorMessage

3.5 Sammanfattning

I detta avsnitt har vi beskrivit vår konstruktion av övervakningstjänsten på en övergripande nivå. Det har stått att läsa om hur tjänsten kommunicerar med klienter genom ett API samt hur övervakningstjänsten möjliggör kommunikation med källor. Det har också i avsnittet stått att läsa om databasens struktur samt hur våra lagrade procedurer fungerar på en överskiktlig nivå.

4 Beskrivning av implementering

Detta avsnitt beskriver implementeringen av övervakningstjänsten i detalj.

Avsnitt 4.1 beskriver arkitekturen hos övervakningstjänsten och avsnitt 4.2 berättar hur programmet startas. Avsnitt 4.3 och 4.4 förklarar hur implementeringen av kommunikation mellan övervakade system och övervakningstjänst respektive klienter och övervakningstjänst fungerar. Avsnitt 4.5 innehåller en förklaring över hur de lagrade procedurerna i databasen fungerar, avsnitt 4.6 beskriver hur implementeringen av singletonmönstret fungerar, avsnitt 4.7 beskriver vilka felscenarion vi tagit hänsyn till och slutligen står i avsnitt 4.8 att läsa en sammanfattning av kapitlet.

4.1 Beskrivning av Figur 4.1, översiktligt UML -diagram

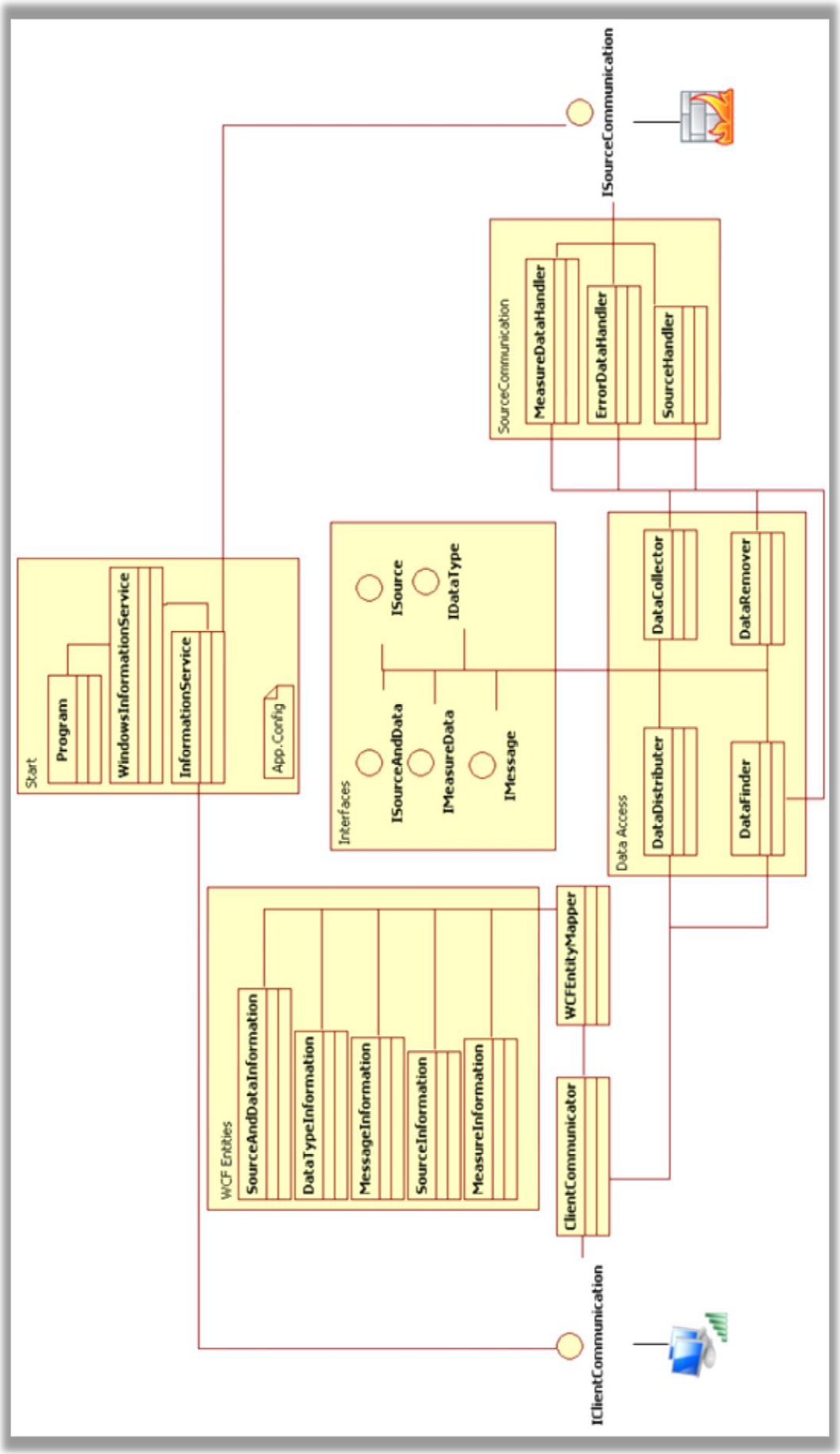
Figur 4.1 visar ett övergripande UML-diagram som beskriver arkitekturen i övervakningstjänsten. Vi har medvetet utelämnat vissa klasser då de endast innehåller stödfunktionalitet för övriga klasser och skulle komplicera bilden över arkitekturen utan att bidra till förståelse.

Klassen Program i start-rutan startar övervakningstjänsten genom att köra igång windowstjänsten WindowsInformationService. Windowstjänsten anropar i sin tur klassen InformationService som ser till att ladda in alla källor övervakningstjänsten ska övervaka och möjliggör för klienter att koppla upp sig mot övervakningstjänsten genom att starta igång WCF- projektet ClientCommunicator.

En källa kommunicerar med övervakningstjänsten genom att implementera ett API vid namn ISourceCommunicator. API:et innehåller funktioner som alla källor måste implementera för att få kommunicera med tjänsten. Beroende på vilken typ av information det är som skickats från källan anropas någon av tre klasser; MeasureDataHandler, ErrorDataHandler eller SourceHandler. Dessa klasser vidarebefordrar data som sänts från källan till klasser som ingår i modulen dataaccess som i sin tur anropar lagrade procedurer i databasen.

En klient kommunicerar med övervakningstjänsten genom WCF- projektet ClientCommunication som implementerar gränssnittet IClientCommunicator som i sin tur innehåller en rad olika funktioner som även klienter måste implementera för att kunna

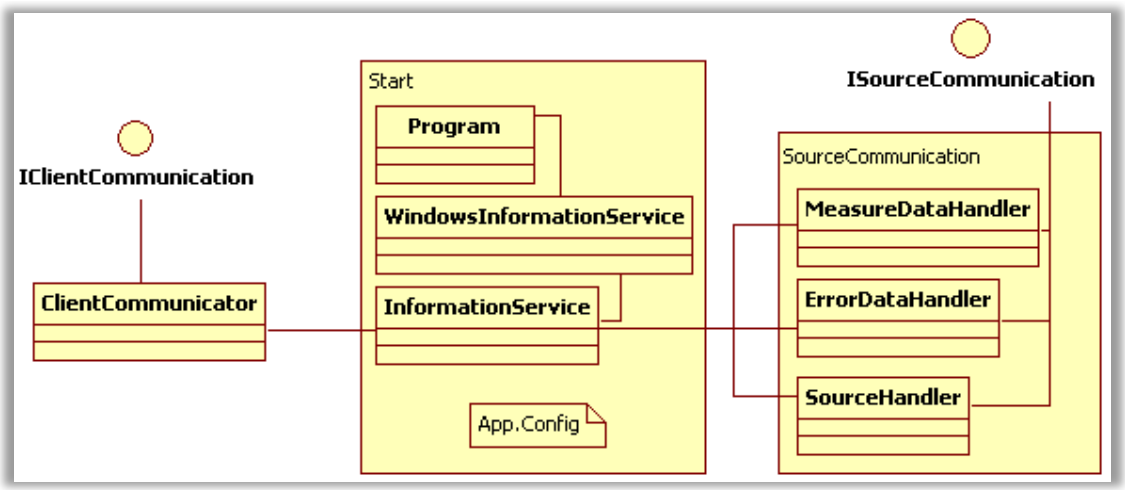
kommunicera med tjänsten genom WCF-projektet. Klienterna ber om data som hämtas av olika klasser som anropar lagrade procedurer i databasen. Dessa anropade klasser ingår i modulen dataaccess.
För att klienten ska kunna ta emot data övervakningstjänsten skickar tillbaka, omvandlas typen av data till en typ som klienten stödjer från interfaceobjekt till vanliga objekt.



Figur 4.1: Översiktligt UML-diagram

4.2 Start av övervakningstjänst

Figur 4.2 visar de klasser som medverkar vid start av tjänsten. Klassen Program innehåller funktionen main som startar upp windowstjänsten, vilken är namngiven till WindowsInformationService. I den klassen finns två funktioner, OnStart och OnStop. Funktionerna OnStart och OnStop anropar funktionerna start respektive stop i klassen InformationService.



Figur 4.2: Start av övervakningstjänst

Start -funktionen i InformationService börjar med att läsa från konfigurationsfilen App.config.

I konfigurationsfilen finns olika inställningar skrivna i xml -kod. Några av inställningarna är adresser till databasen och WCF -projektet men det vi främst vill belysa i rapporten är den kod som berör de olika källorna och som finns under taggen ”sources”.

I Figur 4.3 ses ett utdrag ur konfigurationsfilen som visar hur en källas konfigurationstagg, ’<source>’ ser ut. Det som skrivs i ’<adapter-config>’ -taggen är sådant som är källspecifikt, dvs. det den som skapat den inladdande källan bestämt att den vill ha tillbaka från övervakningstjänsten. I detta fall returneras vilken ip -adress som skall avlyssnas och vilket namn den ska rapportera som.

```
<source assembly="C:\Documents and Settings\ksdlihj\Skrivbord\Ninetech.ServiceMonitoring.dll"
        assembly-type="Ninetech.ServiceMonitoring.MQServiceMonitoring"
        name="MQConverter Prod"
        source-type="Mainframe Queue"
        datatypeid="4">
  <adapter-config>
    <ip-address>192.168.126.82</ip-address>
    <environment-name>PROD</environment-name>
  </adapter-config>
</source>
```

Figur 4.3: Utdrag ur konfigurationsfil

Koden i Figur 4.4 är ett utdrag från funktionen start i klassen InformationService. Den visar hur källor laddas in i övervakningstjänsten.

```
//Load sources
foreach (Configuration.Source source in settings.Sources)
{
    try
    {
        System.Reflection.Assembly sourceAssembly = System.Reflection.Assembly.LoadFile(source.Assembly);
        ISourceCommunication sourceBase = sourceAssembly.CreateInstance(source.AssemblyType) as ISourceCommunication;
        if (sourceBase != null)
        {
            sourceBase.Init(sourceHandler.GetSourceId(source.Name, source.SourceType, source.DataTypeId),
                sourceHandler.GetAdapterConfig(source.AdapterConfig.InnerXml),
                errorDataHandler.OpenError(sourceHandler.GetSourceId(source.Name, source.SourceType, source.DataTypeId)),
                new MeasureDataDelegate(measureDataHandler.AddMeasureData),
                new CreateErrorDelegate(errorDataHandler.CreateError),
                new CloseErrorDelegate(errorDataHandler.CloseError));
            Thread sourceThread = new Thread(new ThreadStart(sourceBase.Start));

            sourceThread.Start();

            activeSources.Add(source.Name + source.DataTypeId.ToString());
            sources.Add(sourceBase);
        }
    }
    catch (Exception ex)
    {
        Logger.LogEvent(ex, "loading sources from config");
    }
}
```

Figur 4.4: Inläsning av källor, utdrag från funktionen start i klassen InformationService

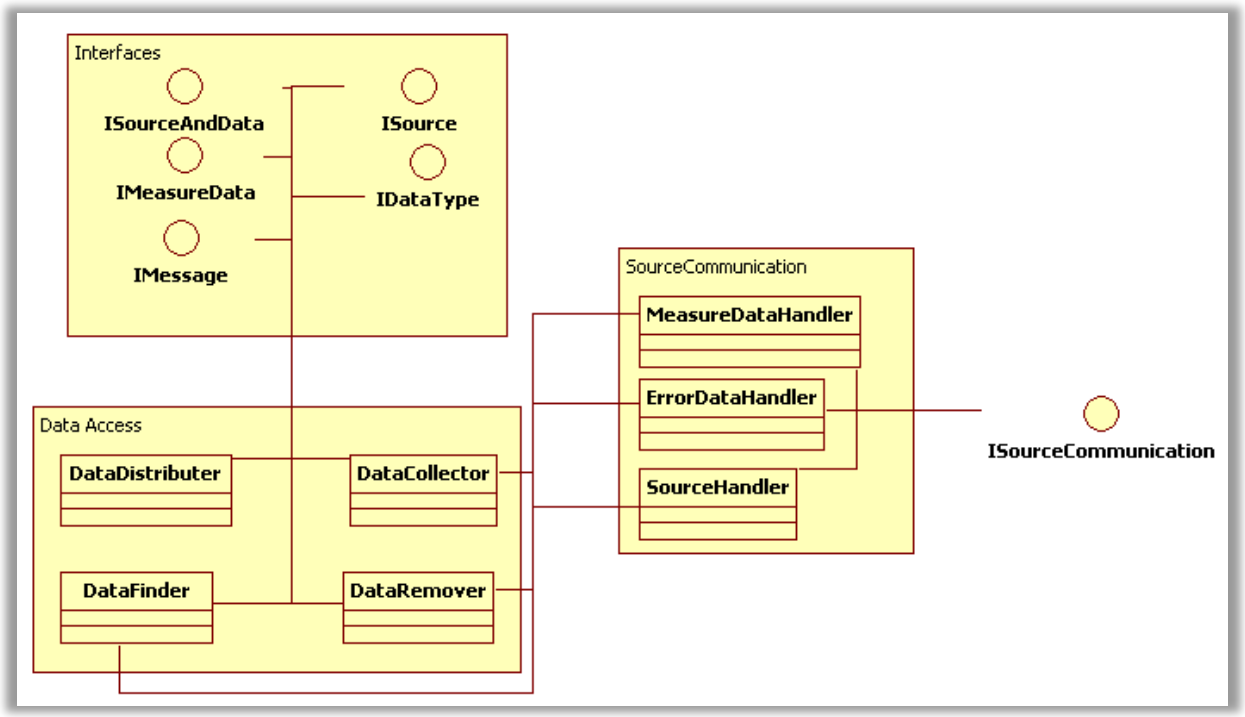
För varje källa som finns angiven i konfigurationsfilen initialiseras id och delegat till de funktioner källan behöver för att kommunicera med övervakningstjänsten. Varje källa körs dessutom i en egen tråd.

Efter att källor lästs in från konfigurationsfilen kontrolleras det att inga inaktuella källor finns i databasen genom att jämföra en lista på de inladdade källorna mot de som redan finns inlagda i databasen. Om en källa inte har laddats in, men finns lagrad i databasen, tas all data som hör till källan bort.

Det sista som görs i startfunktionen hos klassen InformationService är att WCF -projektet ClientCommunicator öppnas så att övervakningstjänsten börjar lyssnar efter klientanrop

4.3 Källkommunikation

Figur 4.5 visar de klasser och gränssnitt som medverkar vid kommunikation mellan källa och övervakningstjänst.



Figur 4.5: Källkommunikation

Tabell 4.1 visar vilka funktioner som återfinns i källAPI:et, kallat ISourceCommunication.

Funktion	Inparametrar	Förklaring
Init()	Id, adapterconfig, öppna fel, MeasureDataDelegate, CreateErrorDelegate, CloseErrorDelegate	Initialiserar källan och sätter dess olika egenskaper. Dvs. delegatmetoder, id m.m.
Start()		
Stop()		Stoppar källan från att sända mer data

Tabell 4.1: Funktioner i ISourceCommunication

Figur 4.6 visar innehållet i ISourceCommunication- gränssnittet vilket är det API källor måste implementera för att kommunicera med övervakningstjänsten.

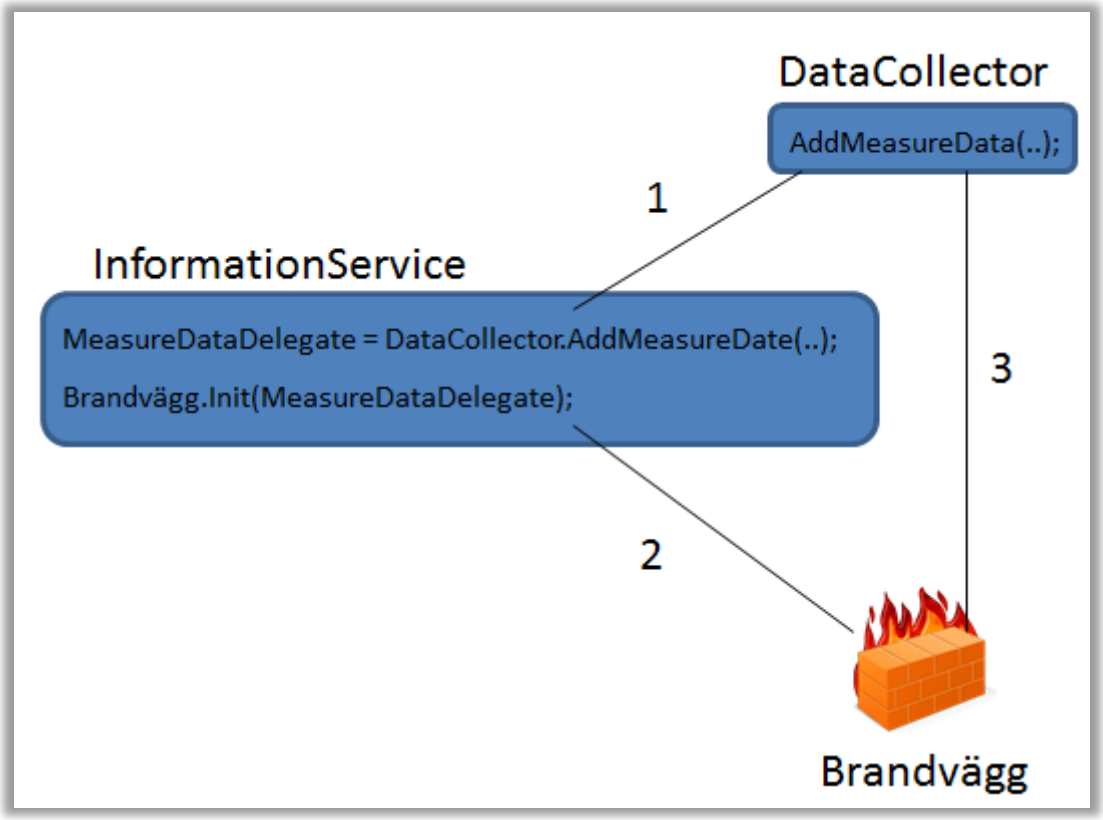
```
public delegate void MeasureDataDelegate(SourceId soruceId, double value,
    DataType dataType, DateTime timeStamp, DateTime removeTime);

public delegate ErrorMsgId CreateErrorDelegate(SourceId sourceId,
    ErrorType errorType, string message, IPAddress ipAddress, DateTime time,
    string keyToken, string environment = "", string customer = "");

public delegate void CloseErrorDelegate(ErrorMsgId errorId);

public interface ISourceCommunication : IDisposable
{
    void Init(SourceId id, string adapterConfig, IList<IErrorMsgIdAndKeyToken> errMsgIdAndKey,
        MeasureDataDelegate measureDelegateFunction, CreateErrorDelegate createDelegateFunction,
        CloseErrorDelegate closeDelegateFunction);
    void Start();
    void Stop();
}
```

Figur 4.6: ISourceCommunication

Som synes i Figur 4.6 skall källor implementera följande tre funktioner: Init, Start, och Stop. Init skall vara uppbyggd på så sätt att den tar in följande data:

- tilldelat id för källan
- en sträng med de adaptervärden den som skapat källan vill returnera
- en lista med aktiva felmeddelande
- delegat till funktion som anropas vid rapportering av mätdata
- delegat till funktion för att rapportera/skapa ett felmeddelande
- delegat till funktion för att stänga/radera ett felmeddelande

För att visa hur delegatfunktionerna som finns i övervakningstjänsten fungerar beskrivs nedan hur delegatmetoden MeasureDataDelegate fungerar (se Figur 4.7).



Figur 4.7: Beskrivning av delegat

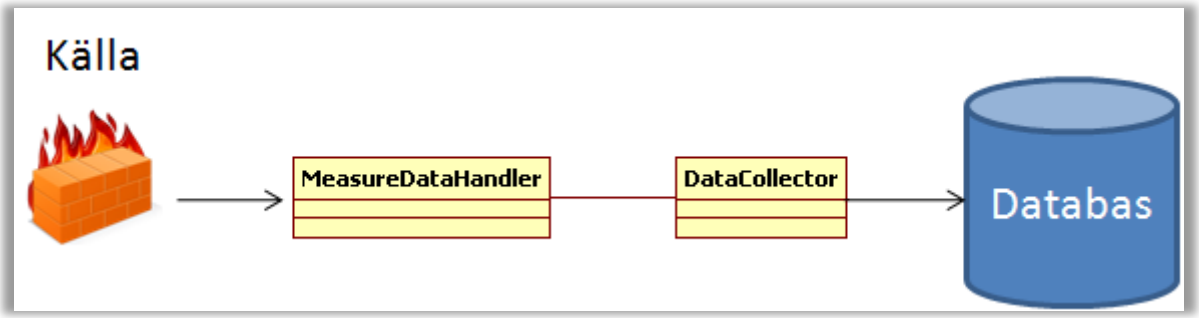
1. Uppgiften för klassen InformationService är att ladda in källor från konfigurationsfil. För att källorna ska kunna kommunicera med övervakningstjänsten finns tre olika delegatmetoder. InformationService tilldelar delegatmetoderna värden. MeasureDataDelegate tilldelas en referens till metoden AddMeasureData i klassen DataCollector.

2. Därefter anropas metoden Init som finns implementerad hos källan, som i detta fall är en brandvägg. Med anropet skickas en rad olika attribut, däribland delegatmetoderna.

3. När brandväggen vill rapportera mätdata till övervakningstjänsten används delegatmetoden MeasureDataDelegate som togs emot från klassen InformationService när metoden Init anropades. Den refererar direkt till metoden AddMeasureData i DataCollector vilket gör att brandväggen kan kommunicera med övervakningstjänsten och rapportera in mätdata.

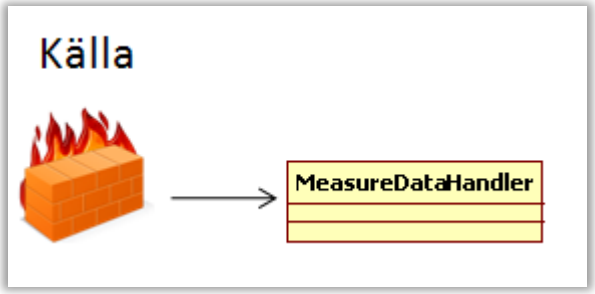
4.3.1 Steg- för- steg- exempel

Figur 4.8 ger en översikt på hur en förfrågan sänds då en brandvägg vill rapportera in mätdata till övervakningstjänsten. För att en källa skall kunna lägga in mätdata i databasen måste källan implementera de funktioner som finns angivna i ISourceCommunication- gränssnittet.



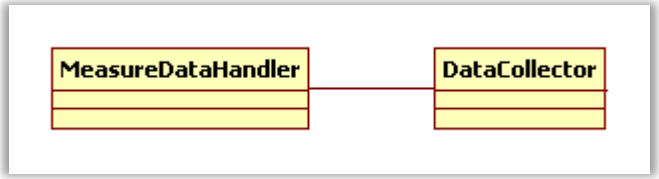
Figur 4.8: Källanrop

1. Figur 4.9 visar en källa som vill skicka in data till övervakningstjänsten. Varje källa tilldelas en rad olika attribut när den laddas in från konfigurationsfil, bl.a. ett delegat som heter MeasureDataDelegate vilken pekar på funktionen AddMeasureData i klassen MeasureDataHandler.



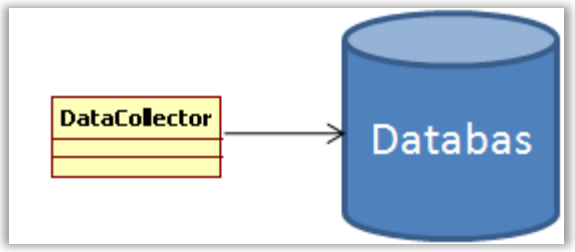
Figur 4.9: Källa anropar delegat som pekar på funktion i MeasureDataHandler

2. Figur 4.10 visar kopplingen mellan MeasureDataHandler och DataCollector. MeasureDataHandler anropar funktionen AddMeasureData i DataCollector- klassen vars uppgift är att utföra operationer mot databasens lagrade procedurer.



Figur 4.10: MeasureDataHandler kopplar till DataCollector

3. Figur 4.11 visar klassen DataCollector och dess koppling mot databasen där den anropar den lagrade proceduren som hanterar insättning av mätdata.

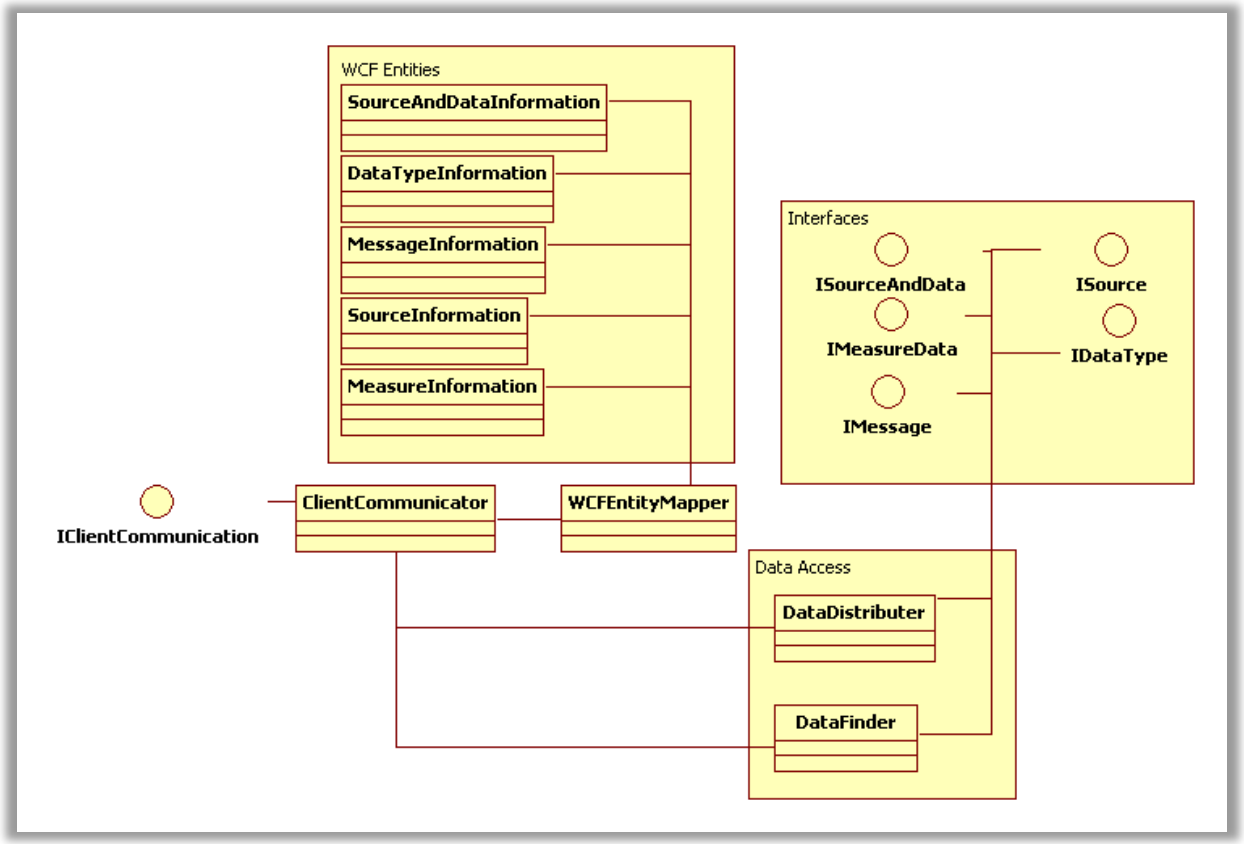


Figur 4.11: DataCollector kopplar till databas

När dessa steg är genomförda har mätdata lagts till i databasen.

4.4 Klientkommunikation

Figur 4.12 visar en översikt av de klasser och gränssnitt som är involverade vid kommunikation mellan klient och övervakningstjänst.



Figur 4.12: Klientkommunikation

Tabell 4.2 visar vilka funktioner som återfinns i klientAPI:et, kallat IClientCommunication och som implementeras av klassen ClientCommunicator, vilket är ett WCF -projekt, som kan anropas av klienter.

Funktion	Inparametrar	Returvärde	Förklaring
GetSources()		List<SourceInformation>	Returnerar ut en lista på alla källor
GetMeasureData()	int, int, datetime, datetime	List<MeasureInformation>	Returnerar ut en lista över mätdata mellan två angivna tidpunkter
GetMessage()		List<MessageInformation>	Returnerar ut en lista över alla felmeddelanden
GetSourcesAndDataInfo()		List<SourceAndDataInformation	Returnerar ut en lista över källor och vilka datatyper de stöder (för mätdata)
GetDataTypeProperties()	int	DataTypeInformation	Returnerar ut namn och enhet för ett specifikt id

Tabell 4.2: Funktioner i klient-API

ClientCommunicator tar emot anrop från klienter som vill ta del av data som övervakningstjänsten lagrar i databasen. ClientCommunicator implementerar gränssnittet IClientCommunication (se Figur 4.13).

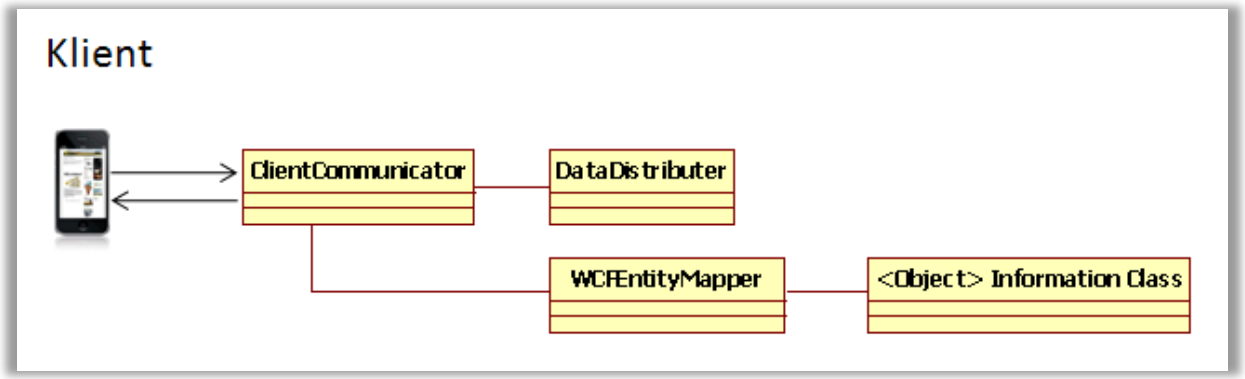
```
public interface IClientCommunication
{
    [OperationContract]
    List<Domain.WCFEntities.SourceInformation> GetSources();

    [OperationContract]
    Domain.WCFEntities.DataTypeInformation GetDataTypeProperties(int dataTypeId);

    [OperationContract]
    List<Domain.WCFEntities.MeasureInformation> GetMeasureData(int sourceId, int dataTypeId, DateTime start, DateTime stop);

    [OperationContract]
    List<Domain.WCFEntities.SourceAndDataInformation> GetSourceAndDataInfo();

    [OperationContract]
    List<Domain.WCFEntities.MessageInformation> GetMessage();

}
```

Figur 4.13: IClientCommunication

Kommunikationen mot klienterna valde vi att implementera som ett WCF -projekt (se avsnitt 2.4.3) eftersom WCF stödjer asynkrona anrop, utan att vi behöver ta hänsyn till det när vi programmerar. Ännu en fördel som WCF gav oss var att det finns en inbyggd testklient som kan användas till att kontrollera att de asynkrona funktionsanropen fungerar korrekt samt att de funktioner som finns ger de förväntade returvärdena.

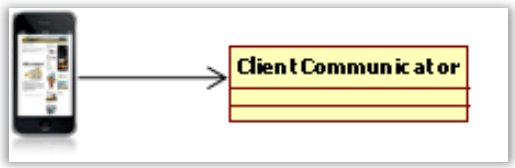
4.4.1 Steg- för- steg- exempel

När en klient ber om data från övervakningstjänsten anropas klassen ClientCommunicator som anropar klassen DataDistributer som i sin tur anropar databasen och returnerar svaret från databasen tillbaka till ClientCommunicator. Varje post i svaret omvandlas till ett objekt genom att anropa klassen WCFEntityMapper. Objekten placeras sedan i en lista som returneras till klienten (se Figur 4.14).



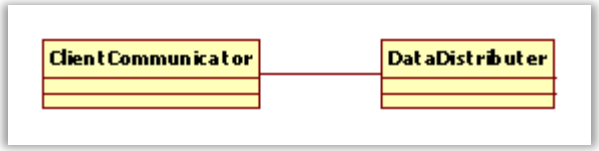
Figur 4.14: Klientanrop

1. Klienten kopplar upp sig mot övervakningstjänsten och anropar klassen ClientCommunicator för att hämta alla de källor som tjänsten övervakar (se Figur 4.15). Detta gör klienten genom att anropa funktionen GetSources i ClientCommunicator.



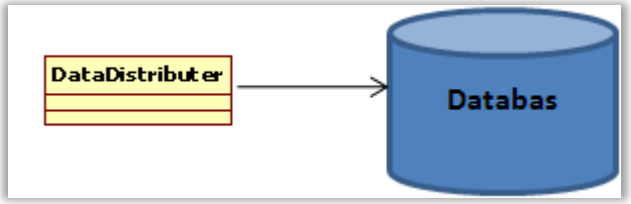
Figur 4.15: Klients koppling mot ClientCommunicator-klassen

2. Klassen ClientCommunicator vidarebefordrar anropet till funktionen GetSources i klassen DataDistributer. DataDistributer har i uppgift att hämta all data som klienten kan tänkas vilja ta del av (se Figur 4.16).



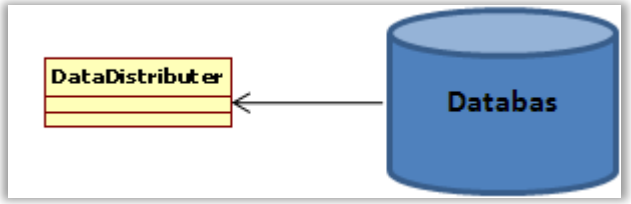
Figur 4.16: ClientCommunicator-klassens koppling mot DataDistributer-klassen

3. Funktionen GetSources i klassen DataDistributer anropar en lagrad procedur i databasen som har i uppgift att hämta alla källor som övervakningstjänsten samlar in data ifrån genom att ställa en SQL-fråga mot databasen (se Figur 4.17).



Figur 4.17: DataDistributer-klassens koppling mot databas

4. Databasen svarar på den lagrade procedurens SQL-fråga och hämtar alla källor ur databasen. Den lagrade proceduren skickar vidare svaret till klassen DataDistributer (se Figur 4.18).

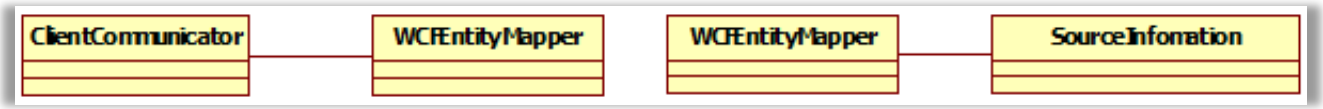


Figur 4.18: Databaskoppling mot DataDistributer-klassen

5. Funktionen GetSources i klassen DataDistributer returnerar svaret till sin anropare (se Figur 4.19).

Figur 4.19: DataDistributer-klassens koppling mot ClientCommunicator-klassen

6. Klassen ClientCommunicator omvandlar listan av objekt den fick returnerat efter anropet till funktionen GetSources i klassen DataDistributer. Objekten omvandlas till instanser av typen SourceInformation genom att anropa funktionen Map i klassen WCFEnitityMapper. Omvandlingen måste göras då det som returneras från DataDistributer är av en typ som klienten inte stödjer. Anledningen till varför inte objekt är av rätt typ från början är för att objekt som hämtas ur databasen skiljer sig från de objekt vi vill ge klienten. Detta är bra om det t.ex. finns känslig eller för klienten onödig information som vi inte vill att denne skall få ta del utav. Map gör om objekt av typen ISource som sänts till funktionen till objekt av typen SourceInformation (se Figur 4.20).



Figur 4.20: Kopplingar

7. Objekt av typen SourceInformation returneras till funktionen GetSources i klassen ClientCommunicator (se Figur 4.21).

Figur 4.21: Koppling WCFEntityMapper mot ClientCommunicator

8. Funktionen GetSources i klassen ClientCommunicator lägger in objekten av typen SourceInformation i en lista som returneras till klienten (se Figur 4.22).

Figur 4.22: Koppling ClientCommunicator mot klient

4.5 Lagrade procedurer

De lagrade procedurerna ligger i databasen och vi har fyra olika typer. I avsnitt 4.5.1 beskrivs sökprocedurer, avsnitt 4.5.2 beskriver insättning, avsnitt 4.5.3 beskriver hämtning och avsnitt 4.5.4 beskriver de procedurer som tar bort data ur databasen.

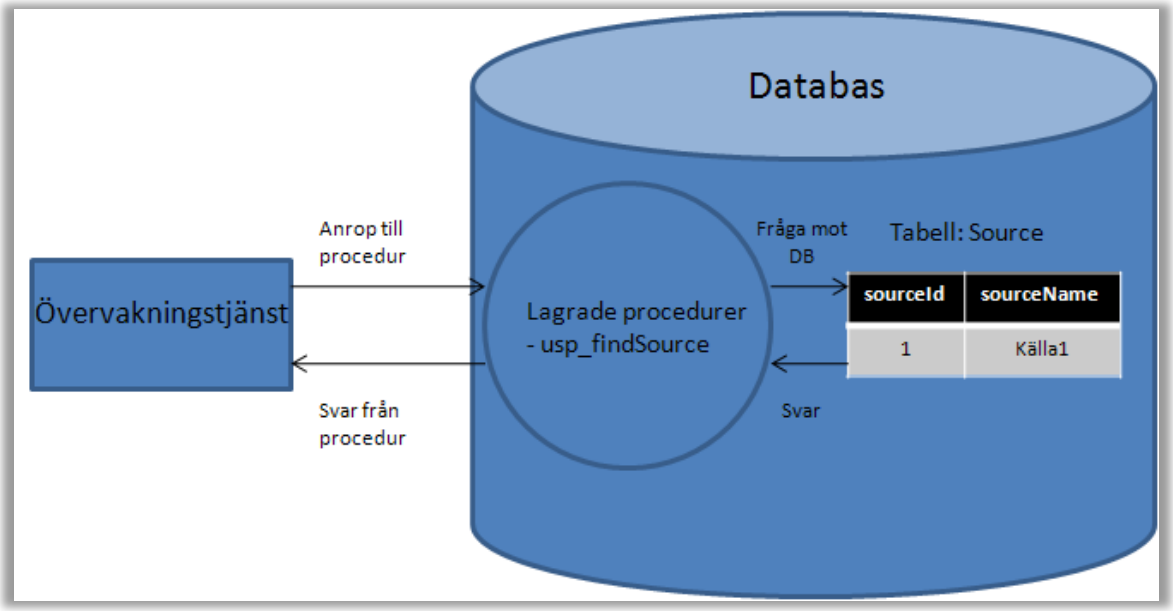
Figur 4.23 visar hur koden ser ut för en av de lagrade procedurerna som finns i databasen. Den lagrade procedurens uppgift är att lägga till en ny källa till databasens tabell Source.

```
ALTER PROCEDURE [dbo].[usp_addSource]

    @SourceName nvarchar(50),
    @SourceType nvarchar(50),
    @DataTypeId int
AS
BEGIN
    INSERT INTO Source (SourceName, SourceType, DataTypeId) VALUES (@SourceName, @SourceType, @DataTypeId)

END
```

Figur 4.23: Lagrad procedur för insättning av källa

4.5.1 Sökprocedurer

Sökprocedurerna heter usp_findSource och usp_findDataType och tar emot namnet på det attribut som eftersöks. T.ex. tar usp_findSource emot namnet på en källa och frågar databasen hur många förekomster av det namnet som finns i tabellen Source. Därefter returneras svaret till anroparen som då får reda på om eftersökt namn finns i tabellen eller inte (se Figur 4.24).

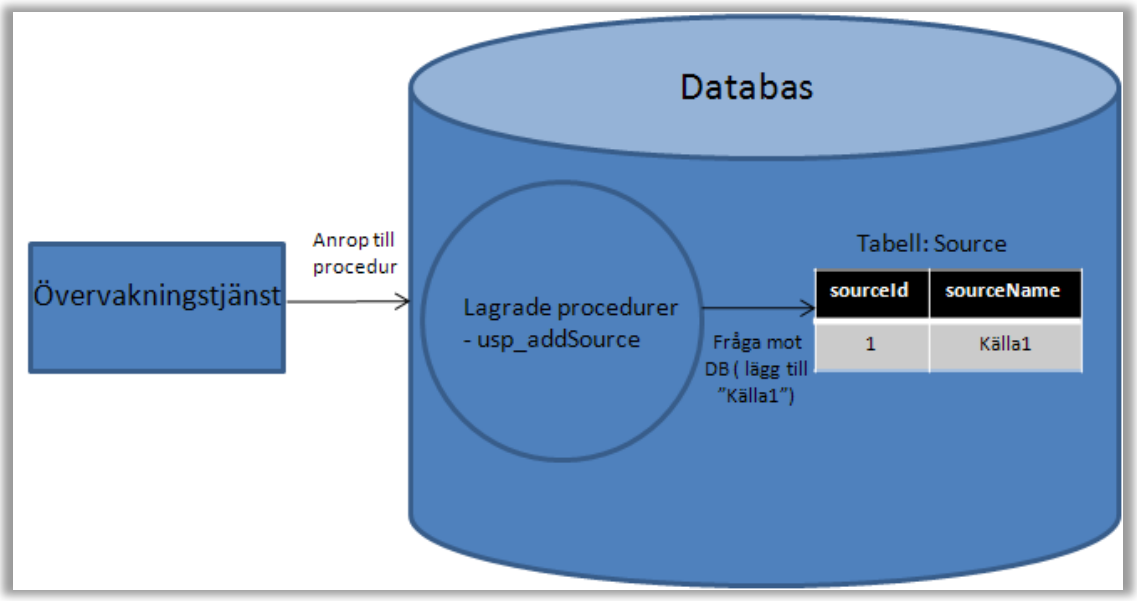


Figur 4.24: Sökprocedur

Anledningen till att dessa behövs är för att kontrollera att en källa, enhet eller datatyp redan finns i databasen innan den kan lägga till nya poster i databasen. Denna kontroll behövs eftersom insättning av poster till databasens tabeller från en källa som inte finns inlagd kommer leda till att fel uppstår då det inte finns någon källa i tabellen Source med det namnet.

4.5.2 Insättningsprocedurer

Insättningsprocedurerna heter usp_addSource, usp_addDataType, usp_addErrorMessage och usp_addMeasureData. Dessa procedurer tar emot de värden som ska läggas in i respektive tabell. Exempelvis tar usp_addSource emot namnet på en källa representerat av en sträng och lägger in det i databasen (se Figur 4.25). Insättningsprocedurerna behövs för att möjliggöra insättning av poster till databasens tabeller utan att anropa databasen direkt från övervakningstjänsten.

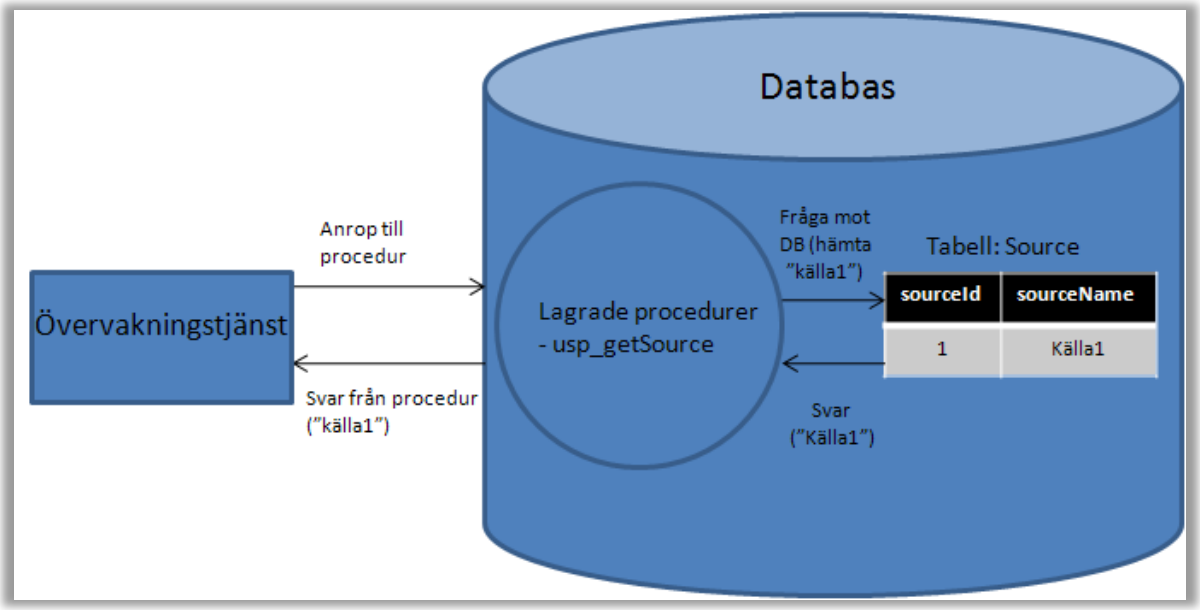


Figur 4.25: Insättningsprocedur

4.5.3 Hämtningsprocedurer

Hämtningsprocedurerna som heter usp_getSource, usp_getDataTypeProperties, usp_getMessage, usp_getSourceAndData, usp_getSourceId och usp_getMeasureData tar emot en hämtningsförfrågan och returnerar svaret.

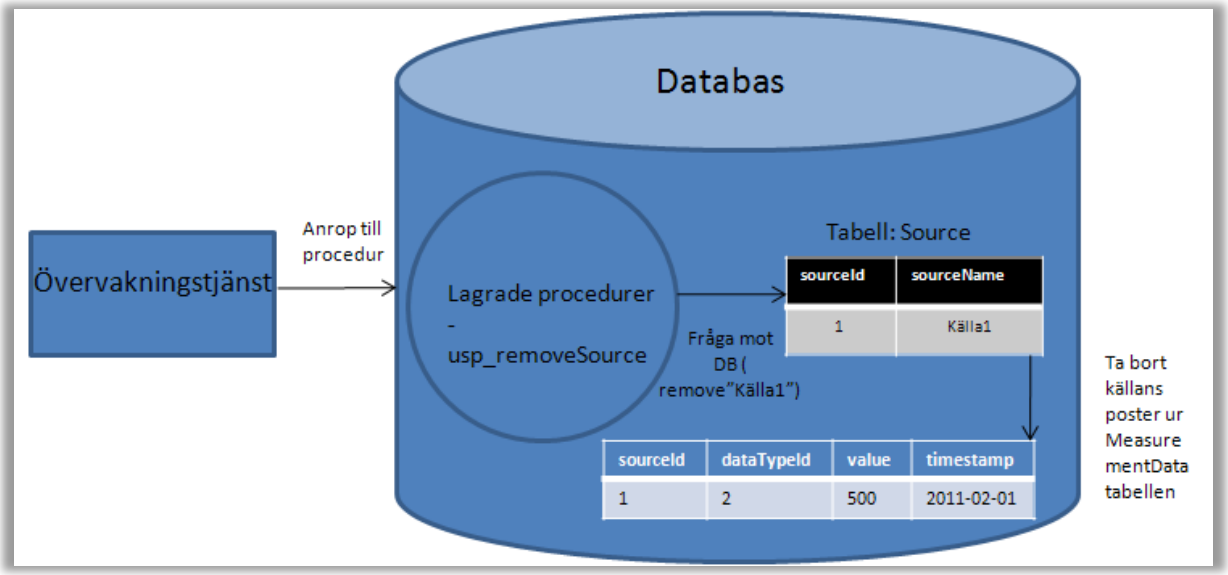
Exempelvis tar usp_getSource emot namnet på en källa representerat av en sträng och hämtar alla förekomster av den källan ur tabellen Source (se Figur 4.26). Hämtningsprocedurerna behövs för att möjliggöra hämtning av data ur databasen och som tidigare nämnts undvika att övervakningstjänsten kommunicerar direkt med databasen.



Figur 4.26: Hämtningsprocedur

4.5.4 Borttagningsprocedurer

Borttagningsprocedurerna usp_removeSource och usp_removeErrorMessage har i uppgift att ta bort poster ur databasen. Proceduren usp_removeSource uppgift är att ta bort en källa ur databasen. När källan tas bort tags även alla rader bort där källan förekommer som främmandenyckel i tabellen MeasureData och tabellen ErrorMessage (se Figur 4.27).



Figur 4.27: Borttagningsprocedur

4.6 Singletonmönstret

Singletonmönstret fungerar på följande sätt; Istället för att tillåta varje klient eller källa att skapa en egen instans av den dataaccessklass den är intresserad av, används samma instans av alla de klienter eller källor som använder klassen. Om flera av intressentklasserna ber om att få använda instansen på samma gång sätts deras förfrågan i en kö och behandlas i tur och ordning. När en klient eller källa använder en av dataaccessklasserna skapas ett lås som stänger ute nya anrop tills dess att klientens förfrågan behandlats.

Vi har bland annat använt singletonmönstret på klassen DataDistributer vars uppgift är att hämta data från databasen till klienterna. Varje gång en klient behöver skapa en ny instans av DataDistributer anropas en funktion som heter Create i DataDistributer som returnerar en instans till anroparen. I funktionen Create kontrolleras om det redan finns en instans skapad av DataDistributer och om så är fallet sänds den tillbaka till anroparen. Om det inte finns någon instans skapas en ny instans som returneras till anroparen (se Figur 4.28).

```
private static DataDistributer dataDistributerInstance = new DataDistributer();

/// <summary>
/// makes sure that datadistributer only exists as one instance
/// </summary>
/// <returns>an instance of datadistributer</returns>
public static DataDistributer Create()
{
    if (dataDistributerInstance == null)
    {
        dataDistributerInstance = new DataDistributer();
    }

    return dataDistributerInstance;
}
```

Figur 4.28: Create DataDistributer

4.7 Felscenarion

För att uppnå målet om en stabil övervakningstjänst gjorde vi följande modifieringar.

4.7.1 Inaktiva källor

För att undvika lagring av inaktuell data så beslöt vi oss för att borttagning av övervakade system och dess relaterade data skall ske ur databasen om de inte längre är aktiva. Det vill säga, om de inte laddats in från konfigurationsfil. Detta har vi gjort genom att jämföra de källor som laddats in med de som redan finns lagrade i databasen. Om en källa finns i databasen men inte bland de källor som laddats in så ska den och all data kopplad till källan tas bort. Detta kan vara bra om övervakningstjänsten av någon anledning kraschat och sedan startas igen. Det kan under den tiden övervakningstjänsten legat nere vara så att en källa slutat fungera och inte längre rapporterar någon data. Om så är fallet kommer källan tas bort ur databasen.

4.7.2 Gammal data

Om övervakningstjänsten samlar in data från många olika källor under en längre period är risken stor att databasen till slut kommer att innehålla stora kvantiteter data som egentligen inte är intressant för klienter. Syftet med vår övervakningstjänst är att ge klienter en ögonblicksbild över de övervakade systemens hälsotillstånd då de redan finns bra verktyg att använda om klienter vill ha lång historik över en specifik källas hälsotillstånd.

På grund av detta valde vi att varje gång ny mätdata läggs till av en källa så ska också utdaterad mätdata tas bort. För att ta reda på vilken mätdata som är utdaterad används ett värde ur konfigurationsfilen som anger hur många dagar en specifik källa ska lagra data. All data som funnits i databasen längre än angivet antal dagar tas bort.

4.7.3 Irrelevanta felmeddelanden

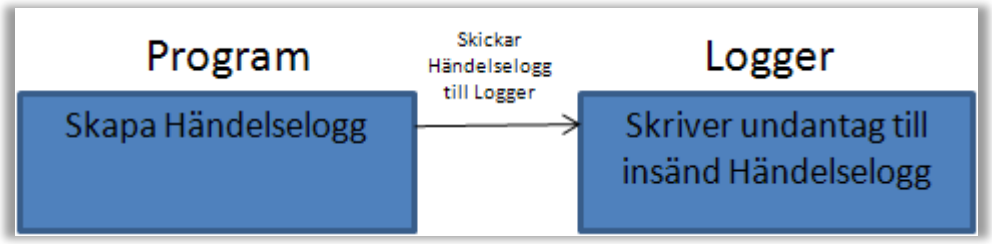
Vid de tillfällen vår övervakningstjänst stängs av och sedan sätts igång igen kommer alla irrelevanta felmeddelanden som finns lagrade i databasen tas bort. Detta sker genom att vi frågar alla källor som laddas in om de felmeddelanden som var aktiva vid avstängning av övervakningstjänsten fortfarande är aktiva när de laddas in igen. Om de inte längre är aktiva ska de tas bort ur databasen annars ska de ligga kvar.

4.7.4 Hantering av saknade mätvärden

Vår övervakningstjänst tar inte hänsyn till om det saknas data från en källa under vissa perioder. Vi kan göra på det viset eftersom övervakningstjänstens syfte är att kunna ge en ögonblicksbild över hälsotillståndet hos övervakade system.

4.7.5 Loggning

Alla fel som kan uppstå under körning av övervakningstjänsten skickas till klassen Logger vars uppgift är att skriva undantag som kastas till en händelselogg (se Figur 4.*29*.) Om något fel uppstår vid skrivning till loggfil fångas undantaget och skrivs ut i konsolen. Genom att skriva alla undantag som kastas till en händelselogg är det lättare att ta reda på var det blivit fel och åtgärda problemen.



Figur 4.29 Initialisering av loggklass

4.8 Sammanfattning av avsnitt

I detta avsnitt beskrivs konstruktionslösningen på en detaljerad nivå där vi berättar hur vi valt att implementera övervakningstjänstens olika delar. Avsnittet beskriver hur övervakningstjänsten startas och hur kommunikation mellan övervakningstjänsten, källor och klienter fungerar. Avsnittet beskriver också de lagrade procedurer som används för att ställa frågor mot databasen, designmönstret singleton samt olika felscenarion vi tagit hänsyn till i vår implementering.

5 Resultat

I detta avsnitt står att läsa om de resultat som uppnåtts efter genomfört projekt.

Vi har uppnått de primära målen och skapat en övervakningstjänst som samlar in statusinformation från olika källor, t.ex. brandväggar. Statusinformation kan t.ex. vara genomströmningshastighet och responstid. Dessa data ska kunna vidarebefordras till de klienter som är intresserade. Övervakningstjänsten kommunicerar med ett antal lagrade procedurer hos en databas av typen SQL-server 2008 Express för hämtning och lagring av data. Databasen möjliggör lagring och hämtning av mätdata samt felmeddelanden.

Klientkommunikation sker genom ett WCF -projekt om klienterna uppfyller det kontrakt som definierats mot klienter. Det fungerar så att klienter skickar en förfrågan varje gång de vill ha en uppdatering av data som finns lagrad och då svarar övervakningstjänsten med att skicka begärd data.

Övervakningstjänsten möjliggör kommunikation med övervakade system genom att ladda in dessa från en konfigurationsfil och tillhandahålla ett API som de skapade källmodulerna måste implementera för att kunna rapportera in data.

Vi har också uppfyllt det sekundära målet om inhämtning av felinformation från övervakade system. Övervakningstjänsten lagrar felmeddelanden som innehåller följande information:

- Typ av fel (ex. varning, kritiskt)
- Sammanfattande felmeddelande

Vi hann också starta implementeringen av en egen källmodul för brandväggskommunikation. Vi kom så långt att det går att ladda in modulen och att denna kan skicka mätdata till tjänsten. Vi hann däremot inte få kommunikation mellan modulen och en brandvägg via SNMP (Simple Network Management Protocol) att fungera. Anledningen till detta var brist på tid samt att det varit svårt att hitta ett kodbibliotek som var gratis och väldokumenterat.

Vi har dock fått hjälp av vår handledare på Ninetech, Henrik Bäck som skapat en källmodul ”åt oss” för testning från ett befintligt övervakningssystem som rapporterar in felmeddelanden så vi har sett att rapportering av felmeddelanden fungerar som tänkt.

Ytterligare ett av de sekundära målen som vi implementerat är att göra tjänsten stabil. Vi fann detta som ett vagt mål och svårt att avgöra om det var uppnått eller inte.
Ninetech hade sagt att de skulle göra en kodrecension av tjänsten och hjälpa till med detta mål, men det blev aldrig av. Vi har heller inte kunnat testa tjänsten i den miljö där det ska vara installerat eller kunnat köra det under en längre tid med olika de program som ska arbeta mot tjänsten. Det vi däremot har gjort med stabiliteten i åtanke är att ta hänsyn till olika felscenarion som vi själva tänkt ut, men vi är medvetna om att detta är långt ifrån tillräckligt för att kunna garantera att tjänsten är stabil.

Ett sekundärt mål vi inte berört alls är skapandet av en windowsklient för visualisering av den lagrade datan. Vi har dock använt oss av en i WCF inbyggd testklient och därigenom kunnat bekräfta att hämtning av data från vårt program fungerar på det sätt vi vill och då även kunnat bekräfta att den kod vi skrivit i WCF -projektet är korrekt.

Arbetet har flutit på bra och vi är nöjda med det uppnådda resultatet. Vi har uppnått alla primära mål samt några av de sekundära. Vi har haft mycket god kontakt med vår handledare på företaget som har bistått med all hjälp vi behövt, vilket gjort att vi sluppit sitta och grubbla länge över småfrågor.

6 Slutsatser och erfarenheter

I detta avsnitt redovisas de slutsatser vi dragit och vilka erfarenheter vi förskaffat oss. Under avsnitt 6.1 beskrivs slutsatser och avsnitt 6.2 redovisar vilka erfarenheter projektet gett oss.

6.1 Slutsatser

Den första slutsatsen vi dragit efter att ha gått denna kurs är att rapportskrivning tar mycket längre tid än vad man tror. Det gäller att vara med ifrån början om man ska slippa stressa vid kursens slut. Tack vare att vi började planera våra dagar med hjälp av en Scrum -board vid examensarbetets start satt vi inte sysslolösa, vilket vi anser bidrog till att vi kom igång fort med arbetet. Vi kan dessutom konstatera att vår handledare på Ninetech gjort ett föredömligt jobb som hjälpt oss så fort vi stött på problem. Detta har bidragit till att vi inte har suttit fast flera dagar i sträck utan att komma någonstans, vilket vi från andra kursdeltagare hört varit ett problem. Om situationen hade varit annorlunda hade vi förmodligen inte hunnit uppfylla något av de sekundära målen som sattes upp inledningsvis i projektet. Vi anser därför att det är viktigt med närvarande handledare och att företaget som är värd för examensarbetet faktiskt är intresserat av slutprodukten. Om inte företaget är intresserat kommer de antagligen inte lägga ner resurser på oss som examensarbetare och handledare skulle få mindre tid avsatt för oss.

Vi anser att examensarbetet varit den lärorikaste kursen under studietiden. Det har varit roligt att få känna på hur det är i arbetslivet och möta människor som är väldigt kompetenta inom det område vår utbildning är inriktad mot.

Vi känner också att detta arbete har utvecklat oss båda, då mycket av det vi gjort i projektet inte är något vi stött på tidigare under vår utbildning. Detta har också gjort att vi fått ett ökat självförtroende för att vi faktiskt fått med oss en stabil och bra grund från skolan ut till arbetslivet.

När vi ser tillbaka på projektet tycker vi att vi borde ställt högre krav på kunden att specificera kravet ”göra övervakningstjänsten stabil” ytterligare då det är svårt att ta hänsyn till alla felscenarion som finns. Speciellt då vi inte har haft tillräckligt med tid för att testa produkten i den miljö den installerats på.

6.2 Erfarenheter

En av våra sekundära uppgifter var att göra övervakningstjänsten stabil. Det är svårt att uppnå full stabilitet eftersom det finns ett oändligt antal testfall och för att vi inte haft tillräckligt med tid för att testköra programmet i dess korrekta miljö. Vi fick istället komma på olika scenarion som kan tänkas inträffa och ta hänsyn till dessa. Hur vi valde att göra programmet stabilt finns beskrivet i avsnitt 4.7**.**

WCF har varit intressant att lära sig då vi anser att det är ett smidigt sätt att kommunicera mellan olika program. Denna egenskap är önskvärd då klientprogram ska kunna kommunicera med övervakningstjänsten.

Vi valde att använda oss av lagrade procedurer vid databashantering efter rekommendation från Ninetech. Tidigare har vi använt oss av SQL -anrop direkt från de program vi skapat. Vi anser att lagrade procedurer varit lätt att lära sig samt att de fungerar bättre för vårt projekt än vad vanliga SQL -anrop direkt från programmet skulle göra då nätverkets belastning minskas. Detta är positivt i vårt fall då vi inte vet var databasen kommer att finnas i framtiden.

För att källor ska kunna kommunicera med övervakningstjänsten måste de använda sig av delegat. Dessa pekar till metoder som utför operationer mot databasen. När en källa laddas in från konfigurationsfil tilldelas den delegat till de metoder den får anropa vilket bidrar till att källan i sig inte behöver känna till något om övervakningstjänstens interna struktur. Detta gör det lättare att skapa nya källmoduler i framtiden. Användandet av delegat ställde till en början till en del huvudbry för oss men efterhand växte förståelsen och vi känner nu att det var en mycket bra erfarenhet att ha med sig.

Vi kan nu i slutet av projektet konstatera att det är viktigt att planera sitt arbete. De dagar vi inte har planerat vad vi behöver göra i förväg har vi gjort mindre. Det har dessutom varit intressant och lärorikt att få komma ut i arbetslivet och arbeta med ett ”riktigt” projekt till skillnad från laborationerna i skolan. Vi har fått erfarenhet av att krav förändras då kunden halvvägs in i projektet prioriterade stabilitet framför implementering av källmodul. Dessutom tycker vi att det är svårt att tidsuppskatta projekt då vissa delar av projektet vi trodde skulle vara enkla visade sig ta längre tid än vad vi räknat med och delar vi trodde skulle vara svåra visade sig ta kortare tid än vad vi från början trott.

Vi har också lärt oss mer om windowstjänster som vi inte använt oss av tidigare. Vi valde att skapa en windowstjänst då dessas främsta användningsområde är för långkörande serverapplikationer.

För att inte skapa för många instanser av de klasser som sköter databashanteringen så använde vi oss av designmönstret singleton. Istället skapas bara en instans av en klass som alla källmoduler använder för att kommunicera med databasen. Vi har insett att detta är att föredra framför att alla källor som övervakas har egna instanser av de klasser som sköter databashanteringen då det minskar risken för konflikter.

Referenser

[1] Visual Studio, http://en.wikipedia.org/wiki/Microsoft_Visual_Studio, 2011-02-14

[2] .NET Framework, http://en.wikipedia.org/wiki/.NET_Framework, 2011-02-14

[3] .NET Framework, http://sv.wikipedia.org/wiki/Dotnet_Framework, 2011-02-14

[4] C#, http://sv.wikipedia.org/wiki/C_Sharp, 2011-02-14

[5] C# / CIL, http://csharpskolan.se/showarticle.php?id=48, 2011-02-15

[6] Bytekod, http://sv.wikipedia.org/wiki/Bytekod, 2011-02-15

[7] SQL-Server, http://sv.wikipedia.org/wiki/Microsoft_SQL_Server, 2011-02-14

[8] SQL-Server Express, http://www.microsoft.com/sqlserver/2008/en/us/express.aspx, 2011-02-14

[9] SQL-Server 2008 Management Studio Express, http://www.microsoft.com/downloads/en/details.aspx?FamilyID=08e52ac2-1d62-45f6-9a4a-4b76a8564a2b#Overview, 2011-03-01

[10] SQL-Server 2008 Management Studio Express, http://msdn.microsoft.com/en-us/library/ms365247.aspx, 2011-03-01

[11] Relationsdatabaser, http://en.wikipedia.org/wiki/Relational_database, 2011-02-14

[12] Mssqltips, http://www.mssqltips.com/tutorial.asp?tutorial=160, 2011-02-08

[13] Lagrade procedurer, http://www.mssqlcity.com/Articles/Adm/stored_procedures_administration.htm, 2011-02-08

[14] Delegates, http://www.ict.kth.se/courses/ID132V/minikompendium/delegat_events.html, 2011-03-01

[15] Delegates, http://msdn.microsoft.com/en-us/library/ms173171%28v=vs.80%29.aspx, 2011-02-28

[16] Windows services, http://msdn.microsoft.com/en-us/library/d56de412%28v=vs.80%29.aspx, 2011-02-08

[17] Windows kommunikationsfundament WCF, http://msdn.microsoft.com/en-us/library/ms731082.aspx, 2011-03-07

[18] Asynkrona anrop, Bäck, Henrik, 2011-02-07

[19] DLL-filer, http://msdn.microsoft.com/en-us/library/ms682589%28v=vs.85%29.aspx, 2011-01-31

[20] Händelselogg, http://msdn.microsoft.com/en-us/library/system.diagnostics.eventlog.aspx, 2011-04-12

[21] Händelsetyper, http://support.microsoft.com/kb/308427/sv, 2011-04-12

[22] Singleton, http://www.dofactory.com/Patterns/PatternSingleton.aspx, 2011-03-22

Förkortningar

KaU – Karlstads universitet

DLL – Dynamic Link Library

API – Application programming interface. Ett API är en regeluppsättning för hur en programvara ska kunna kommunicera med en annan programvara.

CLR – Common Language Runtime

CIL – Common Intermediate Language

CLI - Common Language Infrastructure

WCF – Windows Communication Foundation

SSMS – SQL-Server Management Studio

SSMSE – SQL-Server Management Studio Express

EventLog – inbyggd

SNMP - Simple Network Management Protocol