KARRI LEHMUSVAARA

DYNAMIC COMPLEX EVENT PROCESSING FOR INDUSTRIAL MONITORING SYSTEMS

Master of Science Thesis

Examiner: Professor José L. Martinez Lastra

Examiner and topic approved in the Engineering Sciences Council Meeting on 05.02.2014

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master of Science Degree Programme in Factory Automation
LEHMUSVAARA, KARRI: Dynamic Complex Event Processing for Industrial Monitoring Systems
Master of Science Thesis, 103 pages, 2 Appendix pages
May 2014
Major: Factory Automation
Examiner: Professor José L. Martinez Lastra

Keywords: Complex Event Processing, Service Oriented Architecture, Web Services, SCADA, Industrial Monitoring, StreamInsight, Cloud based services.

Using Complex Event Processing (CEP) as part of monitoring systems is a state-of-the-art approach in the manufacturing industry that still requires development. The industry is increasingly moving towards implementing Service Oriented Architecture (SOA) based systems to respond to increasing demands of interoperability amongst other operations in a business organisation. Complex event processors are used as part of monitoring systems but current complex event processors are usually system specific. This thesis aims to propose and demonstrate a more dynamic approach for implementing an industrial monitoring system using complex event processing.

Service Oriented Architecture uses event-based messaging to communicate between different devices and systems. This creates large amounts of data in the monitored system. In order to infer important information from this vast body of data the CEP is used to query through the events. These queries are predefined and cannot be changed during runtime. The CEP holds the main logic of the monitoring system and thus dictates what the system actually monitors. Monitoring system requires the possibility to change the monitoring logic. This is why a method of dynamically adding queries will be proposed in this thesis. In order for a SOA-based monitoring system to be dynamic the CEP needs to be dynamic.

This thesis proposes a CEP solution with generic implementation, dynamic query definition during runtime and the possibility to use recursive user defined functions that allow reusing query templates in different solutions. The developed CEP is tested with two different implementation use cases. First one a simulated use case that tests the monitoring system performance with large amounts of events. Second one a manufacturing line implementation to demonstrate the monitoring system in an actual manufacturing environment. Tests were run on both use cases to gain information on how the CEP performs and to demonstrate the functionality of the developed monitoring system.

The developed CEP was used as a part of oil lubrication use case for IMC-AESOP project. IMC-AESOP project was an EU project researching how to apply state-of-the-art SOA-based systems to the industrial automation field.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Automaatiotekniikan koulutusohjelma
Lehmusvaara, Karri: Dynaaminen complex event prosessori teollisuuden monitorointijärjestelmille
Diplomityö, 103 sivua, 2 liite sivua
Toukokuu 2014
Pääaine: Tehdasautomaatio
Tarkastaja: professori José L. Martinez Lastra

Avainsanat: Complex event prosessori, SOA, Web palvelu, SCADA, Monitorointi, StreamInsight, Pilvipohjaiset palvelut.

Complex event prosessorin käyttö osana monitorointijärjestelmää on viimeisintä teknologiaa edustava lähestymistapa tuotantoteollisuudessa. Tuotantoteollisuus on yhä enemmän siirtymässä kohti SOA-pohjaisia järjestelmiä, vastatakseen kasvavaan järjestelmien yhteentoimivuuden tarpeeseen. Complex event prosessoreita käytetään monitorointijärjestelmien osina, mutta tämän hetkiset complex event prosessorit ovat yleensä järjestelmäkohtaisia. Tämän työn tavoitteena on ehdottaa ja demonstroida dynaamisempaa lähestymistapaa teollisiin monitorointijärjestelmiin käyttäen complex event prosessoria.

SOA käyttää event-pohjaista viestitystä kommunikoidakseen eri laitteiden ja järjestelmien välillä. Tämä luo suuret määrät dataa monitorointijärjestelmässä. Suodattaakseen tärkeän informaation datasta complex event prosessori ajaa event-pohjaisia viestejä kyselyiden läpi. Nämä kyselyt ovat usein järjestelmäkohtaisia ja määrittävät monitorointilogiikan, jota järjestelmässä käytetään. Monitorointijärjestelmät vaativat kuitenkin mahdollisuuden muuttaa näitä monitorointilogiikkoja. Tästä syystä työssä ehdotetaan kyselyiden lisäämistä dynaamisesti ajon aikana. Jotta SOA-pohjainen monitorointijärjestelmä olisi dynaaminen, tulee myös complex event prosessorin olla dynaaminen.

Tämä työ ehdottaa complex event prosessori – ratkaisua geneerisellä implementaatiolla, dynaamisilla ajon aikaisilla kyselyillä ja mahdollisuudella luoda rekursiivisia käyttäjän määrittämiä funktioita. Kehitettyä complex event prosessoria testataan kahdella eri toteutuskokonaisuudella. Ensimmäisessä toteutetaan simuloitu järjestelmä, jolla voidaan luoda suuret määrät event-pohjaisia viestejä. Tämä mahdollistaa monitorointijärjestelmän testauksen eri käyttöolosuhteissa. Toisessa toteutuksessa monitorointijärjestelmä liitetään osaksi tuotantolinjaa ja sitä testataan normaalia teollisuustuotantoa vastaavissa olosuhteissa. Molempia toteutuksia testataan tavoitteena kerätä tietoa siitä miten complex event prosessori ja koko järjestelmä suoriutuu erilaisista käyttöolosuhteista. Samalla todistetaan monitorointijärjestelmän toimivuus tuotantoteollisuuden ympäristössä.

Kehitetty complex event prosessori on osa IMC-AESOP projektin toteutuskokonaisuutta. IMC-AESOP projekti on EU-rahoitteinen viimeisintä teknologiaa edustava tutkimuskokonaisuus, jonka tavoitteena on kehittää SOA-pohjaisia järjestelmiä käyttöönotettavaksi tuotantoteollisuudessa.

PREFACE

This thesis was made for the Factory Automation Systems and Technologies Lab (FAST Lab.) of the Department of Mechanical Engineering and Industrial Systems at Tampere University of Technology, under the direction of Prof. Dr. José Luis Martínez Lastra, and Associate Professor Andrei Lobov.

Most of the work done for this thesis was made under the funding from European Commission's Framework Package 7 Project, "Architecture for Service-Oriented Process Monitoring and Control (IMC-AESOP)".

I would like to thank Dr. Lobov for all the hours he advised and supported me during this thesis and my time at FAST Lab. I am grateful that you challenged me to always aim higher and motivated me during the process. I would also like to thank Prof. Lastra for giving me the opportunity to work at FAST Lab. I learned a lot while working at FAST Lab and I am sure it has given me the skills I need, to learn more and succeed in the field of factory automation. I would also like to thank FluidHouse for providing a use case for this thesis.

Also big thanks for all the staff working at FAST Lab. It was great working in a happy and supportive environment which towards everyone contributed. Big thanks to Jorge, Juha, Luis and specially Johannes for helping me when I ran into problems. Many of the programs used in this thesis like FluidCirc Sim, DPWS hub and Ignition modules were originally developed by Johannes Minor at FAST Lab.

During my years of study at TUT NääsPeksi became really important to me with all its people. I would like to thank all my dear friends at NääsPeksi for the unforgettable years and all the joy you have brought me.

Finally, thank you to my family and friends at Turku for all the love and support during my studies. You have given me the courage to pursue my goals. No matter where my goals take me, Turku will always be my home as you are there.

Tampere, May 13, 2014.
Karri Lehmusvaara

CONTENTS

1 Introduction ... 1
1.1 Background ... 1
1.2 Problem definition ... 3
1.3 Work description ... 3
1.4 Assumptions and limitations ... 3
1.5 Use cases ... 4
1.5.1 Oil lubrication use case ... 4
1.5.2 Fastory use case ... 6
1.6 Thesis outline ... 6
2 State-of-the-art: SOA-based Monitoring Systems ... 7
2.1 Service-oriented architecture ... 7
2.1.1 ISA-95 ... 9
2.1.2 Event ... 10
2.1.3 Devices ... 11
2.2 SCADA: as a part of distributed monitoring ... 12
2.3 Web Services: the backbone of SOA ... 17
2.3.1 Web service standards ... 17
2.3.2 DPWS ... 19
2.3.3 WCF ... 22
2.4 Complex Event Processing: processing power ... 22
2.5 Cloud computing ... 23
2.6 Summary ... 25
3 Approach ... 26
3.1 Architecture ... 26
3.2 Architecture blocks ... 28
3.2.1 Process block ... 28
3.2.2 Complex event processor block ... 28
3.2.3 SCADA block ... 29
3.3 Testing ... 30
4 Implementation ... 32
4.1 Architecture ... 32
4.2 Technologies ... 33
4.2.1 Component selection ... 33
4.2.2 StreamInsight ... 34
4.2.3 S1000 Device ... 39
4.2.4 Ignition SCADA ... 39
4.2.5 Amazon EC2 ... 40
4.3 Complex event processor ... 41
4.3.1 Main program ... 42
4.3.2 Adapters ... 42

4.4 User-Defined functions 42
4.4.1 reactDouble 43
4.4.2 comparisonIntWithName 44
4.4.3 Recursive UDF for manufacturing monitoring 45
4.5 Query Management Client 49
4.5.1 Query creation 49
4.5.2 Web Service 52
4.6 Oil lubrication use case 53
4.6.1 System diagram 54
4.6.2 FluidCirc Simulator 54
4.6.3 DPWS event hub 58
4.6.4 Complex event consumer 58
4.7 Fastory line use case 59
4.7.1 System diagram 59
4.7.2 Fastory event messages 60
4.8 Experimental implementations 60
4.8.1 Oil lubrication 60
4.8.2 Fastory line 61
4.8.3 Test scenarios 62
5 Results 68
5.1 Test results for CEP measurements 68
5.1.1 Scenario 1: Normal LINQ query, 5min test. 68
5.1.2 Scenario 1: Normal LINQ query, 30min test. 69
5.1.3 Scenario 1: Normal LINQ query, 24h test. 70
5.1.4 Scenario 2: Normal LINQ query, 5min test. 72
5.1.5 Scenario 3: User defined function query, 5min test 73
5.1.6 Scenario 4: Two normal LINQ queries, 5min test. 74
5.1.7 Scenario 5: Two user defined function queries, 5min test. 76
5.1.8 Scenario 6: Additional normal LINQ query, 5min test. 77
5.1.9 Scenario 7: Additional two normal LINQ queries, 5min test. 78
5.1.10 Scenario 8: Fastory UDF query, 30min test. 79
5.2 Test results for cloud performance measurements 81
5.2.1 Scenario 1: Normal LINQ query, 5min test. 81
5.2.2 Scenario 1: Normal LINQ query, 24h test. 83
5.2.3 Scenario 2: Normal LINQ query, 5min test. 85
5.2.4 Scenario 3: User defined function query, 5min test 86
5.2.5 Scenario 6: Additional normal LINQ query, 5min test. 88
5.2.6 Scenario 8: Fastory UDF query. 30min test. 90
5.3 Discussion of results 92
5.3.1 Length of measurements 92
5.3.2 Maximum event processing rate 93
5.3.3 Query initialization issue 93

5.3.4 Small amount compared to large amount of incoming events........... 93
5.3.5 Normal LINQ query compared UDF query.. 94
5.3.6 Multiple queries compared to single queries 94
5.3.7 Recursive UDF 94
5.3.8 CPU and memory affecting CEP performance.................................. 94
5.3.9 Lessons learnt 95
6 Conclusion and Future 96
6.1 Implementation conclusions.. 96
6.2 Result conclusions.. 97
6.3 Future work 97
7 References 98
Appendix 1: Comparison UDF code.. 104

LIST OF ABBREVIATIONS

API	Application Programming Interface
AWS	Amazon Web Services
CEP	Complex Event Processing
CLR	Common Language Runtime
CRM	Customer Relationship Management
CTI	Current Time Increment
DCS	Distributed Control System
DPWS	Device Profile for Web Services
EDA	Event-driven architecture
ERP	Enterprise Resource Planning
ESP	Event Stream Processor
FAST	Factory Automation Systems and Technologies
HMI	Human-machine Interface
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IMC-AESOP	ArchitecturE for Service-Oriented Process – Monitoring and Control
ISA	International Society of Automation
JINI	Java Intelligent Network Infrastructure
LAN	Local Area Network
LINQ	Language Integrated Query
MES	Manufacturing Execution System
MS-CEPM	Microsoft Complex Event Processing Engine Manageability Protocol
OASIS	Organization for the Advancement of Structured Information Standards
OPC	Object Linking and Embedding for Process Control
OPC UA	OPC Unified Architecture
OSGi	Open Service Gateway initiative framework
PaaS	Platform as a Service
PLC	Programmable Logic Controller
QMC	Query Management Client
RFID	Radio-frequency identification
RTU	Remote Terminal
SaaS	Software as a Service
SCADA	Supervisory Control And Data Acquisition
SCM	Supply Chain Management
SIRENA	Service Infrastructure for Real-time Embedded Networked Applications
SLA	Service-level agreement

SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SOCRADES	Service Oriented Cross-Layer Infrastructure For Distributed Smart Embedded Devices
SODA	Service-Oriented Device & Delivery Architectures
UDF	User Defined Function
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WAN	Wide Area Network
W3C	World Wide Web Consortium
WCF	Windows Communication Foundation
WS	Web Service
WSDL	Web Service Description Language
XLST	Extensible Stylesheet Language Transformation
XML	Extensible Markup Language
XSD	XML Schema Definition

LIST OF FIGURES

Figure 1.1. Oil lubrication measurement station from Fluid House................................5
Figure 1.2. Flowmeter used in measurement stations. 5
Figure 1.3. Fastory manufacturing line. 6
Figure 2.1 ISA-95 Layers. (Brandl 2008) 9
Figure 2.2. Publish/Subscribe paradigm. 11
Figure 2.3. SCADA system evolution. (Karnouskos & Colombo 2011).......................14
Figure 2.4. Distributed business processes. (Karnouskos & Colombo 2011)................15
Figure 2.5. Future SCADA architecture. (Karnouskos & Colombo 2011)...................16
Figure 2.6. SOAP Envelope 18
Figure 2.7. Web Service standards (Bean 2010). 19
Figure 2.8. DPWS protocol stack. (Zeeb et al. 2007) 20
Figure 2.9. Cloud computing services (Buyya et al. 2011). 24
Figure 3.1. Architecture for the developed monitoring system.....................................26
Figure 3.2. Networked representation of the architecture. 27
Figure 3.3. Using ISA-95 for planning the messaging...28
Figure 4.1. Implemented monitoring system architecture..32
Figure 4.2. StreamInsight architecture. (MS-SI 2012)...34
Figure 4.3. StreamInsight queries and adapters at runtime (MS-SI 2012)......................38
Figure 4.4 Ignition SCADA HMI for Fastory line demonstration. 40
Figure 4.5. Information of the EC2 instance used for the implementation. 41
Figure 4.6. Instance information of m1.small instance type. (Amazon 2013)..............41
Figure 4.7. Query management client main tab. 50
Figure 4.8. Query Management Client QueryTemplate definition tab. 51
Figure 4.9. Query management client EventType definition tab. 52
Figure 4.10. Creating a query through Web Service interface.......................................53
Figure 4.11. Oil lubrication system diagram. 54
Figure 4.12. FluidCirc Simulator browser interface with measurement station selected. 55
Figure 4.13. System diagram for Fastory line implementation.....................................59
Figure 4.14. Component locations and functionality. 61
Figure 4.15. Component locations and functionality. 61
Figure 5.1. Average propagation delays for scenario 1, 5min test...............................69
Figure 5.2. Average propagation delays for scenario 1, 30min test..............................70
Figure 5.3. Average propagation delays for scenario 1, 24h test. 71
Figure 5.4. Average propagation delays for scenario 2, 5min test...............................73
Figure 5.5. Average propagation delays for scenario 3, 5min test...............................74
Figure 5.6. Average propagation delays for scenario 4, 5min test...............................75
Figure 5.7. Average propagation delays for scenario 5, 5min test...............................77
Figure 5.8. Average propagation delays for scenario 6, 5min test...............................78
Figure 5.9. Average propagation delays for scenario 7, 5min test...............................79

Figure 5.10. Average propagation delay for factory scenario, 30min test. 81
Figure 5.11. Processor time for scenario 1, 5min test. 82
Figure 5.12. Committed bytes for scenario 1, 5min test. 83
Figure 5.13. Processor time for scenario 1, 24h test. 84
Figure 5.14. Committed bytes for scenario 1, 24h test. 84
Figure 5.15. Processor time for scenario 2, 5min test. 85
Figure 5.16. Committed bytes for scenario 2, 5min test. 86
Figure 5.17. Processor time for scenario 3, 5min test. 87
Figure 5.18. Committed bytes for scenario 3, 5min test. 88
Figure 5.19. Processor time for scenario 6, 5min test. 89
Figure 5.20. Committed bytes for scenario 6, 5min test. 90
Figure 5.21 Processor time for scenario 8, 30min test. 91
Figure 5.22 Committed bytes for scenario 6, 30min test. 92

LIST OF TABLES

Table 2.1. Subsystems of a SCADA system and descriptions. 13
Table 2.2. Web Service standard descriptions... 20
Table 2.3. Workings of a CEP simplified to four different phases. 23
Table 3.1. Functions that the CEP needs to implement... 29
Table 3.2. Measurement parameters for CEP measurements. 30
Table 3.3. Measurement parameters for cloud performance measurements. 31
Table 4.1. Components selected for implementation. 33
Table 4.2. Relevant StreamInsight server concepts. (MS-SI 2012) 35
Table 4.3. Table of stream events.. 35
Table 4.4. Supported logic... 45
Table 4.5. Table of event messages sent from FluidCirc Simulator. 55
Table 4.6. Component location and platform characteristics. 61
Table 4.7. Component location and platform characteristics. 61
Table 4.8. Test scenarios with descriptions and LINQ queries. 63
Table 5.1. CEP measurement results for the 5 minute test of scenario 1. 68
Table 5.2. CEP measurement results for the 30 minute test of scenario 1. 69
Table 5.3. CEP measurement results for the 24h test of scenario 1. 71
Table 5.4. CEP measurement results for 5 minute test of scenario 2. 72
Table 5.5. CEP measurement results for 5min test of scenario 3. 73
Table 5.6. CEP measurement results for the 5min test of scenario 4. 75
Table 5.7. CEP measurements for the 5min test of scenario 5. 76
Table 5.8. CEP measurements for the 5 minute test of scenario 6. 77
Table 5.9. CEP measurements for the 5min test of scenario 7. 78
Table 5.10 CEP measurements for the 30min test of scenario 8. 80
Table 5.11. Improvements and lessons learnt. 95

1 INTRODUCTION

Modern factory floors are highly automated and rapidly changing environments. Thus maintaining a modern industrial manufacturing system is becoming increasingly expensive and time consuming. (Cachapa et al. 2010c) The manufacturing industry requires flexible tools for monitoring manufacturing systems to cut down the time and costs for improving the system. The goal of this thesis is to develop a dynamic complex event processor (CEP) to allow implementation of a flexible industrial monitoring system.

Monitoring systems are an important part of an industrial manufacturing system as the production engineers must have a real-time view of the machine's status, production flow, stock management as well as other essential production indexes to keep production efficient constantly. Resource planning (ERP), Supply Chain Management (SCM), Customer relationship management (CRM) and Manufacturing Execution System (MES) have improved operation efficiency in companies but only when they are supported by effective architectural styles like Service-oriented architecture (SOA). (Fan et al. 2005) The monitoring systems implemented in this thesis are based on SOA architecture.

This thesis will present two use case implementations of industrial monitoring systems using the complex event processor developed in this thesis. The first use case was developed as a part of IMC-AESOP projects oil lubrication use case. It is a monitoring system that monitors a simulated oil lubrication system and was developed to provide a proof of concept for using SOA in a monitoring environment. The second use case was implemented on an actual SOA-based assembly system called Fastory line. This use case demonstrates the functionality of the CEP and the whole monitoring system on a non-simulated environment.

1.1 Background

Industrial control systems came around in the late nineteenth century as the aim was to free human labour from operating and monitoring machine processes. In the 1980s, industrial control systems consisted of programmable logic controllers (PLC) and supervisory control and data acquisition (SCADA) systems. In the 1990s, industrial control systems started to incorporate computers as the microprocessors and programmable integrated circuits developed. This allowed more powerful and efficient industrial systems. (Zhang, 2010) The technology around Industrial control systems has been the target of heavy development as the market and demand grew.

According to European Commission study (European Commission, 2007) on Monitoring and Control the worldwide market for monitoring and control represents

just below 200 billion Euros in revenue, with Europe holding one third of a share representing 61 billion Euros. Growth is estimated at 8% per year on average between 2007 and 2020. European share rising to 143 billion compared to worldwide total reaching 500 billion. Factory automation remains the main market with a 58 billion Euros share of the market in 2007. The study represents the growth possibilities in the field of monitoring and control. In order for the European companies to stay competitive in this market and in the field of manufacturing itself, constant development of monitoring systems is required.

As government and customer requirements increase for the products in the form of quality, price and method of production, so does the complexity of the manufacturing. (Cachapa et al. 2010c) Implementing a modern day manufacturing line with all its complexities is both expensive and time consuming. European industries need to compete against the low cost production facilities in other continents. This is why European Union also emphasises research into monitoring and control systems. The work done for this thesis is a part of European Commission-sponsored IMC-AESOP (Architecture for Service-Oriented Process-Monitoring and –Control) research and development project that concentrates on challenges of very large scale distributed systems. IMC-AESOP envisions a Service-Oriented Architecture approach for monitoring and control of Process Control application (Karnouskos et al. 2010). Aim for the project is to develop tools, service specifications and reference architecture for implementing a SOA based monitoring and control system for very large scale processes. Project concentrates on challenges such as:

- Determine how large is the percentage of all devices that reliably can be incorporated in the SOA architecture.
- Define a foundation to predict the performance of such SOA architecture based on a formal approach to event based systems.
- Define a transition path from legacy systems to a SOA compliant system.
- Develop a SOA based monitoring and control system for very large scale distributed systems in process control applications.
- Propose a transition path for SOA based monitoring and control system and the next new system.

To address integration of very large numbers of subsystems and devices, the IMC-AESOP project takes its roots in previous work in several European collaborative projects such as ITEA SIRENA, SOCRADES and VINNOVA which demonstrated that embedding Web Services at the device level and integrating these devices with MES and ERP systems at upper levels of enterprise architecture was feasible. The first results shown in pilot applications running in the car manufacturing, electromechanical assembly and continuous process scenarios have been very successful, confirming that the use of Cross-layer Service Oriented Architectures in the Industrial automation domain is a very promising approach, able to be extended to the domain of control and monitoring of batch and continuous processes. (Karnouskos et al. 2010)

1.2 Problem definition

Using service oriented architecture and complex event processors in the monitoring domain of manufacturing industry is a state-of-the-art approach that still requires development. Some problems identified in earlier FAST Lab developments of similar systems have been identified. The fact that even small changes to the monitored variables requires recoding critical parts of the CEP is a major problem. The ability to change the monitored variable as the system is running is an important functionality for a monitoring system. This means that the CEP requires the ability to add CEP queries dynamically during runtime. A functionality that is not possible with the current implementations. Another problem has been that a CEP coded for a certain monitoring system is implementation specific and cannot easily be reused on other similar cases. The problem is how to generalize the CEP and how to reuse the query templates made for other implementations to be used in other similar cases. All of these problems are widely identified in the CEP development community but no solutions exist at the moment. The CEP developed in this thesis will tackle these problems by developing a generic CEP capable of dynamically adding queries and the ability to use generic user defined functions. The CEP will be the main part of the monitoring system developed in this thesis.

Also a problem that the developer will encounter when developing a SOA based monitoring system is that there is little data on how many devices and how much data can these systems handle. As a part of this thesis the CEP will be tested with different amount of devices and event loads to give results on CEP performance in different scenarios that can help development of other similar systems.

1.3 Work description

The objectives for this thesis are based on IMC-AESOP project and the future work section in Johannes Minor's thesis.

1. Implement a web service interface for managing and defining queries for the StreamInsight component. (From future work part of Johannes Minor's Thesis)
2. Develop a method of dynamically adding query definitions during CEP runtime.
3. Generalize the CEP implementation and develop a method to reuse generic user defined functions.
4. Conduct performance tests on the developed complex event processor in order to test event load capabilities of the complex event processing tool.

1.4 Assumptions and limitations

Current industrial manufacturing systems use a wide range of different devices and technologies in their processes. In order for the developed CEP to work on an industrial system the following assumptions are made.

Assumption 1: The target systems devices have web service capability.

Assumption 2: Information exchange between devices and cloud based CEP is possible without firewall or security problems.

Assumption 3: The target system and CEP adapter bindings match.

1.5 Use cases

Two different use cases were implemented to demonstrate the functionality of the developed CEP and monitoring system. The following use cases were chosen because they both allow different testing and demonstration possibilities.

1.5.1 Oil lubrication use case

Oil lubrication use case was implemented as part of IMC-AESOP projects use case 2. IMC-AESOP project use case 2 was called oil lubrication use case but will be referred to as use case 2 to avoid mix-up. The projects use case 2 addresses the manner of how the FluidHouse lubrication system that is used with paper machines is monitored. Paper machines require constant lubrication of hundreds of different points, each of these points require a flow meter and each of these point need to be monitored. In legacy systems monitoring a single flow meter was hard and required the process operator to check the flow values manually from each meter situated around the paper machine complex. Oil lubrication system compromises of measuring stations and lubrication units. A measuring station is shown in figure 1.1 which holds 24 flow meters like the one shown in figure 1.2. Flow measuring station is a product of FluidHouse ltd and is a part of a larger fluid circulation system that lubricates big industrial machines like paper machines with precise lubrication needs resulting from high production speeds. Any problem with the lubrication could result in production delays and big losses.

Figure 1.1. Oil lubrication measurement station from Fluid House.

Figure 1.2. Flowmeter used in measurement stations.

The IMC-AESOP Oil lubrication use case goal is to develop a flexible, adaptable and agile monitoring system for oil lubrication in paper machines by implementing event based service oriented architecture. Aim is also to create a system architecture based on distributed devices for it to accommodate large amounts of devices and for the system to be easily scalable. Low level data from the devices would be used to create decision helping data for the monitoring system. In order to test with large amounts of devices the process part as in the oil lubrication part of the monitoring system will be simulated.

Oil lubrication use case was made for industry needs to test SOA-based monitoring on FluidHouse products. The use case provides a simulator that allows testing the developed CEP and monitoring system. Simulator also allows simulating different process scenarios that can be used for performance measurements.

1.5.2 Fastory use case

The second use case was implemented on Fastory manufacturing line which is a production line situated at FAST Laboratory facilities in Tampere University of Technology. Fastory line consists of 11 robotic cells and a static buffer. Production line was originally used for assembling mobile phone covers. The production line simulates production by drawing mobile phone parts on to a paper attached to a pallet. Fastory line is shown in figure 1.3.

Figure 1.3. Fastory manufacturing line.

Fastory line is used by the FAST Laboratory for research and development purposes. The line is SOA ready and holds 38 service capable remote terminal units connected to Ethernet. The second use case allows demonstrating the functionality of the developed monitoring system and works as a proof of concept. It aims to proof the monitoring capabilities by running it on an actual manufacturing line performing production monitoring.

1.6 Thesis outline

This thesis is structured into six chapters. Chapter two describes the state of the art for SOA-based monitoring systems, which is followed by approach in chapter three describing the architecture and testing approach for the implementation. Chapter four presents the implementation part with two different monitoring implementations. Results follows in chapter five and conclusions and future work in chapter six.

2 STATE-OF-THE-ART: SOA-BASED MONITORING SYSTEMS

The production monitoring system plays a pivotal role in the modern manufacturing scenario. As the production process evolves in order to face the increasing demand and competition, so too must the production monitoring system adapt to face the new industrial reality. (Cachapa et al. 2010a) Monitoring can be understood in many ways but in this thesis monitoring is defined as the act of identifying the characteristic changes in a process and in the behaviour of production resources by evaluating process and component signatures without interrupting normal operations. (Elbestavi & Wu 1995)

This chapter will go through the state of the art technologies associated with SOA-based monitoring system. It is divided into five segments each describing different technologies, standards and specifications as a part of the big picture.

2.1 Service-oriented architecture

Industrial automation systems are moving towards more distributed systems with increasing requirement for interoperability amongst other operation in a business organisation. One solution to address this comes from Service oriented architecture (SOA). Service oriented architecture proposes an architecture that is suitable for large distributed systems. Jammes and Smit (2005) define SOA the following way "A Service-oriented Architecture is a set of architectural tenets for building autonomous yet interoperable systems". The authors do stress that this definition is incomplete and it includes two key words "autonomous" and "interoperable". Cachapa et al. (2010a, 2010b) uses the same definition and explains the key words the following way "Autonomous systems operate independently of their surroundings and do not depend on others to achieve their full functionality, while interoperable systems expose their interfaces as services at their border making it possible that they can be completely replaced by another system that exposes the exact same interfaces, even if they hide a completely different implementation". OASIS defines SOA in their reference model as: "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains". Jammes and Smit see SOA as set of tenets and OASIS defines SOA as a paradigm. SOA does not have a commonly agreed architecture with main blocks and functions but more of a set of tenets for designing such architectures for services.

In service oriented architecture a service is an autonomous unit of software that performs a specific task. OASIS describes a service as "A mechanism to enable access

to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description" (OASIS 2006). In SOA each service cooperates with other services of the systems to reach a common goal. If one service fails it can affect the functionality of the system as a whole but does not affect any individual service. The failed services can be replaced by a service with the same functionality and interface. The new service can be implemented in any way if the functionality and interface stays the same, thus making updating and hosting services simpler as each service is an independent part of the system.

According to Ragavan et al. the main idea of SOA is to create services that can be constructed together in order to build a system. The authors also define services as well-defined and self-contained functions independent of the context or state of other services. (Ragavan et al. 2010) Then again Hall and Cervantes bring up dynamism and substitutability as the main defining traits of SOA. Authors explain dynamism as the ability for service providers to offer and retract services at any time and for service requestors to bind available services at will. Substitutability is derived from the fact that service descriptions represent contracts. As a result, any service can participate in a SOA is open for services abiding to these contracts and any of these services can be substituted with another service obeying the same contract. (Hall & Cervantes 2004) This highlights the flexibility of SOA architecture from a slightly different viewpoint from the earlier definition. Taking these definitions and key traits into account SOA is in essence a scalable, modular and flexible architecture composed of services orchestrated to fulfil a common operational goal.

In order to understand SOA better it is good to know how it is constructed. Colombo et al. (2010) define instructions in their article "Factory of the Future: A Service-oriented System of Modular, Dynamic Reconfigurable and Collaborative Systems" on how to build a service oriented architecture. The following three steps are required:

- Identification of the cooperative systems: the identification of the collaborative automation units that are able to expose and/or consume services for each production scenario in a defined production domain. A collaborative unit can be a simple intelligent sensor or a part/component of a modular machine, a whole machine and also a complete production system.
- Building the system of systems: networking / bridging the entities together within an SOA or collaborative infrastructure as in putting the units architecturally together
- Making the system work for reaching the production goal: collaborative behaviour of the systems for reaching common objectives, i.e., control objectives, production specifications, markets objectives, etc.

According to Cachapa et al. (2010c) bringing SOA into the production line requires a careful design in order to keep the architecture and hierarchical separation clear. SOA approach can have many benefits when just implemented on the shop-floor,

but it also allows opening up the services to the whole enterprise system. Traditionally enterprise systems such as ERP and MES have been separate. Connecting the two can yield big benefits by improving communication and allowing information exchange between the two as enterprise management and manufacturing processes are no longer seen as completely separate parts of the business.

2.1.1 ISA-95

ISA-95 is the international standard for the integration of enterprise and control systems (ISA 2010). According to Karnouskos et al. (2012) ISA-95 is the most popular definition and widely applied in practice for defining structural and architectural aspects of production management systems. The important part of ISA-95 is the way it separates a production system into 5-level hierarchical model as presented in figure 2.1. It helps designing an industrial system by separating it into distinctive layers.

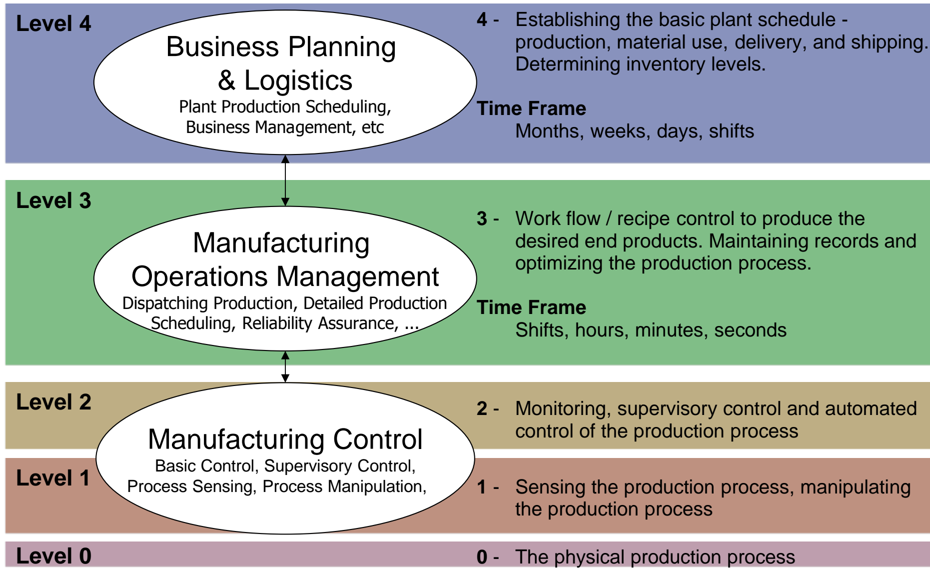

Figure 2.1 ISA-95 Layers. (Brandl 2008)

ISA-95 standards hierarchical layers presented in figure 2.1 are a good tool for planning information exchange in service oriented architecture. This helps to keep the architecture and hierarchical separation clear. Each level of hierarchy hosts different technologies and handles data in different form. By dividing services into each ISA-95 layer it is easy to identify what kind of data is exchanged between specific services and in what format. The main aim is to identify interfaces between layers and make the communication work between them.

Different ISA-95 layers consist of 5 levels. Levels 0 and 1 are usually referred to as the device level where level 1 is for sensing and manipulating the production process usually consisting of sensors and actuators. Level 0 is the actual physical production process where changing sensor states and actuator movements occur. It can be identified that all the process information is generated on this level. Device level consists of usu-

ally small resource-constrained devices that link the system to the physical process. Level 2 in general terms is concerned with the control and visibility of production processes. This does not include the real time control of processing equipment, which handled by Level 1, but more the integration of the level 1 controllers to achieve specific tasks related to production objectives. Manufacturing operations management level 3 is also referred to as Manufacturing Execution System (MES) level. The MES level handles workflow and objectives, in order to produce the desired products. Level 4 deals with business planning and logistics, such as plant production scheduling and operational management. The timeframe of level 4 differs from the other levels with a timeframe of days, weeks, months or shifts, which is a considerable different scale when comparing to the other levels with a time frame from hours to milliseconds. (McLeod & Karnouskos 2013)

2.1.2 Event

Industrial monitoring system monitors for changes in the system. These changes happen constantly and can be anything from a sensor value change to a time value exceeding a certain point. These changes are called events. Event is regarded as a meaningful change in a system (Luckham 2002). System operators are interested exactly on these small changes in the system and monitor these events with monitoring systems. Monitoring systems need to detect each event happening in the system. Devices at the device level of the system detect these events in the process and sends event notification messages to the subscribers. Event does not always need to be a change as Luckham and Frasca (1998) brings out that event denotes an activity. A good example of this according to the authors is the heart beat event which denotes that the source component is still active. This event is reported even if no changes occurred in the system thus only informing the activity of a certain component. Event notification messages are the method used to inform the system about activity or changes occurring in the process.

Events occur on the level 0 of the ISA-95 layers as in the physical process. These events are detected by devices situated in level 1 which sends event notification messages forward to other levels that have subscribed to notification messages from devices. Notification messages can be sent from services residing on any ISA-95 level. Not just on the device level. Services on each level can subscribe to each other’s messages but they can also communicate using messages e.g. invocation messages or reply messages.

Notification messages are sent to a subscriber following the subscription paradigm described in figure 2.2. Device or system requesting for notification messages is called a subscriber and the device sending the message a publisher. Walzer et al. (2008) presents in their journal that Publish/Subscribe system is based on an asynchronous messaging paradigm where there is no direct connection between the producer and the consumer of a notification message. Subscription is added by sending a subscription message to the publisher and the subscribers address information is added to the subscription list. Advantages of Publish/Subscribe paradigm is that it requires no changes

to the system when adding new subscribers thus making it an easy way to connect services together. Publish/Subscribe paradigm is based on a W3C standard for Web service specifications (WS-*) eventing specification called WS-eventing. This specification describes a protocol that allows Web services to subscribe to or accept subscriptions for notification messages (W3C 2011).

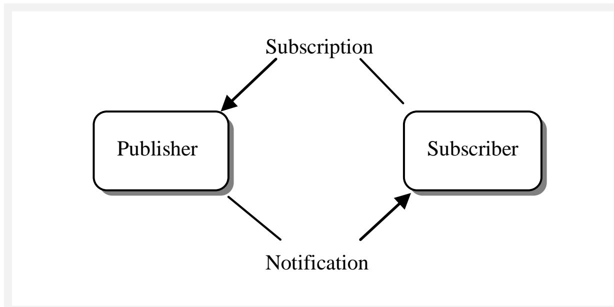

Figure 2.2. Publish/Subscribe paradigm.

Publish/Subscribe paradigm allows a more efficient information exchange. Event based information gathering is more efficient compared to the scan based information gathering that is used by most of the fieldbusses today, which are popular in factory automation. Scan based systems operate by scanning the device status one by one. This is why it could miss a critical change in the system if a change occurs while the system is scanning another device and changes back before it is scanned. Scan times increase as the size of the system grows limiting the size of a system depending on the real-time requirements of the system. Scan based systems operate on a pull mode scanning even when no changes occur on the process. Contrary to scan based systems the event based system operates in a push mode pushing the information only when changes occur. This reduces the bandwidth usage from scans with no new information by only sending information when new information is produced.

It is important to notice that the word event can have two meanings. It can both denote the actual physical change and the message carrying information of this change. When talking about event processing the word event usually means the message containing the data of an event that occurred in the process.

2.1.3 Devices

According to Cachapa et al. (2007) the development of a SOA-based production line requires devices which are autonomous, self-contained and independent of external devices. In addition they need to be able to cooperate with other devices and expose functionalities via services. Devices used in SOA-based production lines should integrate a degree of built-in intelligence, including the machine logic which allows it to achieve its functionality. (Cachapa et al. 2010b) These devices require support for services. Cachapa et al. (2007) define more specifically that devices need to support standardized Web Service interfaces, eventing and discovery in order for devices to be capable of

connecting to the network, and have their full functionality immediately available. Cachapa et al. (2010c) continue in the same lines and claim that a production line requires a modular device which can be plugged into the production line and cooperate in a network configured to accomplish a given task. The authors add that devices should be autonomous and able to communicate in a standard-based manner because "Standardization guarantees better flexibility for the system builder in mixing and matching parts from different suppliers". (Cachapa et al 2010c)

A device designed for SOA environment would yield the following benefits:

- Easy adoption: it is possible to deploy the technology incrementally either by gradually replacing components, or using middleware solutions in older equipment (Priyantha et al. 2008).
- Easy integration: made possible by the standards-based nature of SOA and WS technology.
- Easy to develop new applications: SOA and Web Services are at the heart of new programming paradigms heavily endorsed by influential software companies such as Microsoft® and Sun®.
- Reduced time for setup: made possible by the high-level nature of Web Services, and facilities such as discovery and eventing. (Cachapa et al 2010c)

Device Profile for Web Services (DPWS) and OPC UA are emerging technologies for realizing web service enabled controllers and devices. This thesis will concentrate on DPWS based devices. DPWS will be described in the following chapter 2.3.2.

For the first time SOA-ready devices are entering the markets as the technology matures. Several projects such as SIRENA, SODA and SOCRADES have experimented with SOA-ready industrial automation devices and their integration on industrial applications. (Karnouskos & Colombo 2011) SOA-ready devices are able to integrate to the SOA architecture. Examples of such devices are Inico S1000 and Mulle devices that can integrate to SOA architecture (Inico 2010, Delsing et al 2010).

2.2 SCADA: as a part of distributed monitoring

Supervisory control and data acquisition (SCADA) systems are used to encompass the transfer of data between a computer hosting SCADA and remote terminal units (RTU) (NCS 2004). According to Galloway and Hancke (2012) remote terminal units are the control hardware that communicates with a SCADA and is usually a type of specialised PLC. Daneels and Salter (1999) describe SCADA system as purely a software layer, normally applied a level above control hardware within the hierarchy of an industrial network. As such, SCADA systems do not perform any control, but rather function in a supervisory fashion. The focus of a SCADA is data acquisition and the presentation of a centralised Human Machine Interface (HMI) (Galloway & Hancke 2012).

According to Karnouskos and Colombo (2011) industrial processes as well as many modern systems depend on SCADA and distributed control systems (DCS) in order to perform their complex functionality. DCS systems have a lot of similarities with SCADA systems but as this thesis concentrates on monitoring systems, distributed control systems are out of the scope and only SCADA systems will be addressed.

Typical examples of systems where SCADA is used are electric power grids, oil refining plants, pharmaceutical manufacturing and water management systems with the main task of monitoring and control over a highly diversified infrastructure (Karnouskos & Colombo 2011). SCADA systems are used in many different applications that require the SCADA to communicate over long distances reliably. This is why SCADA systems tend to be event-driven rather than process-driven. Galloway and Hancke (2012) explain that event-driven SCADA systems focus on reporting changes in the state of the monitored system rather than sending a steady stream of process variables. This reduces the communication sent between SCADA and the monitored system.

When talking about SCADA systems it can be seen as the whole monitoring system as seen in table 2.1 that describes the SCADA subsystems that are required for a functioning SCADA system. But SCADA as itself is usually understood as the HMI and the server collecting the information displayed by the HMI.

Table 2.1. Subsystems of a SCADA system and descriptions.

Subsystem	Description
Human-Machine Interface	Where the information is depicted and is used by human operators to monitor and control the SCADA linked processes.
Monitoring computer	A computer which does the monitoring (gathering of data) as well as the control (actuation) of the linked processes
Remote Terminal Units	Collect data from the field deployed sensors, make the necessary adjustments and transmit the data to the monitoring and control system
Programmable Logic Controllers	Used as an alternative to RTUs since they have several advantages over the special-purpose RTUs
communication infrastructure	Connects all the SCADA components together

As described at the beginning of the chapter SCADA systems have previously been used as a part of event-driven systems. As SOA has been gaining popularity SCADA systems have also began to support this approach. Karnouskos and Colombo (2011) describe the evolution of SCADA systems in their journal “Architecting the next generation of service-based SCADA/DCS system of systems” with three generations and that current SCADA system architectures were designed for more closed and controlled industrial environments. This same division to three generations was done earlier

by McClanahan (2003) who described evolution from monolithic to networked SCADA and highlighted the importance of open protocols. Figure 2.3 represents the SCADA system evolution described in Karnouskos and Colombo's journal that is similar to McClanahan's description. The first generation had monolithic systems connected via WAN to RTUs near the actual processes. The second generation technology moved toward a more distributed layout by using LAN. Second generation allowed distributed processing, real time information sharing and was more cost effective compared to the first generation. According to IEEE standard for SCADA and Automation systems the advantage of distributed processing is that a failure in one part of the system does not necessarily affect the system as a whole (IEEE 2007). The Still emerging third generation moves towards more open system architecture and uses the internet network as such for communication infrastructure. (Karnouskos & Colombo 2011)

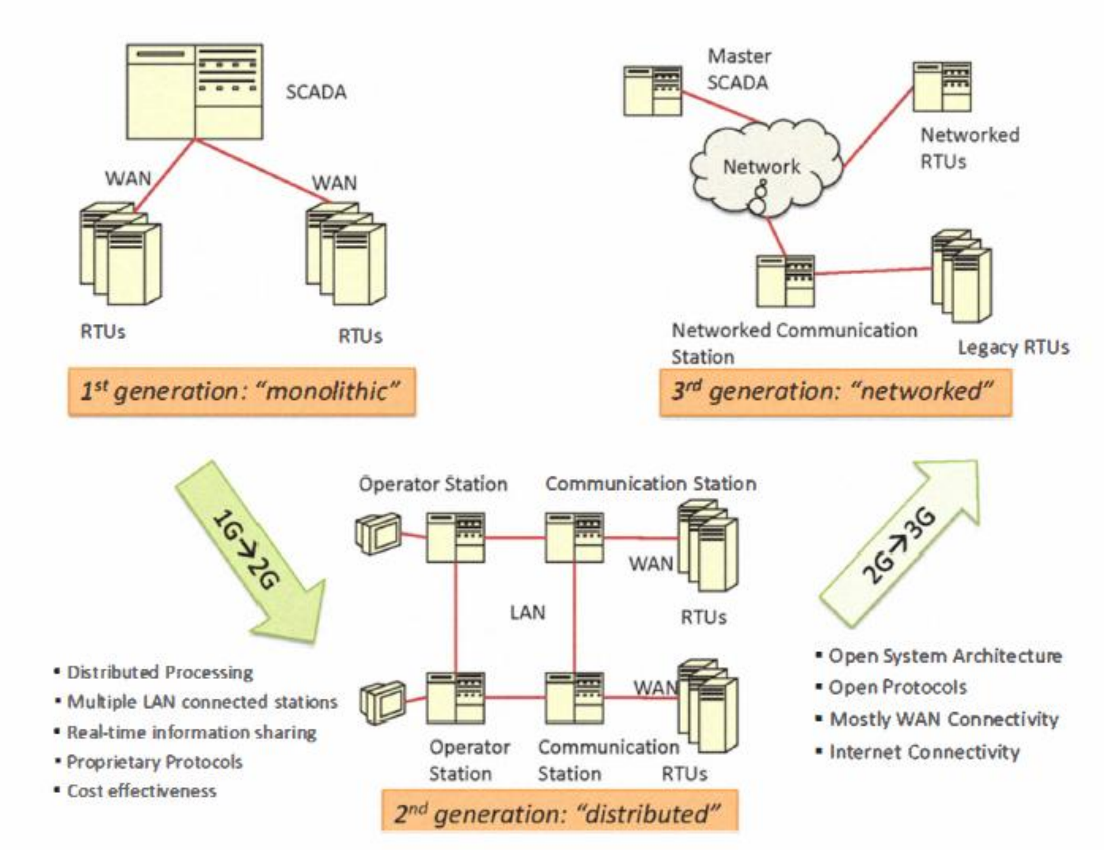

Figure 2.3. SCADA system evolution. (Karnouskos & Colombo 2011)

Many of the third generation SCADA systems are based on service oriented architecture for its open systems architecture and open protocols. The transition from monolithic systems to networked systems can be illustrated by the figure 2.4 which describes the transition that occurred when devices became increasingly capable of computing. All computing was traditionally performed in the enterprise layer that could also be seen as MES layer. Transition enabled parts of the computing tasks to be outsourced to the device and network layer. This transition was necessary to answer the demands of

modern enterprises as they require agility and quick decision making at different levels. Critical information is required to be available in several different layers at a timely manner. (Karnouskos & Colombo 2011) As industrial processes generate huge amounts of data that require processing and communication on-demand and on-time has 3rd generation SCADA been developed to answer to these demands with service oriented architecture.

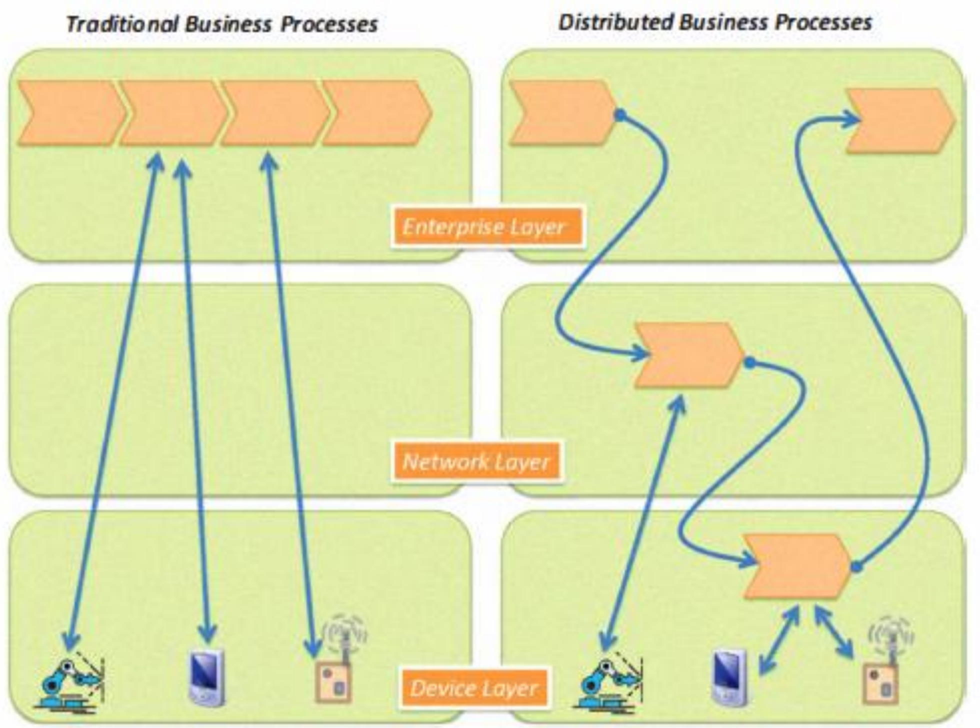

Figure 2.4. Distributed business processes. (Karnouskos & Colombo 2011)

Karnouskos and Colombo (2011) envision that the next generation SCADA system will have to cope with much higher amounts of distributed data and information In real-time by cooperating with internal and external services. Figure 2.5 depict their vision of a next generation SCADA/DCS system. The system is information driven with all interactions done via services. From the SOA viewpoint all the systems (ERP, devices, MES ect.) expose their functionality as service that can be composed by and interact with other entities. Logic is hosted where it makes sense in example near the point of action.

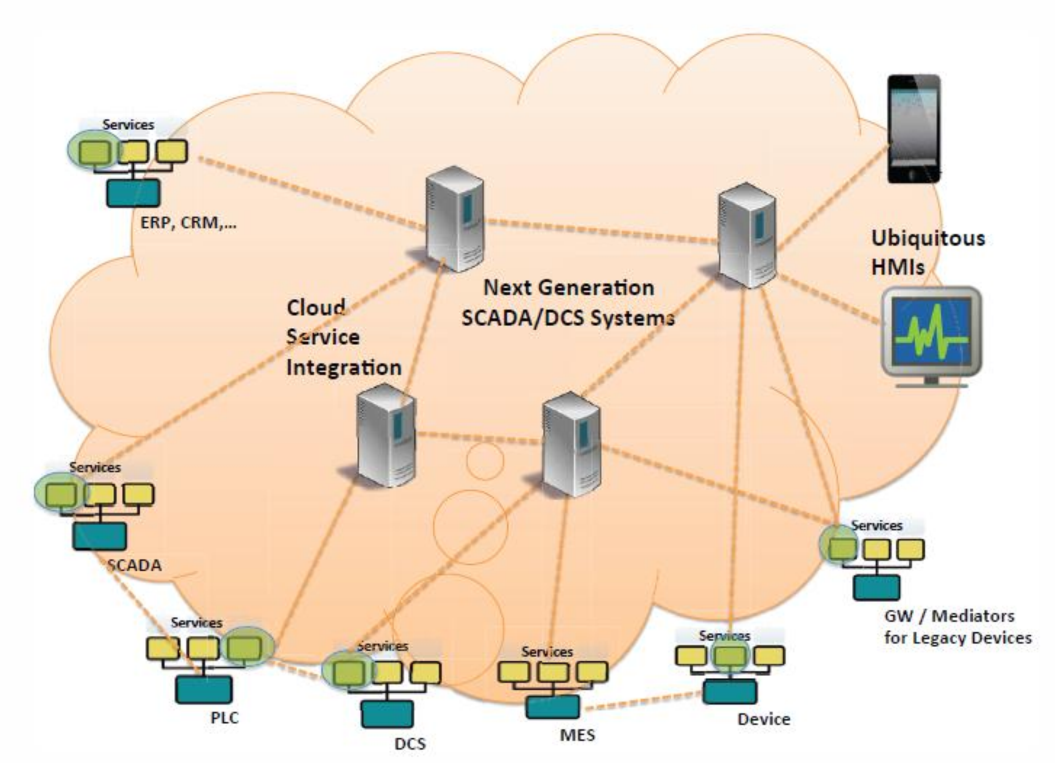

Figure 2.5. Future SCADA architecture. (Karnouskos & Colombo 2011)

The SCADA architecture envisioned in figure 2.5 is no longer a vision of the future but the state-of-the-art for SCADA systems. The figure also serves as a good example of a state-of-the-art SOA based monitoring system with different level services communicating with cloud-based applications.

Using SCADA as a part of SOA based monitoring system it performs only supervisory operations and can perform a limited amount control such as starting the process or changing measurement points from the human machine interface (HMI). Monitoring system SCADA concentrates on data acquisition and presentation. Human machine interface (HMI) is the part of SCADA that displays important information about the process to the user. According to Zhang (Zhang 2010) human machine interfaces are usually linked to databases and software of a SCADA to provide status, trending, diagnostic data and management information. Galloway and Hancke (2012) present in their journal that SCADA usually consist of two application layers, a client application which presents the HMI and server application which co-ordinates and records data being displayed by the client as well as manage communication with devices. Server is required to handle communication and open up endpoint for devices as the industrial systems become more networked. In the state of the art example presented previously figure 2.5 the servers are situated in the cloud.

A vast amount of different SCADA applications and solutions are offered by different companies. Deciding which solution is the best for a specific need can depend on anything from technological requirement to usability or costs. Many of the SCADA

software's in the market are intended for specific industry application like for process manufacturing or electric power grids.

2.3 Web Services: the backbone of SOA

The W3C group describes Web Services the following way: "Web Services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks" (W3C 2004a). Cachapa et al. find this definition too broad and uses a more hand on definition for Web Service on journals (2010a) and (2010b) as "Web Services are usually understood to those services that have their interfaces described by using the WSDL format, and communicate through SOAP formatted XML envelopes." The more specific definition from W3C for Web Service is defined as "a software system designed to support interoperable machine-to-machine interaction over a network". They also add that a Web Service has a interface described in a machine-processable format more specifically WSDL and that other systems interact with the Web service in a manner prescribed by its WSDL description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. (W3C 2004b)

2.3.1 Web service standards

Cachapa et al. (2010b) Web Service definition listed a set of different standards which the web services use to enable its operability. Web services are compliant with different web service standards which can be used to add different functionalities to the web service but are not mandatory. According to Bean (2010) when consumers and services adopt and comply with the WS-* stack of standards, there are relatively few issues with the exchange of messages and greater levels of interoperability. Bean also name four core standards for Web Services and its interface that are

- Extensible Markup Language (XML)
- XML Schemas Definition Language (XSD)
- Web Service Description Language (WSDL)
- Simple Object Access Protocol (SOAP)

XML is a mark-up language and it was created for carrying data. It is an open standard published by W3C that is widely adopted. According to W3C "XML is playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere" (W3C 2003). According to Thirumala et al. (2006) XML provides a domain and platform independent, interoperable, cost effective, open and standardized management interface. This allows the standard to be easily adopted as the format for data transmission. A single XML message can represent and transmit large amounts of data and the messages are easily generated, parsed and processed.

XSD defines the structure of an XML document. W3C describes XSD the following way "XML Schema Definition Language offers facilities for describing the

structure and constraining the contents of XML documents” (W3C 2012). XSD can be used to describe the required xml format expected from an xml file. This way a service can check that xml messages are in the right form.

WSDL was developed to describe network services as a set of communication endpoints capable of exchanging messages. WSDL file provides documentation of a specific service. It holds key information of a network service and enables other systems to easily utilize this information for communication between them. WSDL file is written in XML and contains Types, Operations, Messages, Port Types, bindings, Ports and relates endpoint information associated with a given service. (W3C 2001)

SOAP is an XML-based messaging protocol. SOAP describes the message format and a set of serialization rules for data types. This information can be used for exchanging structured and typed information between peers in a distributed system. SOAP is independent of both programming languages and operational platforms. SOAP messages can be described as one-way transmissions between endpoints on a network. The SOAP message structure is depicted in figure 2.6. A SOAP message is identified by its envelope which contains a header and a body element. The header element is an optional element that can contain extensions and information for intermediate processors. The body element contains the actual data to be transmitted. (Roshen 2009)

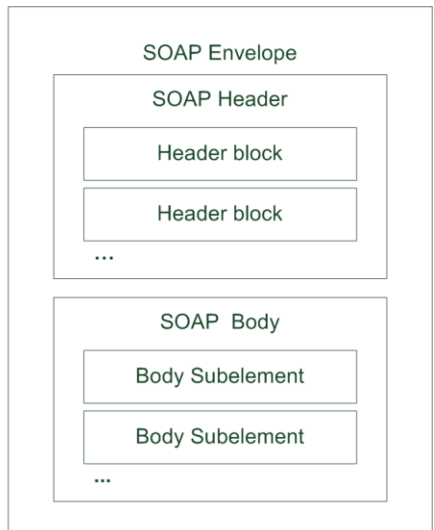

Figure 2.6. SOAP Envelope

Figure 2.7 illustrates the Web Service standard stack. As the usage of Web Services is widely adopted a number of additional functionalities have been developed and standardizes. The most widely used and mature set of Web Service standards can be seen in the Web Service stack.

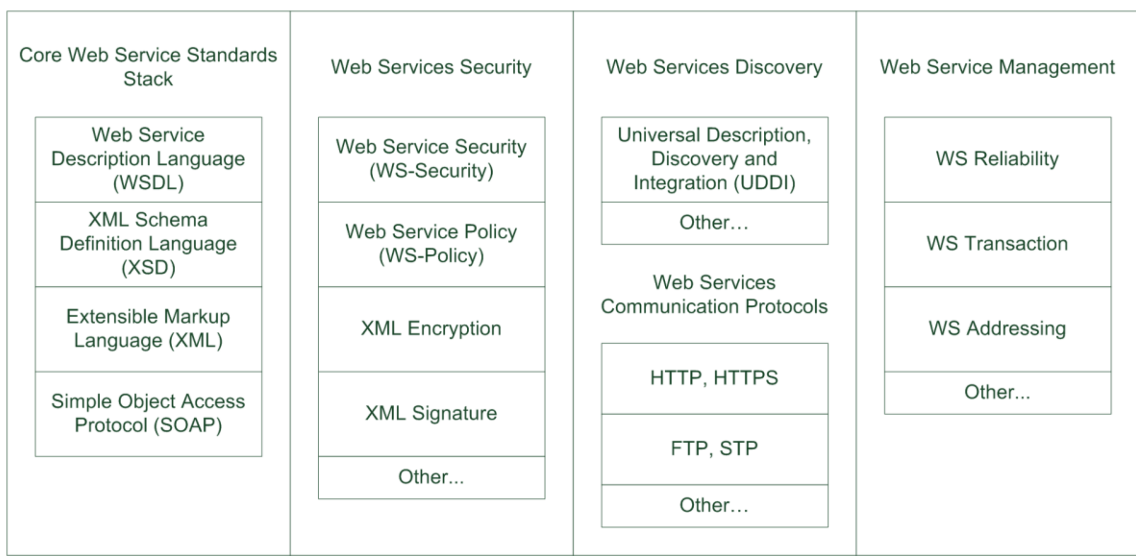

Figure 2.7. Web Service standards (Bean 2010).

The standards shown in the figure 2.7 all have a distinctive functionality from security and discovery to management. The core Web Service standards stack shows the most important standards required by Web Services. What standards are implemented on a certain Web Service implementation depends on the requirement for example for security and communication speed. For example Device Profile for Web Services uses a specific set of Web Service standards. These standards are described in the next chapter.

2.3.2 DPWS

The Device Profile for Web Services specification defines a minimal set of implementation constraints to enable Web Service messaging on resource constrained devices. DPWS supports secure messaging, discovery, description and eventing (Jammes et al. 2007). DPWS is among other SOA based approaches for device to device communication Technologies such as Open Service Gateway Initiative (OSGi) and Java Intelligent Network Infrastructure (JINI). The advantage of DPWS compared to these technologies is the reliance on web services (Zeeb et al. 2007). DPWS has vast acceptance among developed platforms, for example from Microsoft (Shodjai 2006).

DPWS was developed for bringing the SOA paradigm to the device space. It builds on core Web Service standards: WS-Addressing, WS-Discovery, WS-Transfer and WS-Eventing. It uses SOAP for messaging, WSDL for description and supports WS-Metadata, WS-Policy and WS-PolicyAttachment. DPWS is designed for resource constrained devices which is why parts of the supported standards have restricted functionality. The whole DPWS protocol stack is described in figure 2.8 and the Web Service standards used in DPWS are described in table 2.2.

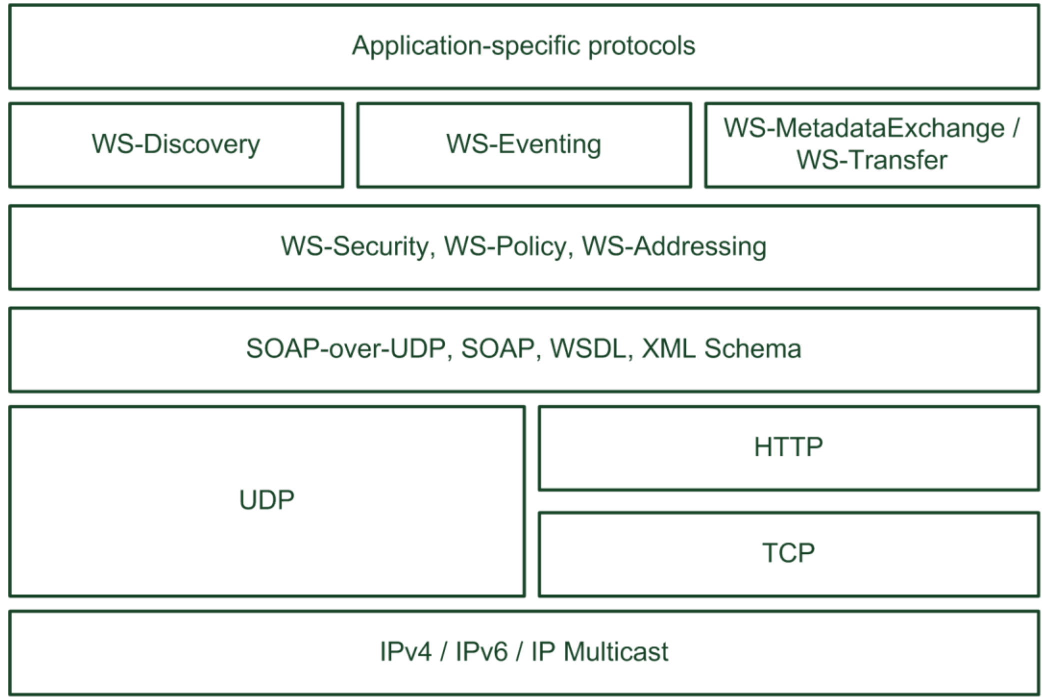

Figure 2.8. DPWS protocol stack. (Zeeb et al. 2007)

Table 2.2. Web Service standard descriptions.

SPECIFICATION	DESCRIPTION
WS-Addressing	"WS-Addressing provides transport-neutral mechanisms to address Web services and messages" (W3C 2004c). WS-Addressing defines a set of XML elements that are used for referencing the endpoint. It is also used in the event message header containing message action, destination address, sender address and message identification property. WS-addressing enables message transmission through firewalls and gateways in a transport-neutral manner. (W3C 2004c) It also overcomes the lack of SOAP's independence of underlying protocols and supports asynchronous message exchange. (Zeeb et al. 2007) According to DPWS specification a DPWS application "should rely solely on WS-Addressing 1.0." with restrictions to device identifiers (OASIS 2009a).
WS-Discovery	WS-Discovery is an IP multicast based discovery protocol to automatically locate services. The primary function of WS-discovery is a client searching for one or more target services. WS-Discovery defines four operations or messages to discover services in a network. Two one-way messages called Hello and Bye can be used to implicitly discover services and two two-way search messages called probe and resolve. Services are located from a local network using multicast discovery proto-

col based on SOAP-over-UDP.

WS-Discovery defines two discovery modes: ad-hoc and managed mode. In ad-hoc mode the client sends a probe or resolution request message to a multicast group and the target service that marches the message sends a response directly to the client. In managed mode the client sends unicast messages to a discovery proxy that has received announcement messages from target services. DPWS requires the device to support WS-Discovery and to act as the target service because if each service were to participate in discovery it could overwhelm a bandwidth limited network. (Zeeb et al. 2007; OASIS 2009a; OASIS 2009b)

WS-Transfer

"This specification describes a general SOAP-based protocol for accessing XML representations of Web service-based resources." (W3C 2006) WS-Transfer is used for acquiring XML representations of web service endpoints. This representation describes what other endpoints need to know to interact with the described endpoint (Zeeb et al. 2007). Operations contain sending and receiving representation of a given resource and creating and deleting a resource and its representation (W3C 2006). WS-metadata was used for service and device description retrieval but in the latest DPWS version of July 2009 WS-Transfer is used to retrieve the metadata. Device still needs to recognize WS-MetadataExchange GetMetadata request messages. (OASIS 2009a)

Ws-Eventing

W3C describes this specification the following way "this specification describes a protocol that allows Web Services to subscribe to or accept subscriptions for event notification messages." (W3C 2011). Event based messaging was described in chapter 2. WS-eventing enables event messaging between web services and defines a protocol for managing subscriptions. It also allows creating and deleting subscriptions, cancelling or renewing subscription and defining a preferred delivery mechanism.

WS-Eventing provides an extensible delivery mechanism for the event source and there are no limitations to the delivery mechanisms supported by this specification. Asynchronous "push" delivery is the default delivery mechanism and can be extended. In push mode the hosted service pushes notifications to the client. DPWS requires full support for WS-

Eventing. (Zeeb et al. 2007; W3C 2011; OASIS 2009a)

2.3.3 WCF

Windows communication foundation (WCF) is a framework for building service applications under .NET. “Advantage of WCF is its support for open industry standards (WS-*)”, which makes it suitable for communications between heterogeneous systems. WCF uses XML in message exchange on basis of SOAP and WSDL. Due to message based communication WCF is well suited for use in service oriented architecture. (Stopper & Gastermann 2010) Web Services protocols interoperability guide list all the specifications WCF supports. It contains parts of WS-Addressing, WS-Policy and WS-Transfer but does not contain WS-Discovery and WS-Eventing. (MS-WSPIG 2012)

2.4 Complex Event Processing: processing power

According to David Luckham the first implementation of complex event processing was developed between 1989 and 1995 to analyze event-driven simulation on a distributed architecture. Afterwards event processing has gone a long way from the first language created for complex event processing called Rapide. (Luckham 2006) Now there are many different tools for event processing, like complex event processors (CEP) and event stream processors (ESP) with many different languages. According to Garcia et al. (2011) Both CEP and ESP techniques process events using different approaches, while CEP is more interested in extracting data from patterns of events inside a so called cloud of events, ESP technique is focused in querying streams of events at high-speed and applying algorithms to the data. This thesis will concentrate only on CEP approach to event processing.

Complex Event Processing (CEP) has emerged as a new paradigm to monitor and react to continuously arriving events in real time (Mendes et al. 2009). According to Garcia et al. (2011) CEP defines “a set of tools and techniques for real-time analyzing and handling series of events that circulate at fast speed in distributed information systems”. The authors add that CEP provides an alternative to understanding, identifying, and solving problems automatically in a system. CEP is the continuous and incremental processing of event (data) streams from multiple sources based on declarative query and pattern specifications with near-zero latency (MS-CEPM 2012). The goal is to identify meaningful patterns, relationships, and data abstractions from among seemingly unrelated events and to trigger immediate response actions. CEP can be integrated as a solution to manage event driven manufacturing systems as well as information system applications, business process automation, schedule and control processes, network monitoring, and performance prediction. CEP engine provides a variety of functions such as event correlation, event extraction, event sampling, event filtering, event parsing, semantic matching, structure transformation, event enrichment, content based routing, event aggregation, event composition, event splitting, event generation, event storing,

action triggering, among others. Authors stress that it is important to highlight that all of these functions may vary from platform to platform. (Garcia et al. 2011)

CEP systems are designed to handle real time data that arrive constantly in the form of event streams. CEP queries are continuous in the sense that they are registered once and then run indefinitely, returning updated results as new events arrive. Due to low-latency requirements, CEP engines manipulate events in main memory rather than in secondary storage media. Since it is not possible to keep all events in memory, CEP engines use moving windows to keep only a subset (typically the most recent part) of the event streams in memory. CEP engines also provide the ability to define reactive rules that fire upon detection of specific patterns. Ideally, CEP engines should be able to continuously adapt their execution to cope with variations (e.g., in arrival rate or in data distributions) and should be able to scale by sharing computation among similar queries. . (Mendes et al. 2009)

The design of a CEP can wary between different implementations but the concept stays the same. Zang et al. (2008) explains the workings of a CEP by dividing it into four phases. The concept of a working CEP is described in table 2.3.

Table 2.3. Workings of a CEP simplified to four different phases.

Phase	Definition
First	Primitive events are extracted from a large volume of data.
Second	Event correlation or aggregation is performed to create business events with event operators according to specific rules
Third	Event processing of primitive or composite event to obtain their time, causal, hierarchical and other semantic relationships.
Fourth	Response to the actionable business information.

There are many different CEP solutions offered in the market with different approaches but the simplified workings of CEP described in table 2.3 holds true on most cases. Only the inner working of each phase differs.

2.5 Cloud computing

More and more companies are moving towards cloud based computing with their computing needs. Cloud computing can basically be seen as on-demand computing power accessible through the network. There are many definitions of cloud computing. Buyya et al. (2009) have defined it as follows: "Cloud is a parallel and distributed computing system consisting of a collection of inter-connected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements (SLA) established through negotiation between the

service provider and consumers". And Vaquero et al. (2009) have stated "clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized Service Level Agreements." (Buyya et al. 2011)

It is hard to give a specific definition of what cloud computing is but from these definitions it is possible to sum up that a cloud is a distributed computing system as in servers renting computing resources to customers. In addition to raw computing and storage, cloud computing providers usually offer a broad range of software services. They also include APIs and development tools that allow developers to build seamlessly scalable applications upon their services. The ultimate goal is allowing customers to run their everyday IT infrastructure "in the cloud".

Cloud computing services provided by companies like Amazon, Google and Microsoft are usually divided into three classes, according to the level of capability provided and the service model of providers. Figure 2.10 describes the three classes: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). These three layers can also be seen as architecture layers where service of a higher layer can be composed from services of the underlying layer.

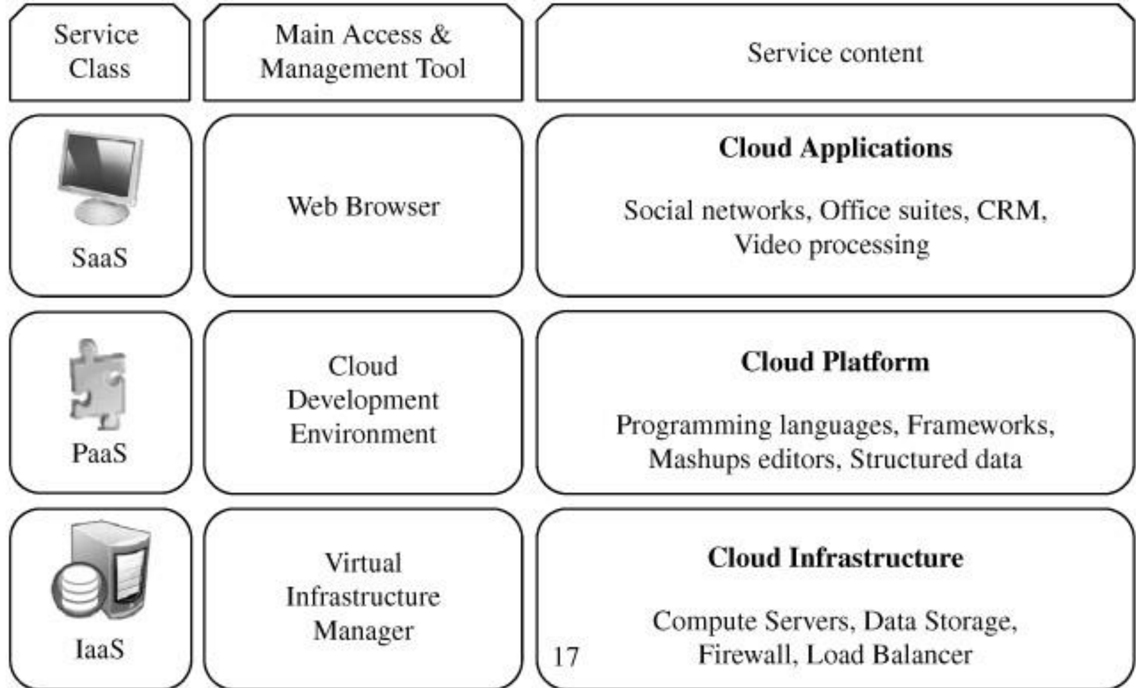

Figure 2.9. Cloud computing services (Buyya et al. 2011).

Infrastructure as a Service offers virtualized resources like computation, storage and communication on demand (Sotomayor et al. 2009). A cloud infrastructure enables

on-demand provisioning of servers running several choices of operating systems and a customized software stack. Infrastructure services are considered to be the bottom layer of cloud computing systems (Nurmi et al. 2009). Software as a Service means that the application resides on the cloud and can be accessed through a web portal. The customer does not need to install the program on to his/hers computer but can access it from any computer with a web access. An example of SaaS can be for example Google Drive that offers data storage and an office software suite for users that can be accessed with a web browser.

Platform as a service offers in addition to IaaS a platform on which to develop and deploy application. The developer does not necessarily need to know how much processing power or memory the developed application requires. IaaS also offers multiple programming models and specialized services. An example of PaaS could be Google AppEngine which offers a platform for developing and hosting web applications.

2.6 Summary

This chapter went through the main technologies associated with SOA-based monitoring by first describing what SOA stands for. Services, events and devices were described that are important parts to understand in SOA-based systems. Next the SCADA was introduced with some background on how industrial systems have evolved to the current state. This was followed by web service technologies and protocols that are used to send messages to complex event processor. The idea behind complex event processing and how it works was described before going into cloud computing.

The problem definition introduced in chapter one describes the problems associated with current complex event processors and what will be developed in this thesis. This is why it is not only important to know the functionality of CEP but also the technologies associated with the whole monitoring system. CEP is the main part of the developed monitoring system but to understand the CEP it is important to understand the whole system.

3 APPROACH

This section will outline the approach for developing the dynamic complex event processors and the monitoring system. The requirement for the dynamic CEP are defined, steps on how to design a service oriented architecture and the architecture for the solution are described. Also component descriptions for the development are presented.

3.1 Architecture

The problem of defining architecture for a monitoring system was approached using the three steps described by Colombo et al. (2010) for building a Service Oriented Architecture. First step was to identify collaborative automation units that are able to expose and consume services. Three distinct units were identified to be the process that generates the event information for the monitoring system, CEP that queries through the data identifying important data and the SCADA that gathers the processed information. These tree units are independent of each other and would host and consume services to and from each other. The basic architecture is described in figure 3.1 which highlights that the CEP is hosted on the cloud. The identified units will be used as architecture blocks used to describe different functionality throughout this thesis.

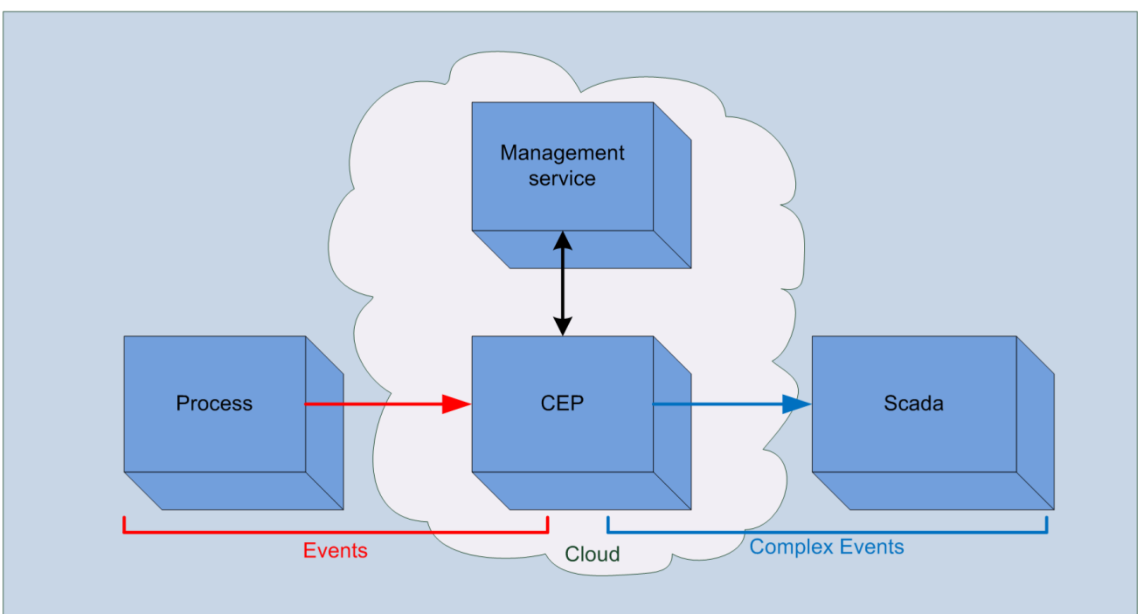

Figure 3.1. Architecture for the developed monitoring system.

Management service was added as a part of the architecture to indicate that the management service is part of CEP and will be capable of adding queries dynamically

without downtime. Figure 3.1 also shows that events arriving to the CEP are event notification messaged from the process and events leaving from the CEP are complex events that contain higher level information derived from the event notification messages. The architecture blocks introduced in figure 3.1 will be described in detail later in this chapter.

Second step for building a SOA was networking the entities together. The aim is to connect the separate devices and machines through a network and to ensure communication between them. In figure 3.2 the networking is represents. It is important to ensure that each block has network connection capability as the blocks will be connected to each other through network connections. Second step binds the blocks architecturally together.

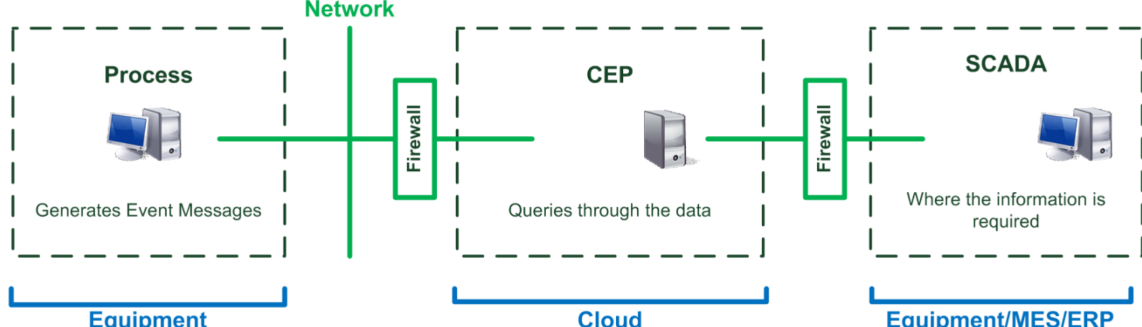

Figure 3.2. Networked representation of the architecture.

Thirds step was to make the system work for reaching a common objective. For a monitoring system the common goal is to get accurate information from the process. In this case the common goal is to get the events to flow from the process through the CEP and into the SCADA. The system should be designed so that the event can flow though the whole system without interruptions. As described in Chapter 2 the ISA-95 hierarchy is a good tool to help identify different parts of the process and clarify the information exchange between different levels. Figure 3.3 represents the different levels from a monitoring point of view and how the different blocks from the architecture fit in.

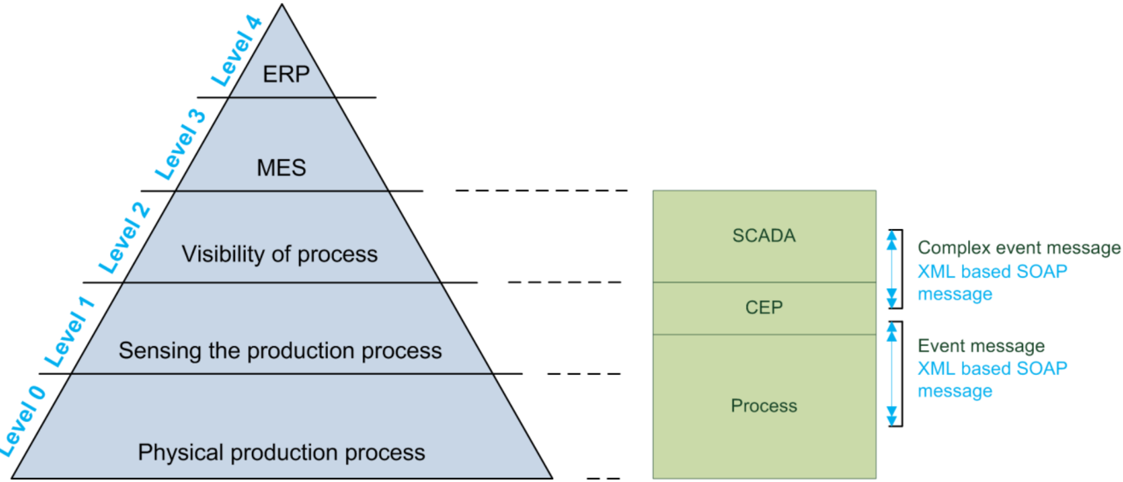

Figure 3.3. Using ISA-95 for planning the messaging.

As described in figure 3.3 all the messages exchanged between different blocks in the system are XML based SOAP messages. Even though messages are in the same format the exchange information need to be correct. SOA based systems usually share the same kind of messaging formats which simplifies building the system. By networking the blocks together and establishing communication with XML based event messaging the SOA should be complete.

3.2 Architecture blocks

Each of the architecture blocks are described separately. Architecture blocks refer to the figure 3.1 presented at the beginning of approach chapter. The blocks were identified in order to design a Service Oriented Architecture.

3.2.1 Process block

The process block describes an event producer which is in this case an industrial process. Process can be any kind of event producer where events occur and produce event notification messages. A process can be anything from a manufacturing system with hundreds of RTU's sending events to the CEP or a stock market ticker reporting changes in stock values to the CEP.

3.2.2 Complex event processor block

The concept of CEP was introduced in chapter 2.4. As described in the problem definition part of introduction, complex event processors are implementation specific and usually require recoding when changing queries. To overcome these problems a dynamic complex event processor will be developed. The main functionality of the dynamic CEP is to perform as any other CEP and query through the arriving event in order to generate complex event from the results. The main difference is that the dynamic CEP will be generic so that the same CEP can be easily adapted to different solutions,

queries can be added during run-time and the CEP can reuse query templates through the capability of using recursive user defined functions. A list of requirements are listed below that will be implemented for the dynamic CEP. Also a table of functions to be implemented by the CEP is shown on table 3.1.

Requirement for the dynamic CEP:

- Capable of receiving messages containing any event type.
- Ability to add new queries during run-time.
- Can be managed remotely with service client.
- Capability to define new Language Integrated Query (LINQ) queries.
- Capable of using recursive user defined functions.

Table 3.1. Functions that the CEP needs to implement

Name	Description
Define Query Template	The user can define a new query definition with LINQ to be used as the template when creating a new query.
Define Event Type	The user can define the event type used by the query.
Create Query	The user can create a new queries to the CEP server by using the event types and query templates.
Start Query	Starts a query.
Stop Query	Stops a query.
Remove Query	Removes the created query from the CEP server.

The functionality represented in table 3.1 will be the functions that can be invoked via Web Services using a Web Service client. Giving the possibility to define, start and stop queries on the CEP server during run-time. Event type was limited to be five data types as it allows testing with most systems and limits the required amount of coding.

3.2.3 SCADA block

As described in chapter 2.2 the focus of SCADA is data acquisition and presentation of a centralized HMI. SCADA block as a part of the monitoring system architecture is the block that presents the results to the user. SCADA in this case is not a distinctive SCADA software but some means of illustrating the monitoring system results to the

user. It can be high-end SCADA software with detailed HMI or a simple console window presenting the results from the monitored process.

3.3 Testing

Tests are to be performed to measure how the SOA based monitoring system works. By performing 8 different measurements for different scenarios it is possible to gain metrics on how the monitoring system performs under different situations. The testing scenarios will be variations of different queries and number of input events. Measurements will also be done for the Cloud hosting the CEP to gather performance measurements. The following tables 3.2 and 3.3 contain the parameters to be measured for each measurement scenario. Table 3.2 describes the parameters used to measure the CEP and table 3.3 shows the parameters that measure the clouds performance as it hosts the CEP.

Table 3.2. Measurement parameters for CEP measurements.

Parameters	Description
Total number of events	Number of events sent from the event producer.
Query passable events sent	Number of events that should pass the query.
Total number of complex events	Number of complex events that have arrived in total.
Average complex event rate per minute	Number of sent complex events per minute.
Minimum propagation delay	The minimum time it took for an event from the source to get processed as a complex event at the consumer side.
Maximum propagation delay	The maximum time it took for an event from the source to get processed as a complex event at the consumer side.
Average propagation delay	The average time it took an event from the source to get processed as a complex event at the consumer side.

By evaluating the results measured with the parameters shown in table 3.2 it should be possible to evaluate the amount of events the CEP is capable of handling, how do different queries affect the CEP performance, what kind of cloud infrastructure does the CEP require and is the monitoring system capable of performing monitoring in an industrial setting.

Table 3.3. Measurement parameters for cloud performance measurements.

Performance parameter	Description
Processor Time (%)	The measured processor time of the cloud CPU in percentages.
Commited bytes	The measured memory usage of the cloud measured in commited bytes.

The measurement parameters presented in table 3.3 should show how much the CEP requires from hardware. These will also provide information on how hardware choices affect the CEP performance as we can compare the cloud performance measurement to the CEP measurements.

4 IMPLEMENTATION

This chapter focuses on describing the developed complex event processor and the monitoring system. Chapter will go through the architecture, chosen technologies, developed CEP parts, implementations and testing.

4.1 Architecture

The following figure 4.1 describes the architecture for the developed monitoring system. More precisely it is the architecture for oil lubrication use case. The architecture follows the design from approach chapter. The technologies shown in the figure will be described in the next part of this chapter. The developed dynamic CEP can be seen as the combination of all the part residing in the Amazon cloud shown in figure 4.1.

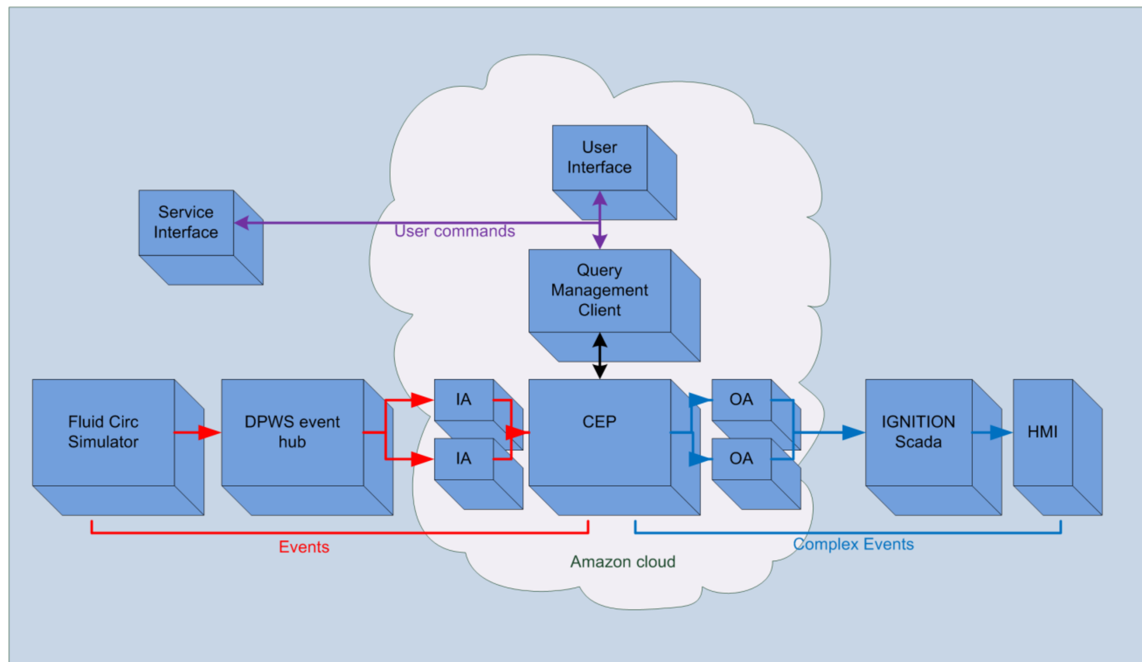

Figure 4.1. Implemented monitoring system architecture.

The architecture described in figure 4.1 describes a situation where there is two queries running as there are two input and output adapters. The different architecture blocks can be clearly seen as the cloud separates them.

4.2 Technologies

This chapter describes the technology selection and introduces the important parts of the chosen technologies. CEP technology will be introduced more thoroughly as for the reader to get a better understanding on the functionality of the developed CEP.

4.2.1 Component selection

After identifying the architectural blocks required, a set of components were selected to fulfil the requirements set by the architecture and block specific functionality. The aim was to select proven technologies to develop the monitoring system. These technologies are used to develop, host or assist the monitoring system functionality. Some components did not exist and were developed at FAST lab for this project implementation.

Table 4.1. Components selected for implementation.

Name/Company	Component	Description
StreamInsight / Microsoft	Development platform for CEP.	Microsoft StreamInsight is a platform that can be use to develop and deploy complex event processing (CEP) applications. Its mature development platform enables implementation of efficient event processing applications. (MS-SI 2012)
S1000 device / Inico Technologies	Process block, event source.	S1000 is a programmable remote terminal unit device which offers process control capabilities and web service support that is designed for industrial settings.
Ignition SCADA / Inductive automation	SCADA block, HMI.	Ignition offers SCADA, HMI and MES capabilities on a mature, web-based industrial application server.
Amazon EC2 /Amazon Web Services	Hosts CEP block.	Service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. (AMAZON 2013)
Fluid Circ simulator / FAST lab	Process block, event source.	Simulates a fluid house lubrication system. Sends event messages according to user commands.
DPWS event hub / FAST lab	Used with process blocks as event router.	Subscribes to devices and works as an event router. Routing the events from event sources to the CEP.
Complex event consumer / FAST lab	SCADA block.	Receives complex events and performs event flow calculations based on the information received.

Next the chosen technologies will be described more thoroughly. The developed components will be described under the implementations.

4.2.2 StreamInsight

StreamInsight is a product of Microsoft which was chosen to be the platform for developing the dynamic CEP because of its proven technology. The run-time component of Microsoft StreamInsight is the StreamInsight server which consists of a core engine and an adapter framework. The adapter framework allows development of interfaces to different event sources such as web services, devices, databases and to event targets such as monitoring devices, KPI dashboards and databases. The incoming events from event sources are constantly streamed into standing queries in the StreamInsight server, which processes and transforms the data according to the logic defined in each query. The query results can be used to trigger specific actions and forwarded to the event targets. The following figure 4.2 presents a high-level overview of the StreamInsight architecture. (MS-SI 2012)

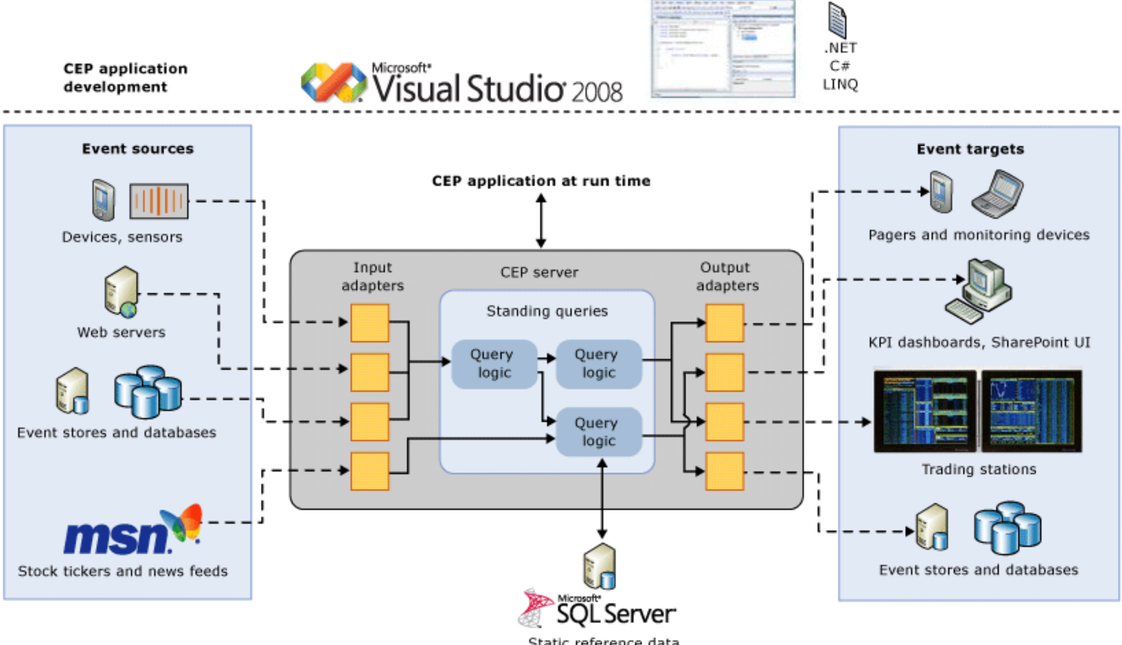

Figure 4.2. StreamInsight architecture. (MS-SI 2012)

Figure 4.2 describes the StreamInsight architecture with the CEP at the middle. The CEP server during run-time contains input and output adapters that send and receive events, while the standing queries process the events. Figure 4.2 gives the high level picture of how the CEP works in relation to event sources and event targets.

The inner workings of the StreamInsight server are described by the StreamInsight server concepts. These concepts describes the manner in which the data is represented, operated on, brought into and transferred out of the StreamInsight server and

describes data structures and server components that act on or processes data. The following table 4.2 described relevant concepts of the StreamInsight server. Full description of the server concepts can be found from the reference MS-SI 2012. (MS-SI 2012)

Table 4.2. Relevant StreamInsight server concepts. (MS-SI 2012)

Concept	**Description**
Streams	StreamInsight handles data in streams. A stream is compiled of a flow of events. Basically a queue of data in the form of events. Each event carries event payload containing event specific data. Stream data is handled only in main memory and the stream data is lost after the event has been processed. The events are not stored anywhere. The following table 4.3 shows an example of what a stream could contain.

Table 4.3. Table of stream events.

Time	MeterID	Pressure
2009-07-27 10:27:23	1	90
2009-07-27 10:27:24	1	91
2009-07-27 10:27:51	2	92
2009-07-27 10:28:52	2	94

Concept	**Description**
Events	Events are defined differently in StreamInsight than in this thesis chapter 2.1.2. In StreamInsight event is defined as the basic unit of data processed by the StreamInsight serve. Each event consists of a header and a payload. Header contains metadata associated to the event like event kind and start time. Payload contains the event data. The input adapters convert the arriving event messages into the StreamInsight event format used by the engine and add the events to the server's internal event stream. Event arriving to the StreamInsight server can arrive from multiple sources and may not always be in order according to their timestamps and have arrived out of order. StreamInsight guarantees that the query result will always be the same event if events arrive in or out of order. There are three different event models in StreamInsight: interval, point and edge. Interval and edge event models represent events whose payloads that are valid only for a given period of time. Interval events header contains the start and end time

	defining the time period. Edge event arrives with a header containing the start time with the end time arriving as a separate event. This thesis will concentrate only on point events as they represent an event occurrence of a single point in time. Point events lifetime is a single tick, which represent the shortest amount of time the StreamInsight server recognizes.
Event payload	The payload of a StreamInsight event is a .NET data structure that contains the data associated with the event. The fields in the payload are user-defined and their types are based on the .NET type system. Most CLR scalar and elementary types are supported for payload fields. Event payload field cannot contain nested types as StreamInsight engine does not support them.
Adapters	Adapters handle the event communication between the CEP and other parts of the system. Input adapters translate the incoming events into StreamInsight events and Output adapters translate them back. In this case SOAP messages arriving to input adapters are translated into events consumable by the StreamInsight engine. If a complex event is generated an event passes the query and output adapter translates this StreamInsight event back into a SOAP message. Adapter's role is to translate the messages into the right format which is why they need to know what the types of the translated variables are. Adapters can be typed or un-typed, but at runtime the adapters always emits events of one specific type. This means that event un-typed adapters are typed during runtime. If adapter is typed it means that it accepts or emits events of one particular type. Un-typed adapters provide a flexible implementation to accept the specification of event type at query bind time rather than defining the type at the time the adapter is developed. Input adapters are created to handle a specific event source. Typed output adapters are designed against a specific event payload, whereas un-typed output adapters are supplied with the event type only at runtime when the query is instantiated. Using un-typed adapter it is possible to use the same adapters with different event types.

Query templates	Query template is the query definition. Query templates contain the query definition written with a combination of LINQ and C#. LINQ is a mechanism for expressing declarative queries over data sets that is fully integrated into a host language such as C#. The query defines what information is filtered from the event stream and how that data is processed. StreamInsight server allows following functionalities for queries: • Calculations to introduce additional event properties • Filtering events • Grouping events • Windows over time • Aggregation • Identifying TOP *N* candidates • Matching events from different streams • Combining events from different streams in one • User-Defined Functions (UDF)
User-Defined Functions	User-Defined functions are used to extend the functionality of StreamInsight queries. UDF's allow using custom expressions in LINQ that call to a function that is written by the user. User can code a function and use it in the LINQ query. Restrictions are that parameters and return values of the UDF must be StreamInsight primitive types the same way as in events. It is not possible to use nested types as parameters or return values as the StreamInsight engine does not support them.
Query instances	Query instance is a composed of a query template with input and output adapter. A query instance is registered into the StreamInsight server when a query template is bound to adapters. After this the query instance can be managed in the StreamInsight server. Query instances are continuously processing data. Data arrives from the input adapter and complex events are generated according to the query template. If complex events are generated they are sent out from the output adapter. The following figure 4.3 shows the StreamInsight queries and adapters at runtime. The figure represents how input adapters receive events and forward them to query templates that are

	bound to them. Query templates process the events and possibly generate complex events which are forwarded by output adapters.

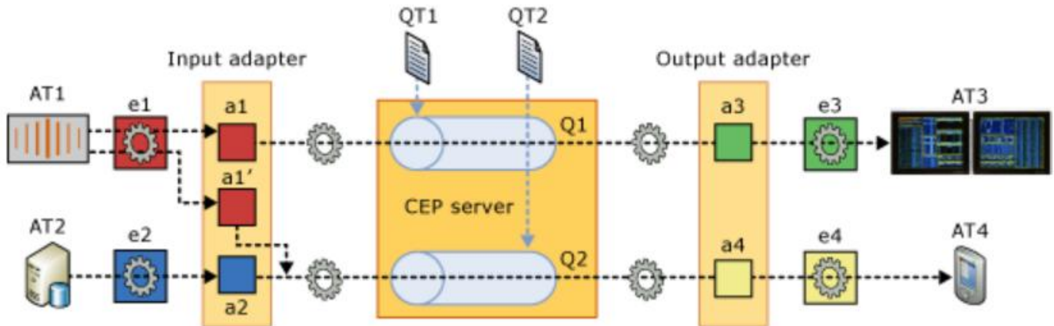

Figure 4.3. StreamInsight queries and adapters at runtime (MS-SI 2012).

Also an important part of StreamInsight is the Complex Event Processing Engine Management protocol (MS-CEPM) which is a web service protocol that defines the communication between a client application and a StreamInsight CEP server. By using this protocol, a client application can create metadata objects like adapters on a CEP server, start and stop queries, and query about the CEP system state. (MS-CEPM 2012)

The CEP engine provides a dedicated web service to handle requests from client applications for managing the system. The MS-CEPM protocol is used to communicate with the web service that is provided by the CEP engine to define and manage all of the CEP system's objects. As soon as all of the objects are defined and in place in the CEP engine, a protocol message to start the query causes the CEP engine to tap into the streaming data and to calculate and send output data. Another such message will stop the engine from recording and computing data. The MS-CEPM protocol is used to create and manage the following objects. (MS-CEPM 2012).

- Application object
- Entity object
- EventType object
- InputAdapter object
- OutputAdapter object
- Query object
- QueryTemplate object

The MS-CEPM interface called IManagementService defines methods and properties that are used to control the management functions of the Complex Event

Processing engine. Both the MS-CEPM and IManagementService interface are important when developing a client for managing the CEP engine.

4.2.3 S1000 Device

S1000 RTU is a SOA ready device. S1000 devices are used as the event source in Fastory line use case and can be seen as the Fluid Circ simulator node of figure 4.1 in oil lubrication use case. The device is developed by a Canadian company called Inico technologies. This device was used in the implementation of this thesis because Inico S1000 devices are compliant with DPWS specification and XML/SOAP interface that allows integration to SOA (Inico 2010) making it a SOA ready device. (Inico 2010). Real-time control and SOA capability is also why FluidHouse and FAST Lab uses S1000 devices in the oil lubrication measurement stations and Fastory manufacturing line.

4.2.4 Ignition SCADA

Ignition SCADA is used as the SCADA architecture block in the figure 4.1 depicted as SCADA and HMI. Ignition SCADA was chosen for this project because it is a well tested mature product with open API for module development. The possibility for module development allowed development of modules that enable communication with DPWS services and StreamInsight CEP output adapters. Ignition offers the following features:

- Web-based cross-platform SCADA software
- Mobile HMI/ SCADA clients
- SCADA designed for scalability
- Secure and stable application
- Web-based gateway configuration and HMI editor
- Integrated python scripting environment

IMC-AESOP project partner FluidHouse has been developing different monitoring implementations using Ignition SCADA with good experience which also affected why this product was chosen.

Ignition does not support event messaging by default but allows creation of custom modules. A module called WS-Module was used in this implementation to communicate with the CEP adapters. The WS-Module was developed in FAST lab and it creates a web service in Ignition that enables the SCADA to receive complex events sent from the CEP. The data received by the module can now be used in the HMI for the operator for monitoring. Figure 4.4 shows the HMI used with the monitoring system demonstration.

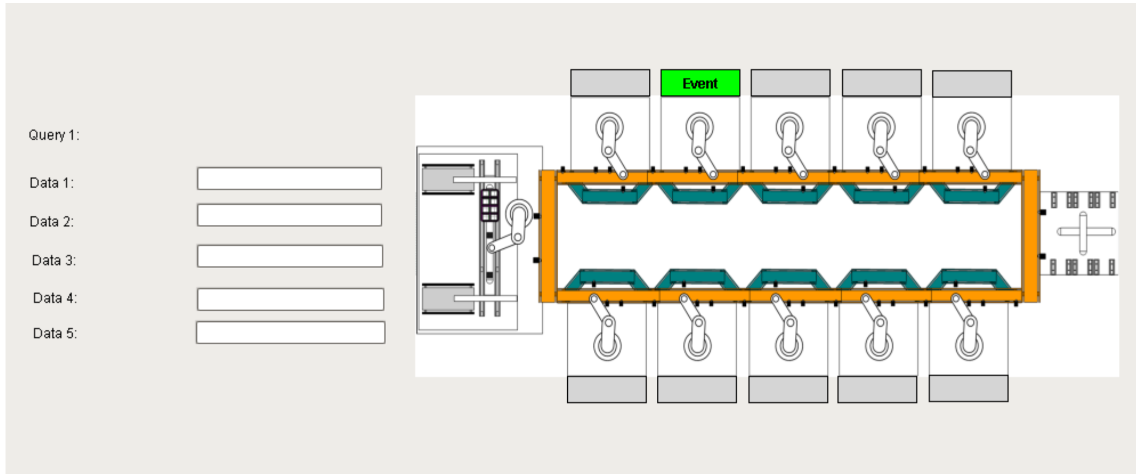

Figure 4.4 Ignition SCADA HMI for Fastory line demonstration.

Figure 4.4 shows the HMI used for testing and demonstrating the Fastory line use case. The grey boxes next to robot cells would flash green every time an event is sent. Data fields on the left would show information received from complex events. For example which pallet has completed a sequence.

4.2.5 Amazon EC2

Amazon Web services (AWS) Elastic Compute Cloud (EC2) infrastructure was chosen for hosting and testing the cloud based software implemented in this thesis. Amazon EC2 is represented as the cloud in figure 4.1. "Amazon Elastic Compute Cloud is a web service that provides resizable compute capacity in the cloud" (Amazon 2013). EC2 instance provides servers, operating system images, firewall entries, load balancers, IP addresses and disk storage volumes available instantly. (Barr 2010)

EC2 provides a web services API for provisioning, managing, and de-provisioning virtual servers inside the Amazon cloud. In other words, any application anywhere on the Internet can launch a virtual server in the Amazon cloud with a single web services call. Amazon has Data centers around the world with several in Europe. Each data center is insulated from failures in other zone. The information of the EC2 instance used can be seen in figure 4.5.

```
     Instance ID  :  i-5d339111
      IP Address  :  54.247.41.39
Availability Zone :  eu-west-1a
   Instance Size  :  m1.small
    Architecture  :  AMD64
    Total Memory  :  1.7 GB
Processing Power  :  1 ECU
  I/O Performance :  Moderate
```

Figure 4.5. Information of the EC2 instance used for the implementation.

Figure 4.5 depicts the central information related to the instance used in the implementations. Instance ID and IP address are specific to this instance. Availability zone tells which data center is in use which is in this case eu-west-1a that is situated in Ireland. Instance size is explained on figure 4.6. The elastic compute unit (ECU) depicts processing power. “One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor” (Amazon 2013). I/O performance depicts the read and write capability of the instance. I/O performance can become a bottle neck with application using and generating big databases.

Instance Family	Instance Type	Processor Arch	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	EBS-optimized Available	Network Performance
General purpose	m1.small	32-bit or 64-bit	1[1]	1	1.7	1 x 160	-	Low

Figure 4.6. Instance information of m1.small instance type. (Amazon 2013)

This thesis uses cloud technology to host the applications on the cloud for it to be accessed anywhere. Cloud based solutions could also offer scalability feature when the amount of queries or CEP instances would rise.

4.3 Complex event processor

The developed complex event processor will be described in this section. CEP is situated in the middle as the central part of the architecture described in figure 4.1. The CEP was coded using StreamInsight 2.0 and uses .NET 4.0 framework. The CEP was designed to work in pair with the Query Management Client (QMC). No query or adapter instances are initiated from the CEP. All of the query information will be added through the QMC. The CEP will be a generic CEP that can be modified for different solutions.

4.3.1 Main program

Main program is used to create a new CEP server instance that is the basis for the whole CEP. Adapters, queries and event types are normally defined in the main program but in this development they are defined using the QMC. This allows the CEP to be modified into different uses. For communicating with the QMC the main program opens up a management service endpoint of IManagementService type interface. The main program also contains the assemblies for the un-typed adapters and the user defined functions as they need to be accessible during runtime. All the other objects required for the CEP to work as a part of the monitoring system will be added through the QMC.

4.3.2 Adapters

The generic usage of CEP requires flexible adapters. This is possible by using un-typed adapters. The adapter type information is inputted into the QMC and provided to the StreamInsight server through configuration parameters during adapter bind time. This provides the StreamInsight server with data type information of the incoming messages enabling communication between the event source and CEP.

As described in the architecture (figure 4.1), CEP will receive event notification messages from the FluidCirc Simulator devices and is required to send complex event messages onward to Ignition SCADA. CEP is required to handle XML based DPWS messages from the devices and send XML messages that the SCADA recognizes. This communication is done by the adapters. Communication is achieved by using windows communication foundation (WCF) API in .NET framework for building connections between service-oriented applications. Achieving the connection between S1000 device and the input adapter requires a specific binding. The S1000 uses SOAP binding from august 2004 that is not compatible with often used bindings like WsHttpBinding that uses a newer version of the same SOAP binding. When creating the input adapter endpoint or the output adapter channel the binding needs to be defined with SOAP WS-Addressing message version from august 2004 in order get the messaging work. Output adapters use WCF to create a channel for sending messages to a specified endpoint address. Channel is WCF method for sending messages with specific bindings. Binding defines how the information being sent is wrapped into an XML message. The binding used with the adapters are custom bindings specific to the implementation. Binding are assumed to be matching in this thesis so they are not seen as limiting factors.

4.4 User-Defined functions

User-Defined functions allow creating custom functions that can be used in a LINQ query. This allows creating more complex query templates as LINQ allows only limited possibilities. In order to use the dynamic CEP in an industrial setting it requires appropriate functions that a monitoring system could require. UDF’s are coded into the CEP and thus not showing separately in figure 4.1.

Five different UDF were developed for this project. They were developed to be generic as to work with similar monitoring solutions. The UDFs are demonstrated with the Oil lubrication system and factory implementations. All UDFs will be described under this chapter in detail. They are the most important part to understand if applying the dynamic CEP into a different solution.

Motivation behind developing UDFs comes from StreamInsight supported LINQ language. LINQ is a mechanism for expressing declarative queries over data sets that is fully integrated into a host language such as C# (MS-SI 2012).The LINQ used in StreamInsight platform has been adapted for StreamInsight requirements and is a restricted set of LINQ language. All the LINQ operations available to StreamInsight can be found behind reference MS-SI 2012. Restricted LINQ results in that the normal queries made with it are simple and that a more complex query requires the use of windowing functionality. The short coming of the windowing functionality is that it does not allow complex events to be created immediately when a certain event is detected. It requires waiting to the end of the window. In order to use CEP in monitoring environment it is important that when a certain event is detected a complex event is generated immediately to keep the propagation time low. The LINQ language does provide a solution for this. It allows the possibility to use user defined functions in queries that enables creating complex queries without using windowing and makes it possible to react to events as they arrive.

4.4.1 reactDouble

reactDouble UDF was made to detect leaps in value changes. It could possibly be used to warn about sudden changes in a process or to identify sudden leaps in values still residing inside monitored tolerances. The program code for *reactDouble* is depicted in programme 4.1.

Arguments:

- *eventNameFromEvent* – String typed argument. Incoming event payload field containing the event name. Argument will be compared to the user defined event name.
- *requirementFieldFromEvent* – String typed argument. Incoming event payload field containing extra information of the event. Additional argument to be compared with requirement field defined by user. Field coming from event needs to be a string typed value or casted in the query template into string.
- *dataFromEvent* – Double typed argument. Incoming event payload field containing the double variable. Input for the monitored value.
- *eventNameFromUser* – String typed argument. Event name defined by the user. Used to filter in the events user wants to monitor.

- *requirementFieldFromUser* – String typed argument. Extra field requirement defined by the user. An additional user defined filter to filter between events of the same name.
- *scanTimeSeconds* – Integer typed argument. Defines the scan period for one event value. Defined how often the compared value will change.
- *tolerance* – Double typed argument. A user defined argument to define the tolerance for monitored value.

```
static double reactStorage; //stores the value to be compared
static bool reactStorageBool = false; //tells if a value is being stored
static DateTime reactTime; //used for scan period assignment

//ReactDouble detects if measured event field value changes more than the tolerance both ways during.
public static bool reactDouble(string eventNameFromEvent, string reguirementFieldFromEvent,
    double dataFromEvent, string eventNameFromUser, string reguirementFieldFromUser, int scanTimeSeconds,
    double tolerance)
{
    if (reactStorageBool == true && reactTime < DateTime.Now) //if scan period exceeds
    {
        reactStorageBool = false; //new compare value will be taken
    }

    //takes a new compare value and sets scan period
    if (eventNameFromEvent == eventNameFromUser && reguirementFieldFromEvent == reguirementFieldFromUser
        && reactStorageBool == false)
    {
        reactStorage = dataFromEvent;
        reactStorageBool = true; //set that there is a compare value
        reactTime = DateTime.Now;
        reactTime = reactTime.AddSeconds(scanTimeSeconds); //scan period for stored value
    }

    //compares the current event to required information and gives true if tolerances exceed
    if (eventNameFromEvent == eventNameFromUser && reguirementFieldFromEvent == reguirementFieldFromUser
        && reactStorageBool == true)
    {
        if ((reactStorage + tolerance) < dataFromEvent || (reactStorage - tolerance) > dataFromEvent)
        {
            reactStorageBool = false;
            return true;
        }
        else
        {
            return false;
        }
    }
    return false;
}
```

Programme 4.1. reactDouble UDF code.

Programme 4.1 shows the C# code of the reactDouble UDF that is situated in the main program of the CEP. reactDouble returns true value if the *dataFromFvent* has changed more than *tolerance* value.

4.4.2 comparisonIntWithName

comparisonIntWithName is a UDF developed to compare integer fields of different events with logic defined by the user. Logic supported by comparisonIntWithName are shown in table 4.4. The code is added as appendix 1.

Arguments:

- *eventNameFromEvent* – String typed argument. Incoming event payload field containing the event name. Argument will be compared to the user defined event names.
- *storedDataField* – Double typed argument. Incoming event payload field containing the double value that will be stored and compared to *comparedDataField*.
- *storedEventNameFromUser* – String typed argument. Event name of the event containing the *storedDataField* value to be stored for comparison. Argument is user defined.
- *comparedDataField* – Double typed argument. Incoming event payload field containing a double variable.
- *comparedEventNameFromUser* – String typed argument. Event name of the event containing the *comparedDataField*. Argument is user defined.
- *Logic* – String typed argument. That defined the logic for comparing stored and compared data field values.
- *scanTimeSeconds* – Integer typed argument. Defines the scan period for one event value. Defined how often the compared value will change.

Table 4.4. Supported logic.

Logic	Description
==	Equal to
<=	Less than or equal to
>=	Greater than or equal to
<	Less than
>	Greater than

Table 4.4 shows the logics that can be used in the logic parameter field of the UDF in order to compare the *storedDataField* and *comparedDataField* giving a bool value as result.

4.4.3 Recursive UDF for manufacturing monitoring

The recursive UDF consists of 3 different UDF's to complete the required functionality. Recursive UDF was developed in order to be able to add queries from the QMC without limitations to the length of the monitored sequence. UDF's are usually developed for a certain purpose and cannot be extended to other purposes without changing the source code. In this case the target was to develop a UDF that could be easily extended to other similar cases by enabling the addition of more variables with recursive functions.

The recursive UDF is constructed of *addEvent*, *sequence* and *con* UDF's. They require each other to function correctly. *addEvent* catches the data from events, *se-*

quence goes through the data trying to identify a matching sequence and *con* is used to extend *sequence* by adding inputs parameters to *sequence*. *addEvent*, *sequence* and *con* UDF's are shown on programmes 4.2, 4.3 and 4.3. Additional functions called *seqCompareField* and *seqResult* were used to simplify the UDF's. They are shown in programmes 4.5 and 4.6.

```
static Dictionary<string, List<string>> seqContainer = new Dictionary<string, List<string>>();
//addEvent gathers information from the production, checks
//if it contains it already. information stored in a dictionary
//each ID has a list of information.
public static bool addEvent(string ID, string Info)
{
        //add only info if ID already present
        if (seqContainer.ContainsKey(ID))
        {
            if (!seqContainer[ID].Contains(Info))
            {
                seqContainer[ID].Add(Info);
            }
            return true;
        }
        //add new ID if not present
        if (!seqContainer.ContainsKey(ID))
        {
            List<string> list = new List<string>();
            list.Add(Info);
            seqContainer.Add(ID, list);
            return true;
        }
        return false;
}
```

Programme 4.2. Code for *addEvent* UDF.

addEvent UDF shown in programme 4.2 uses a dictionary container to store event information. The user defines what information is inputted to the ID and Info fields. The idea is to use ID field as a constant field that does not change and the Info field as field that contains changing information. A possible case could be that the ID field is inputted with the event name field of each event and info field gets the source. This way *addEvent's seqContainer* would contain a key for each event name and each event name would have a list of sources from where the message has been sent.

```
//compares fields inputted by user. end of the sequence
public static string sequence(string compare1, string compare2)
{
    seqCompareField(compare1);
    //x indicates if sequence was used with con UDF
    if (compare2 != "x")
    {
        seqCompareField(compare2);
    }
    return segResult();
}
```

Programme 4.3. Code for *sequence* UDF.

sequence UDF shown in programme 4.3 is used in pair for *addEvent* UDF to function properly. *SeqCompareField* and *seqResult* functions that are used in *sequence* UDF require *addEvent* to fill its *seqContainer* dictionary with incoming events to be able to compare user inputted compare fields with the information received from events. This is how the recursive UDF detects sequences from the event stream.

```
//compares fields inputted by user. extension for seguence
public static string con(string compare1, string compare2)
{
    seqCompareField(compare1);
    //x indicates if compare2 was a recursive instance
    //of con UDF
    if (compare2 != "x")
    {
        seqCompareField(compare2);
    }
    return "x";
}
```

Programme 4.4. Code for *con* UDF.

con UDF shown in the programme 4.4 is used to extend the *sequence* UDF. *con* UDFs name “con” was deliberately left short of “continue” to keep the LINQ query shorter when writing long sequences. By using *con* UDF the user can continue inputting as many stings to the recursive UDF as required for a certain solution. The *con* UDF uses “x” as a return variable to indicate when compare2 field was used for invoking another instance of *con*.

```
static bool contains;
static List<string> falseList = new List<string>();
//compares input field to dictionary from addEvent
//does not return anything just fills the falseList
public static void seqCompareField(string compare)
{
    //loops through both dictionary and inner list
    //if innerlist does not contain compare field
    //then its added to false list.
    for (int i = 0; i < seqContainer.Count; i++)
    {
        var pair = seqContainer.ElementAt(i); //pair.key is the ID
        var innerContainer = pair.Value; //value is the inner list
        contains = false;
        for (int a = 0; a < innerContainer.Count; a++)
        {
            if (innerContainer[a] == compare)
            {
                contains = true;
            }
        }
        if (contains == false)
        {
            falseList.Add(pair.Key);
        }
    }
}
```

Programme 4.5. Code for *seqCompareField* function used in *sequence* and *con* UDFs

SeqCompareField function shown in programme 4.5 is used to compare the user assigned compare field value to the value in the *addEvent* container. If the value is not found from inner list containing the comparable value the ID is added to false list that tells which ID’s do not complete the sequence.

```
static bool keyFound;
//Is ran at the end of the sequence after all Con UDF's have ran.
//if ID key is not in the false list it has completed.
public static string segResult()
{
    for (int i = seqContainer.Count - 1; i >= 0; i--)
    {
        var pair = seqContainer.ElementAt(i);
        keyFound = false;
        for (int a = 0; a < falseList.Count; a++)
        {
            if (falseList[a] == pair.Key)
            {
                keyFound = true;
                break;
            }
        }
        if (keyFound == false)
        {
            seqContainer.Remove(pair.Key);//removes from dictionary
            falseList.Clear();//empties false list.
            return "Sequence complete for element ID: " + pair.Key + ".";
        }
    }
    falseList.Clear();//empties false list
    return "nothing to report";//no sequence completed
}
```

Programme 4.6. Code for *seqResult* function used in *sequence* UDF.

seqResult function shown in programme 4.6 is called last from the *sequence* UDF as it reports if any of the ID's monitored completed the user assigned sequence. *seqResult* goes through the *falseList* and if an ID is not present in the *falseList* this ID has completed the sequence.

4.5 Query Management Client

Query Management Client (QMC) is a StreamInsight client program that communicates with StreamInsight server of the CEP using the MS-CEPM protocol and resided in the cloud with the CEP. The QMC connects to the Servers management service endpoint. The managements service endpoint is basically a web service but with an added layer of security so that it can only be connected with a StreamInsight client program.

4.5.1 Query creation

Query Management Client was developed to enable definition of queries while the CEP is running. A client program with simple user interface presented in figure 4.7 was developed for users to easily define the required information for a new query instance. In order to define a query to the CEP server the client requires input and output adapter information, event type definition for incoming messages and a LINQ query definition.

This information can easily be inputted to the query management client interface to create and start a query. The client also allows some management functionality like stopping queries and removing created resources.

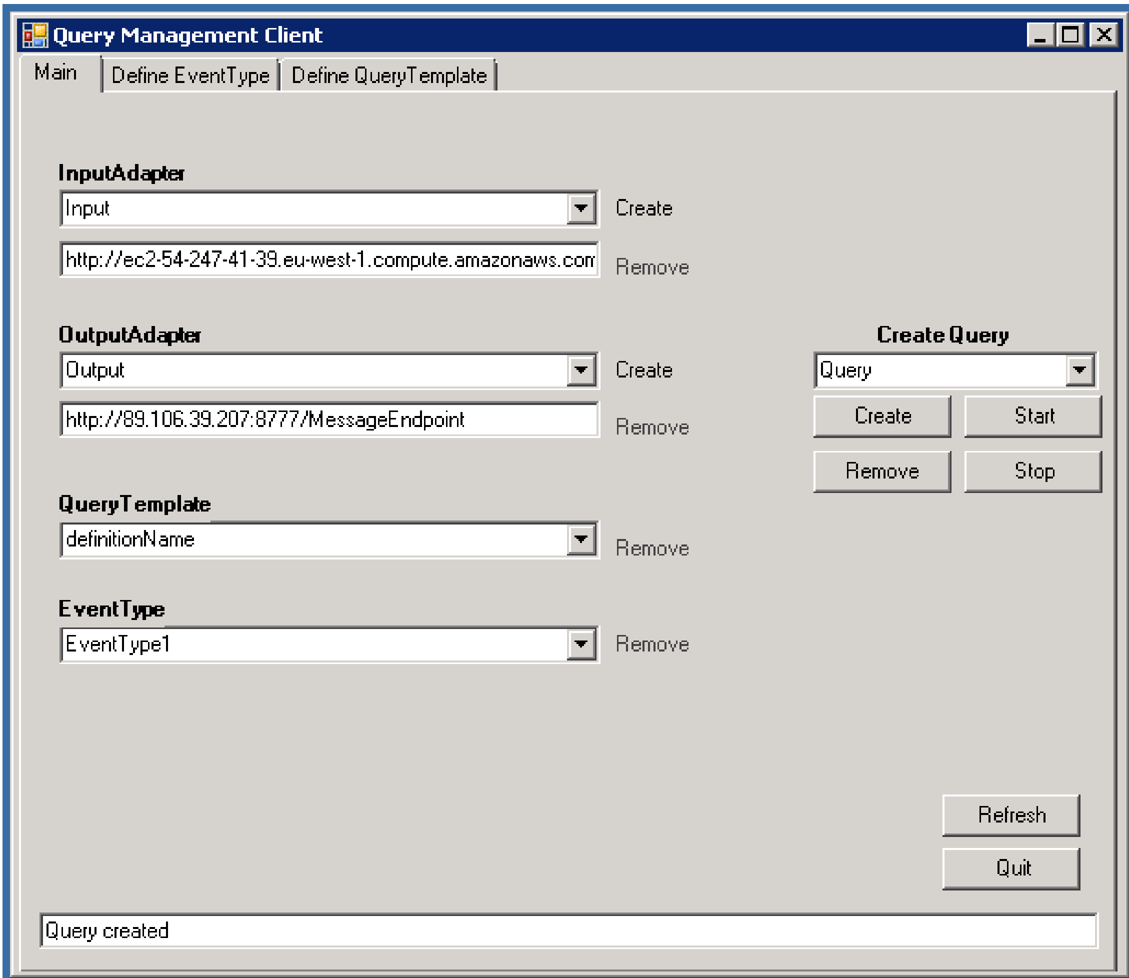

Figure 4.7. Query management client main tab.

The main tab presented in figure 4.7 allows creating and managing a query instance. The first step is to create adapter by inputting a name and an Uri. After adapters have been created the user would define an event type and a query definition from the other tabs. These would create a new QueryTemplate and an EventType. By choosing from the dropdown lists the user could choose the right combination of adapaters, QueryTemplate and an EventType for which to create a new Query onto the CEP server. The information bar at the bottom would inform that the query has been created and the user could start the query by pressing start and a new query would be running on the CEP server.

StreamInsight does not support dynamic addition of queries during run-time at default. This is because StreamInsight requires the compiled assemblies of query definitions and event types to be accessible. The user would need to type the definitions and event types to code and compile it in order to add or remove queries or event types. MS-CEPM provides the possibility to create new resources to the CEP server but still re-

quires referencing query definitions and event type instances coded to the client program. To address this problem CodeDOM .Net library was used that allows dynamic creation of classes during run-time. By using CodeDOM and the input from the user it is possible to generate source code for event types and query definitions to create new resources to the server dynamically. This enables using a simple client application to input query definition in text format. Example of using this simple interface is shown in figure 4.8 where a query LINQ definition is written to the text field and defined into an actual format that the CEP server can understand by using a parser. The example figure 4.9 presents the method of defining event types using the QMC.

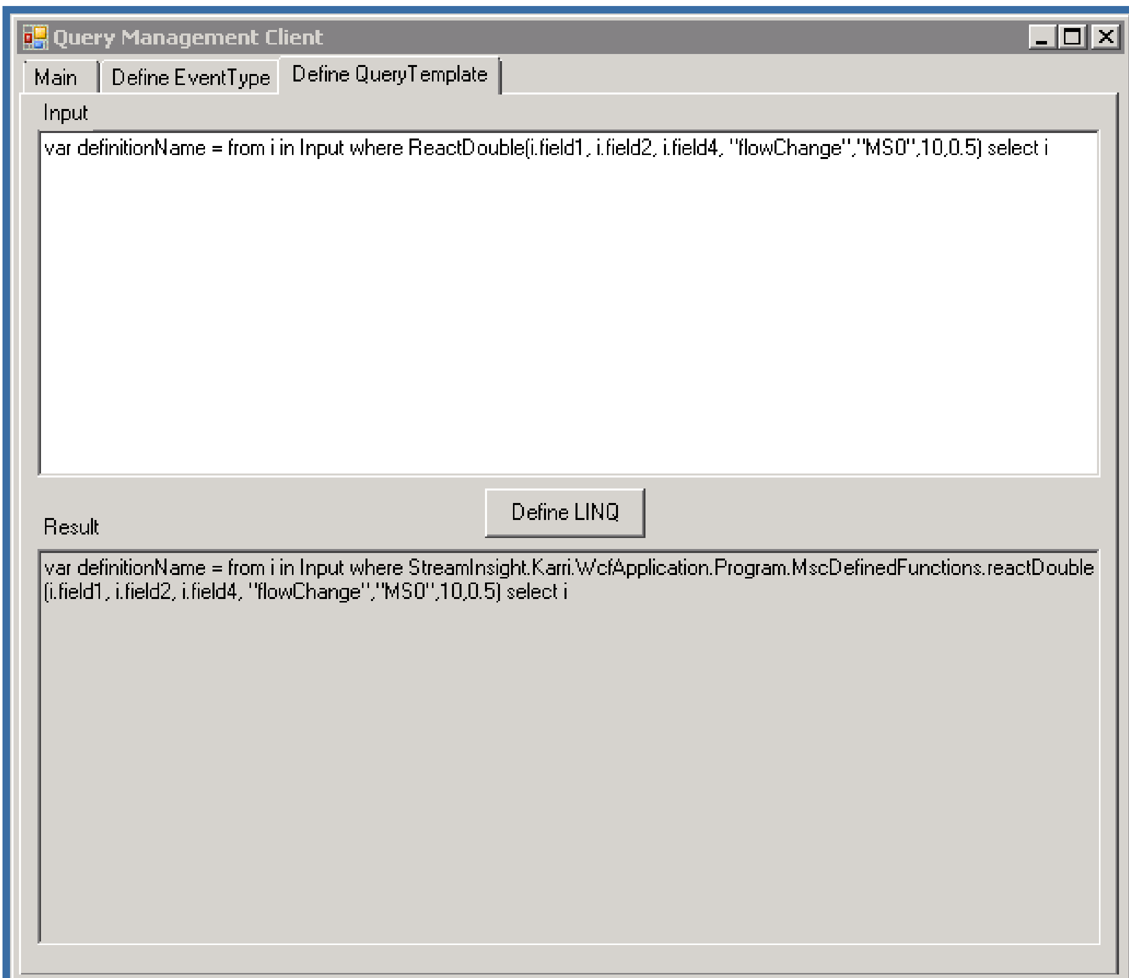

Figure 4.8. Query Management Client QueryTemplate definition tab.

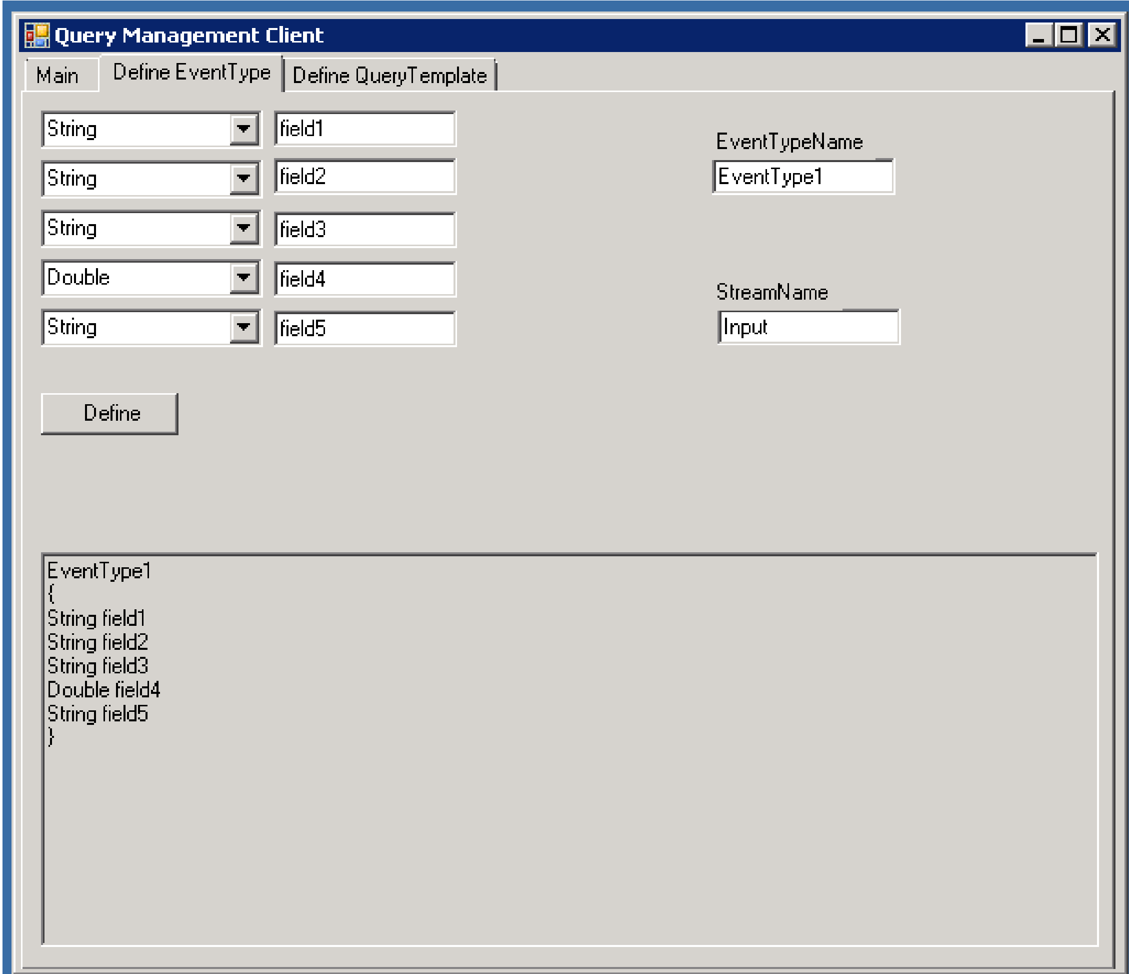

Figure 4.9. Query management client EventType definition tab.

Query definition shown in figure 4.8 shows that writing queries with QMC is no different to writing them into the actual code. At the beginning of the query definition is the name for the queryTemplate that is in this case *definitionName* which is followed by the actual query definition written in LINQ. The result field presents the parsed version of the LINQ definition containing a little more information and will be sent to the CEP server when a new query is defined. Event types are defined by selecting a type and defining a name in figure 4.9. Also a stream name is required which in this case is same as the input adapter. It is possible to have multiple queries in sequence by changing the stream names and sequencing them together.

The dynamic addition of adapters, event types and queries enables using the CEP in different solutions. This combined with recursive UDF's where the inputted parameters are not limited gives the CEP flexibility required from modern monitoring systems and enables it to be used in any solution.

4.5.2 Web Service

To be able to connect to the client and thus to the CEP a Web Service interface was added to the QMC. This allows managing the CEP by using any Web Service client tool

and connecting to the service. An endpoint is opened from the QMC to host the Web Service. The developed Web Service interface allows the following operations:

-Create input and output adapters with a specific name and URI.
-Define an eventType with specific type and name.
-Define query templates with LINQ.
-Create queries with specific input adapter, output adapter, event type, query template and name.
-Start and stop queries.
-Remove created object.

WCF Storm Web Service client was used for testing the service. The following figure 4.10. shows defining a query template using the web service.

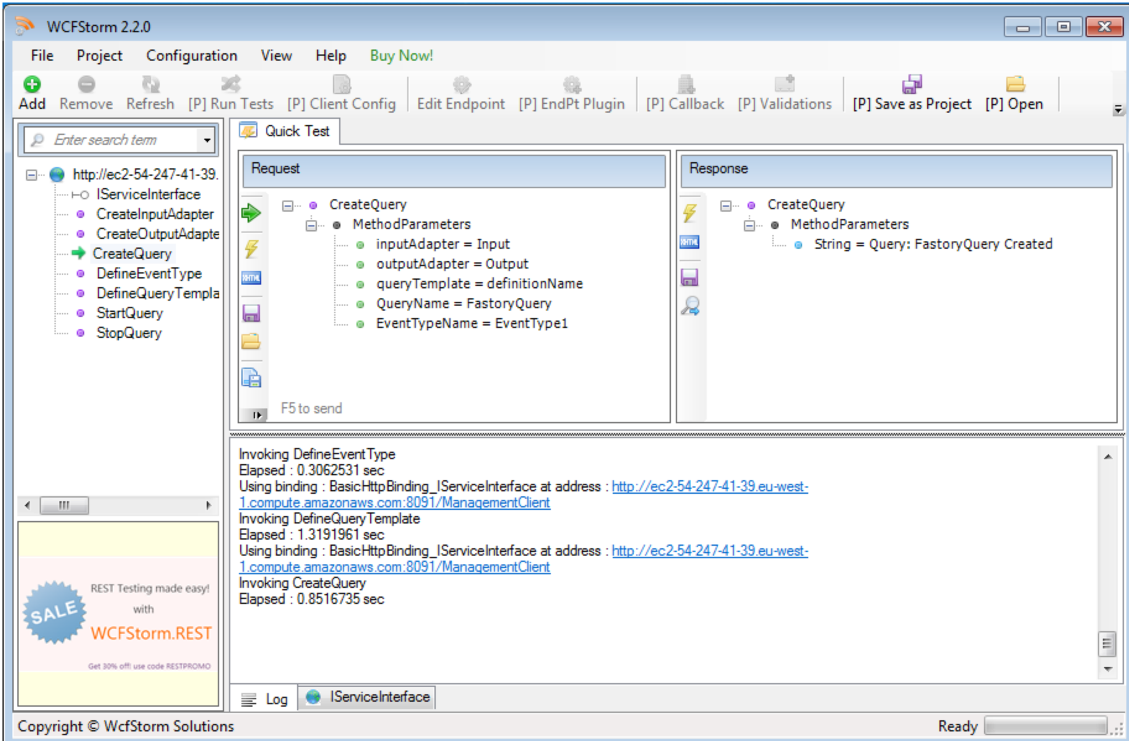

Figure 4.10. Creating a query through Web Service interface.

Figure 4.10 shows that creating new queries to the CEP is done as easily as using the QMC user interface. When creating a query through the Web Service a verification of created object will be sent as return messages.

4.6 Oil lubrication use case

The oil lubrication system is the main use case for testing. Oil lubrication system was part of the IMC-AESOP project and the reason why the dynamic CEP was developed. This chapter describes the system diagram used in this use case implementation and

introduces the important components used to get this use case working. Testing scenarios 1 to 7 were done with this use case. The testing scenarios are introduced in chapter 4.8.

4.6.1 System diagram

The following figure 4.11 describes the system diagram for oil lubrication system use case. FluidCirc simulator works as the event producer and simulates the event flow from an actual oil lubrication system.

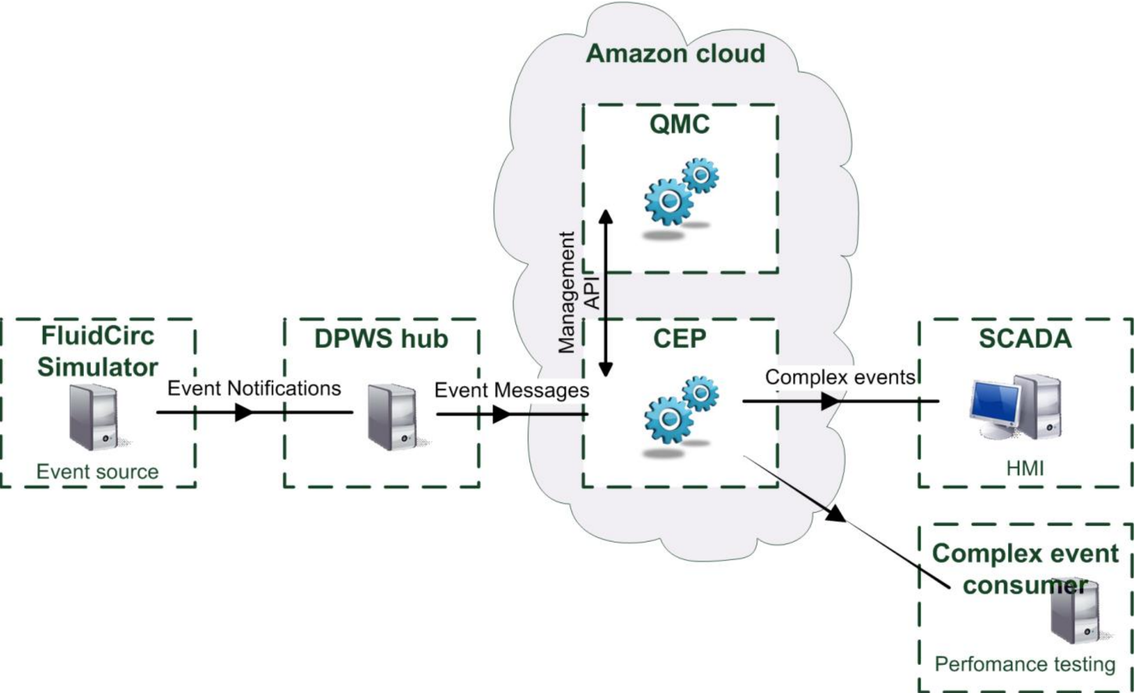

Figure 4.11. Oil lubrication system diagram.

As seen from the system diagram FluidCirc simulator and DPWS hub work as the process block generating and providing events to the CEP residing in the cloud. The Ignition SCADA and complex event consumer were used as the SCADA block. Components assisting the CEP in this monitoring system will be described next.

4.6.2 FluidCirc Simulator

FluidCirc Simulator is a simulator FAST laboratories development for FluidHouse ltd and IMC-AESOP project. FluidCirc simulator simulates the event flow of an actual fluid circulation system. The simulator is coded in java and uses camel integration framework, bootstrap front-end framework and JavaScript. FluidCirc simulators functionality and user interface is developed with these frameworks. FluidCirc simulators simulation is based on creating a set of virtual devices. There are 11 virtual devices on this simulation, one device for lubrication unit and 10 devices for measurement stations. Measurement stations hold 24 instances of flow meters each. The simulator generates events according to the default system values or according to the values inputted though

the user interface. Values correlate to each other and vary randomly within given tolerances to simulate the actual functioning of a lubrication system. The user interface allows changing all the variables affecting the simulation which allows testing different lubrication scenarios. The simulator user interface is depicted in figure 4.12. The user interface is a browser based applications.

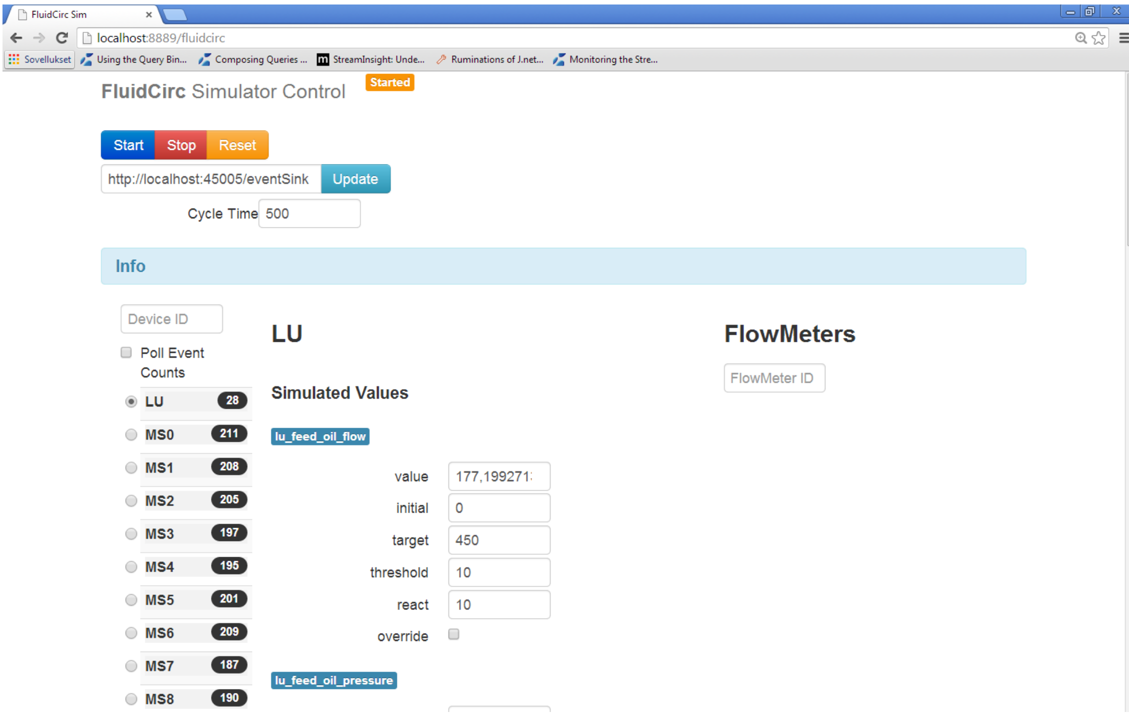

Figure 4.12. FluidCirc Simulator browser interface with measurement station selected.

As the main functionality of the simulator is to simulate the event flow of an actual lubrication system there are 13 different events sent. Lubrication unit generates 9 event notification messages and measurement stations 4 different event notification messages. All messages generated are described in table 4.5. The Flow event messages are the most prevalent event sent as each flow meter sends its own messages as its value changes. The flow event message is depicted in programme 4.7.

Table 4.5. Table of event messages sent from FluidCirc Simulator.

Lubrication unit		
Name of Event	**Parameters**	**Description**
returnTempChange	returnTempID, returnTemp	Indicates a change in the temperature of oil returning from the lubricated system to the oil container.
filterValueChange	filterValueID, filterValue	Measures the clogging of the oil filter in lubrication unit.

oilLevelChange	oilLevelID, oilLevel	Measures the oil level in the lubrication unit
reservoirOilTempChange	reservoirOilTempID, reservoirOilTemp	Tells the temperature of the oil in the oil container of the lubrication unit.
feedOilTempChange	feedOilTempID, feedOilTemp	Measures the temperature of the oil being fed into the lubricated system.
oilPressureChange	oilPressureID, oilPressure	Measures the oil pressure from the lubrication unit end.
waterContentChange	waterContentID, waterContent	Tells the water content level of the oil.
flowRateChange	flowRateID, flowRate	Measures the oil flow rate being sent from the lubrication unit.
statusChange	statusID, status	Indicates a status change from the lubrication unit.
	Measurement station	
Name of Event	**Parameters**	**Description**
flowChange	Station ID, Meter ID and Meter flow	Indicates a change in oil flow rate on a specific flow meter.
pressureChange	Station ID and Pressure	Indicates a change in oil pressure on a specific measuring station.
temperatureChange	Station ID and Temperature	Indicates a change in oil temperature on a specific measuring station
viscosityChange	Station ID and Viscosity	Indicates a change in oil viscosity on a specific measuring station

```xml
<?xml version="1.0" encoding="UTF-8"?>
<s12:Envelope
xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
<s12:Header>
<wsa:Action>
http://www.tut.fi/fast/aesop/IPointInputAdapter/InputMessage
</wsa:Action>
<wsa:MessageID>
urn:uuid:6648d070-3885-11e2-8028-d7edce9ed125
</wsa:MessageID>
<wsa:To>
http://ec2-54-247-41-39.eu-west-1.compute.amazonaws.com:8093/WcfPointInputAdapter
</wsa:To>
<wsa:From>
<wsa:Address>
http://192.168.2.62:80/dpws/ws01
</wsa:Address>
</wsa:From>
</s12:Header>

<s12:Body>
<InputMessage xmlns="http://www.tut.fi/fast/aesop">
<Data1>flowChange</Data1>
<Data2>MS6</Data2>
<Data3>FM2</Data3>
<Data4>0.017192299073596394</Data4>
<Data5>27.10.2013 17:34:55.546</Data5>
</InputMessage>
</s12:Body>
</s12:Envelope>
```

Programme 4.7. FluidCirc simulator event notification message.

The FluidCirc Simulator user interface depicted in figure 4.12 allows the user to define the event output address from the top left hand corner. The address shown in figure 4.12 is the endpoint address of the DPWS hub. Cycle time can be set from the user interface and it defines the speed at which the simulator sends new events in milliseconds. It is simply the amount of time the simulator waits before starting a new cycle of event generation form each device.

There are 11 devices as explained earlier. The lubrication unit creates 10 virtual devices for measurement stations making it a total of 11 devices. Each measurement station creates 24 instances of flow meters. As described in table 4.5 each device sends different events. Each event generation cycle goes though the same loop starting from the lubrication unit and going through each measurement station and flow meter that generates events according to the status of the device or flow meter. Each device and flow meter is an autonomous unit with its own values and generating its own events.

It can be seen from figure 4.12 that the user interface shows on the left next to the station name the amount of events sent from each measuring station. The number of event sent wary between each station as the values change a random amount and are discarded if it exceeds a certain tolerance. This is why stations generate different amount of events. It also cannot be predicted how many events a certain measuring station will generate. Single event message types were calculated using a console output for the test scenarios.

4.6.3 DPWS event hub

DPWS event hub is basically a route builder for event messages. DPWS event hub was coded with Java and uses Apache Camel integration framework for routing event messages. DPWS event hub was a separate FAST lab development. DPWS event hub was taken as a part of this implementation because .NET does not support WS-Eventing which means that StreamInsight adapters cannot subscribe to event notification messages. This is why the DPWS event hub was used to subscribe to the events and route them to the CEP adapters. DPWS event hub uses WS-Discovery to identify Web Services hosted by S1000 devices or the simulator and subscribes to their events. The events arriving from the event producers are then routed to an endpoint which is in this case the input adapter of the CEP. An example of the routing is shown in programme 4.8.

```
.setHeader("newAddress", ExpressionBuilder.constantExpression
("http://ec2-54-247-41-39.eu-west-1.compute.amazonaws.com:8093/WcfPointInputAdapter"))
.setHeader("newAction", ExpressionBuilder.constantExpression
("http://www.tut.fi/fast/aesop/IPointInputAdapter/InputMessage"))
.to("xslt:fi/tut/fast/xml/eventToInput_flowChange.xsl")
.to("log:message")
.removeHeaders("*")
.setHeader("Content-Type", constant("application/soap+xml"))
.to("jetty:http://ec2-54-247-41-39.eu-west-1.compute.amazonaws.com:8093/"
+ "WcfPointInputAdapter?bridgeEndpoint=true")
.to("log:Result")
```

Programme 4.8. DPWS event hub routing logic for event messages.

Some test scenarios used multiple queries for testing purposes. This required routing the messages to two different adapters as when using QMC each new query creates new adapters. This was done by adding an extra routing logic to the DPWS hub. Programme 4.8 also shows the usage of XLST in row 5 that changes the messages action header to the correct one to match with the binding.

4.6.4 Complex event consumer

Complex event consumer was developed for event flow and performance testing of the developed CEP. It was coded using C# and .NET framework. The program opens two service endpoints with matching interfaces for the complex events arriving from complex event processors output adapters. Complex event consumer calculates the number of arriving events, average event rate per minute, minimum propagation delay, maximum propagation delay and average propagation delay. Most of the measurement information is outputted to the console. After the measurement is finished the complex event consumer generates an excel .csv file that was used to create measurement graphs.

4.7 Fastory line use case

Fastory manufacturing line based at Tampere University of Technology Factory automation systems and technologies laboratory was used for demonstrating the monitoring system and to perform testing. Fastory line consists of 12 robot cells that were previously used for mobile phone assembly. Each cell holds a main conveyor and a bypass conveyor delivering pallets to the assembly cells. All cells have S1000 devices that are controlled by a distributed control system (DCS) that controls the assembly and pallet flow of the line. In order to demonstrate the monitoring system on this line the Recursive UDF was used to identify when each pallet has completed a sequence required for completing an assembly process. This chapter describes the use case implementation with a system diagram and introduces the event messages received from the Fastory line to help understand the query. The actual query used in this implementation is introduced in the test scenarios as scenario eight.

4.7.1 System diagram

The following figure 4.13 describes the system diagram for Fastory use case implementation. Fastory line conveyors will produce actual real world events for the monitoring system as the pallets move through the manufacturing line. The system will be run with both SCADA HMI and complex event consumer for event flow and performance metrics.

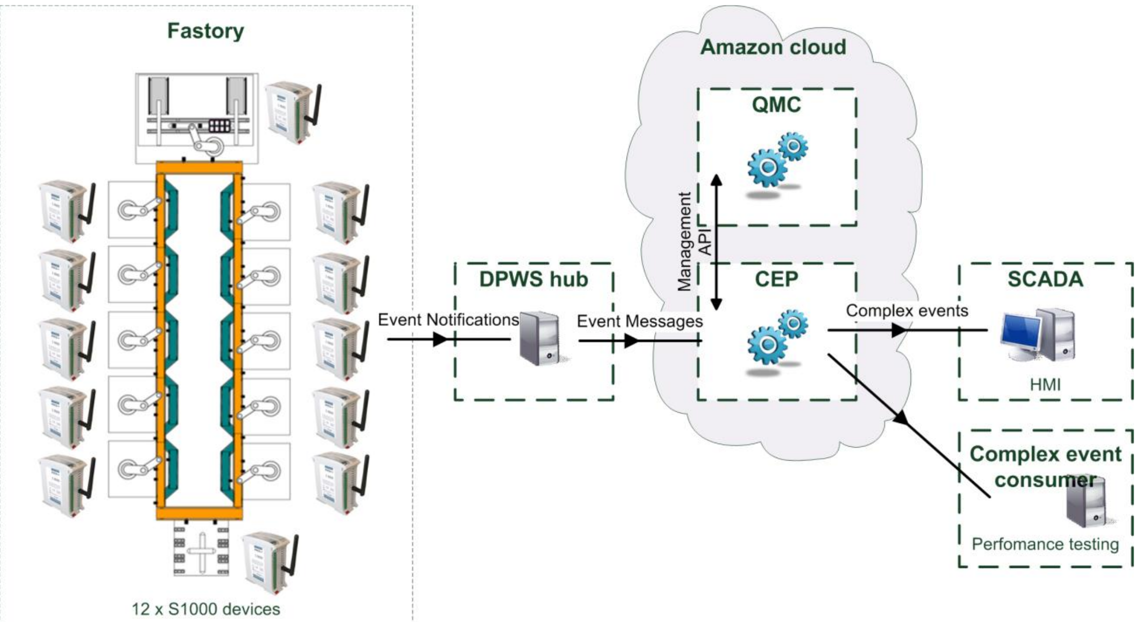

Figure 4.13. System diagram for Fastory line implementation.

Different message structures sent between each part of the monitoring system are described between the different components in figure 4.13. Structure of the system

is identical to oil lubrication system implementation with the exception of different process block generating events.

4.7.2 Fastory event messages

The events sent by the Fastory line devices were designed to work with the DCS to control the production. The event notification messages were not designed for monitoring purposes and only carry information essential for the DCS. The monitoring system can subscribe to the same messages sent to the DCS and use this data. This is a problem that arises when developing a monitoring system on top of another system and requires flexibility from the monitoring system. The following programme 4.9 shows the event notification message that arrives from the Fastory line devices.

```
<NotificationMessage xmlns="http://www.pe.tut.fi/fast/wsdl/ConveyorService"
    cellID="5"
    dateTime="2000-01-01T00:10:28.270"
    eventId="ItemTransferZoneDep"
    fromZoneId="4"
    palletId="xx"
    prodId="xx"
    rfidTag="04A25CF1D02580"
    toZoneId="5" />
```

Programme 4.9. Body part of factory event notification message.

As seen in programme 4.9 all the information carried within the notification message are set as attributes, not as individual parameters. As the input adapter needs the information in separate parameters for it to correspond to the input adapter interface an XLST transformation is done at the DPWS hub to transform the attributes into separate elements inside the message. The XSLT transformation is done within the camel routing command as seen in programme 4.8. This problem is seen as a binding problem between the Fastory line and the adapters and as binding problems are out of scope of this thesis is not considered to limit the generic nature of the system.

4.8 Experimental implementations

Experimental implementations were performed to prove the functionality of the developed CEP and to gain some performance metrics on the CEP and cloud performance. Two different set of tests will be run on two different platforms. The platforms will be introduced in this chapter followed by the test parameters and test scenarios.

4.8.1 Oil lubrication

Figure 4.14 shows the component locations and networking between the components. Component location and characteristics are described in more detail in table 4.6.

Figure 4.14. Component locations and functionality.

Table 4.6. Component location and platform characteristics.

Component	Location	Characteristics
FluidCirc Simulator	PC, FINLAND	Windows 7 (64bit), Intel i5 2x2.67 GHz processor, Memory 6 GB RAM
Event Hub		
CEP Service	Amazon Elastic Cloud Computing (EC2), IRELAND	m1.small, 1 ECU, AMD64, 1.7 GB
Query Management Service		
Complex Event Consumer	PC, FINLAND	Windows 7 (64bit), Intel i5 2x2.67 GHz processor, Memory 6 GB RAM

For the experimental implementations only the complex event consumer was used as SCADA as the HMI was not needed for the measurements. The measurements were taken with the components characterized in table 4.6 and could vary if measured with different hardware.

4.8.2 Fastory line

Figure 4.15 shows the component locations and networking between the components. Component locations and characteristics are described in more detail in table 4.7.

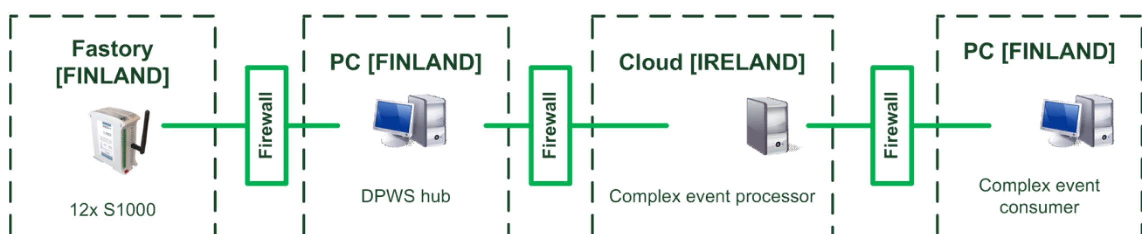

Figure 4.15. Component locations and functionality.

Table 4.7. Component location and platform characteristics.

Component	Location	Characteristics
12xS1000	Fastory Line, FINLAND	32bit CPU @ 55MHz, 8 MB flash memory
Event Hub	PC, FINLAND	Windows 7 (64bit), Intel i5 2x2.67 GHz processor, Memory 6 GB RAM

CEP Service	Amazon Elastic Cloud Computing (EC2), IRELAND	m1.small, 1 ECU, AMD64, 1.7 GB
Query Management Service		
Complex Event Consumer	PC, FINLAND	Windows 7 (64bit), Intel i5 2x2.67 GHz processor, Memory 6 GB RAM

For the experimental implementations only the complex event consumer was used as SCADA as the HMI was not needed for the measurements. The measurements were taken with the components characterized in table 4.7 and could vary if measured with different hardware.

A few timing problems were noticed with the S1000 devices during the measurements. First one occurred when all the device clocks were synchronized to the PC at the beginning of measurements. The time between each individual device and between the PC alternated. This was due to the fact that time was synchronized only ones to the PC and time did not run at the same pace on each device. As in the length of a second was different in individual devices. This would account for a big variation of propagation time between individual events and for an increasing difference in the propagation time in the long run. The second problem was that the propagation time delay between the devices and the PC was measured in multiple seconds. The timestamps of the events were several seconds old as the event arrived to the PC. The PC was connected to the S1000 devices through the same router as the S1000. This delay and was attributed to the devices as it takes several seconds between registering a time value to the event and sending a notification forward. As a result propagation delays were high on Fastory line measurements.

4.8.3 Test scenarios

The following scenarios resemble actual requirements for an industrial monitoring system. The scenarios vary by the requirements for the system in form of number of events sent, number of complex events generated and the type of the LINQ query. These scenarios were chosen to give a wide picture of different situations and requirement for the monitoring system to identify how it performs.

Table 4.8 contains all the scenarios that will be measured. The first seven scenarios are scenarios implemented with the Oil lubrications use case developed for this thesis. With Oil lubrication use case tests the main target is to measure how the dynamic CEP works with large amounts of events. Scenario eight is run on the Fastory line use case. Scenario eight will work as a proof of concept for the dynamic CEP, recursive UDFs and the whole monitoring system by testing on an actual manufacturing line.

It is important to notice that the event parameters are called by the query with a symbol. As in this case the “i” symbol representing the current event. Event parameters that are called in scenarios one to seven are listed on table 4.5. For scenario eight the parameters are listed in factory event message at programme 4.9.

Table 4.8. Test scenarios with descriptions and LINQ queries.

No.	Scenario	Description
1	**Normal LINQ query, small amount of complex events.** Production engineer wants to monitor the lubrication fluid temperatures in each measurement station in "real-time".	Query filters in all the "temperatureChange" events sent from the lubrication system. TemperatureChange has the same structure as the event in programme 4.7 presented in chapter 4.6.2 but contains Temperature values instead of flow values. **With this information filtered the engineer would get the latest temperature value from each measurement station** displayed on the HMI and also know how often the values are updated.
	LINQ Query: *var definitionName = from i in Input where i.name == "temperatureChange" select i*	
2	**Normal LINQ query, large amount of complex events**. Production engineer want to monitor each flow meter value separately to have a "real-time" view on the whole lubrication system.	This query filters in all the "flowChange" events sent from the lubrications system. Similar to the first scenario but higher number of query passable events. Each event contains the measurement station and flow meter information to identify the source. **Production engineer would have a monitoring view of each flow meter that updates as the flow rates change.**
	LINQ Query: *var definitionName = from i in Input where i.name == "flowChange" select i*	
3	**User defined function query,**	The query uses the reactDouble

	events from one station. Production engineer has already set up threshold values for the flow-Change events s/he is receiving to the SCADA but is interested in sudden changes in flow meter values. Transient state changes happening inside thresholds would not be seen on the monitoring system because of the thresholds but could be of interest for example for maintenance planning as transient states can cause vibrations to the system. Quick changes in values can indicate problems. LINQ Query: *var definitionName = from i in Input where reactDouble(i.name, i.stationID, i.meterFlow, "flowChange","MS0",10,0.5) select i*	UDF described in chapter 4.4.1 to detect quick changes in the flow meter values. **The UDF compares "flowChange" events sent from measuring station zero and looks for an increase or a decrease exceeding 0.5 in the flow value.** The tolerance of 0.5 is defined by the engineer to the query as is the event name, station name and scan time. Simulators measurement station zero flowmeter target, threshold and react parameters are set to 5 to simulate quick changes in values to achieve transient changes detectable by the query in order to obtain complex events.
4	**Two normal LINQ queries, scenario 1 and 2 combined.** The same scenarios are applied here as were applied in scenario 1 and 2. Monitoring systems require a lot of information that would result possibly in multiple queries running simultaneously on one CEP instance. LINQ Query 1: *var definitionName = from i in Input where i.name == "temperature-Change" select i* LINQ Query 2: *var definitionName2 = from i in Input2 where i.name == "flowChange"*	**Both queries are to be run on the same CEP instance.** Both queries will have individual adapters so they won't be sharing the event flow. Events are routed to separate input adapters by the DPWS event hub. Complex event consumer will host two different endpoints to receive events from both output adapters.

select i

5	**Two user defined function queries.** This scenario is similar to scenario 4 but with two UDF's. In order to compare multiple UDF's running on a CEP to multiple normal queries running on the CEP.	**Two queries are run on same the CEP instance similarly to scenario 4.** Both queries run a reactDouble UDF filtering different events. Tolerances are set to so that complex events are generated from the event stream when simulator parameters target, threshold and react are set to 5 for measurement station zero.
	LINQ Query 1: *var definitionName = from i in Input where reactDouble(i.name, i.stationID, i.temperature, "temperatureChange","MS0",10,0.01) select i* LINQ Query 2: *var definitionName2 = from i in Input2 where reactDouble(i.name, i.stationID, i.meterFlow, "flowChange","MS0",10,0.5) select i*	
6	**Additional normal LINQ query.** This scenario was added to as a middle step between scenario 1 and 3 to be able to compare the results better.	**The query helps with comparing UDF and normal LINQ queries** as the query used on this scenario filters in the same events as the UDF in scenario 3 would. In other words this scenario is the same scenario as scenario 3 but the query is written differently to be able to compare UDF queries performance compared to a normal query.
	LINQ Query: *var definitionName = from i in Input where i.name == "flowChange" && i.stationID == "MS0" select i*	

7	**Additional two normal LINQ queries.** This scenario was added to as a middle step between scenario 4 and 5 to be able to compare the results better.	**A similar case to scenario 6 where the queries would pass through the same events as in scenario 5 but the queries are written in normal LINQ.** It is important to notice that in scenario 5 and 3 the tolerances and simulator parameters were set so that all events would be noticed by the UDF looking for quick changes. In this one no changes are made to the simulator default values and the query just filters all the MS0 temperature and flow changes resulting in comparable situation.
	LINQ Query 1: *var definitionName = from i in Input where i.name == "temperature-Change" && i.stationID == “MS0” select i* LINQ Query 2: *var definitionName2 = from i in Input2 where i.name == "flowChange" && i.stationID == “MS0” select i*	
8	**User defined function query using recursive functions** Production engineer wants to monitor when each pallet has visited each production cell and are completed. In this case the production is completed when pallets have visited production cells 3, 5, 8 and 11.	This query uses the addEvent, sequence and con UDF’s to follow the production. addEvent UDF defines which variables are to be monitored and stores them. Sequence and con UDF’s go through the containers defined by addEvent to conclude if the sequence has been completed. **This allows that each pallet that has completed the sequence will generate a separate notification once the sequence is complete.** This allows up to date monitoring of production with easily modified query through the MSC.
	LINQ Query:	

var definitionName = from i in Input where addEvent(i.fromZoneID.ToString(),i.cellID.ToString()) == true select new { dateTime = i.dateTime, result = sequence("3",con("5", con("8",”11")))}

Each of the scenarios presented in table 4.6 will be measured using the measurement parameters described in the chapter 3.3. As the processing power of the cloud is limited it will be tested on how it handles different event loads. For the first 7 scenarios 3 different measurements will be performed with different event loads. Event load can be varied by changing the FluidCirc simulators cycle time. Cycle time defines the amount of time the simulator waits before sending a new set of events forward. The FluidCirc simulator and cycle time was described in chapter 4.6.2. The test measurements will be measured with 500, 2000 and 5000 as cycle times that translate into 500, 2000 and 5000 milliseconds of waiting time between incoming set of events from the simulator. Measurements are comparable as the amounts of events sent are linearly dependable. Scenario one will be measured for 5min, 30min and 24 hours measurements to detect if any anomalies appear on longer tests. Otherwise all measurements will be 5 minutes tests. The factory scenario 8 was measured for 30 minutes with 8 pallets to achieve decent amount of event flow.

Scenarios one to seven pass on the whole event arriving as a complex event if the query requirements are fulfilled. This ensures more comparable results as the event size does no vary. Scenario 8 passes on only the information required for the measurements as the event size differs from the event used in other scenarios.

5 RESULTS

The development done for this thesis resulted in a dynamic Complex Event Processor that is both capable of dynamic query addition and generic in implementation. The following results were obtained by using the dynamic CEP as a part of monitoring systems in two different implementations described under headings 4.6 and 4.7. The tests were run on two different test beds described under headings 4.8.1 and 4.8.2. This chapter will go through the obtained results. Section 5.1 will go through the results from CEP measurement parameters and section 5.2 will go through the performance results obtained from the cloud. Discussion of result is in section 5.3.

5.1 Test results for CEP measurements

The performance tests followed test descriptions outlined in chapter 3.2 and test scenarios outlined in 4.8.3. This part will go through the testing for parameters described in table 3.2 for all test scenarios. Tests were run for 5 minutes in most cases as the results did not change significantly by extending the measurement period. Scenario one was also tested for 30min and 24 hours to test for any anomalies happening with longer tests.

5.1.1 Scenario 1: Normal LINQ query, 5min test.

Scenario tests the event flow on a normal LINQ query. Table 5.1 shows the measured results for the three measured simulator cycle times. The measured average propagation delays for each complex event are shown on figure 5.1. At the beginning of measurements all parts of the system were restarted to ensure tests were identical.

Table 5.1. CEP measurement results for the 5 minute test of scenario 1.

Measured point	Result		
	Cycle time = 500	**Cycle time = 2000**	**Cycle time = 5000**
Events sent from FluidCirc simulator	24282 units	18890 units	8026 units
Query passable events sent*	127 units	115 units	112 units
Total number of complex events	127 units	115 units	112 units
Average complex event rate per minute	25.66 complex events/minute	23.50 complex events/minute	22.00 complex events/minute
Minimum propagation delay	240.38 ms	119.95 ms	136.2164 ms
Maximum propagation delay	3233.39 ms	3195.674 ms	3104.11 ms

Average propagation delay	336.8 ms	296.76 ms	320.20 ms
*Number of events sent from the event source that should pass the query defined for this scenario			

On Table 5.1 the query passable events and total number of complex events match, which means that each temperature change event got through the query and no event were lost. The average propagation delay times mach the values seen in figure 5.1. Minimum propagation delay stays well under the average.

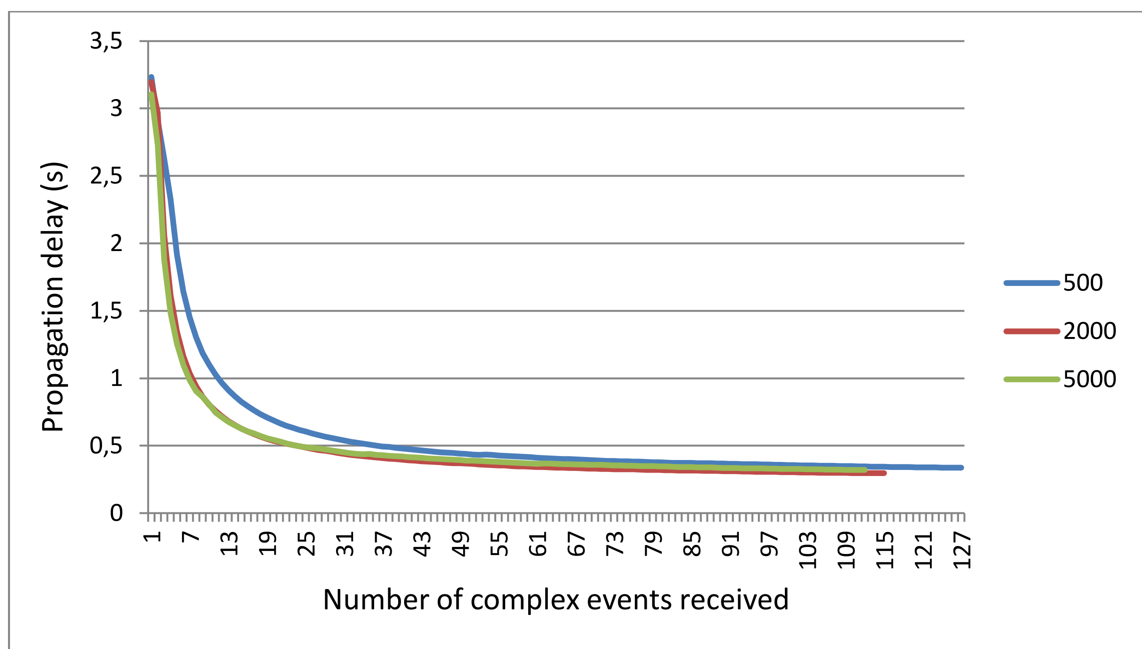

Figure 5.1. Average propagation delays for scenario 1, 5min test.

Figure 5.1 shows the average propagation delay plotted for each arriving complex event. The figure clearly shows that the first events take a considerably longer time to process as the average propagation delay starts from 3 seconds. Average values quickly reduce to under 0.5 second after approximately 30 complex events. This happens when using the query for the first time as can be seen form scenario 3 where the CEP is not reconfigured between measurements.

5.1.2 Scenario 1: Normal LINQ query, 30min test.

A thirty minute test was performed for scenario one to see if the event flow changes drastically overtime. Table 5.2 present the different results for each measured simulator cycle time and figure 5.2 shows the average propagation delays for complex events.

Table 5.2. CEP measurement results for the 30 minute test of scenario 1.

Measured point	**Result**		
	Cycle time = 500	**Cycle time = 2000**	**Cycle time = 5000**

Events sent from FluidCirc simulator	149760 units	110246 units	43906 units
Query passable events sent*	245 units	238 units	148 units
Total number of complex events	245 units	238 units	148 units
Average complex event rate per minute	10.65 complex events/minute	9.88 complex events/minute	5.80 complex events/minute
Minimum propagation delay	222.73 ms	120.91 ms	126.99 ms
Maximum propagation delay	3114.00 ms	3097.74 ms	3192.32 ms
Average propagation delay	269.33 ms	293.48 ms	290.28 ms
*Number of events sent from the event source that should pass the query defined for this scenario			

Table 5.2 shows that all the query passable events have produced a complex event and arrived to the complex event consumer. No events were missed. Average propagation delay is less than 300 ms in each case.

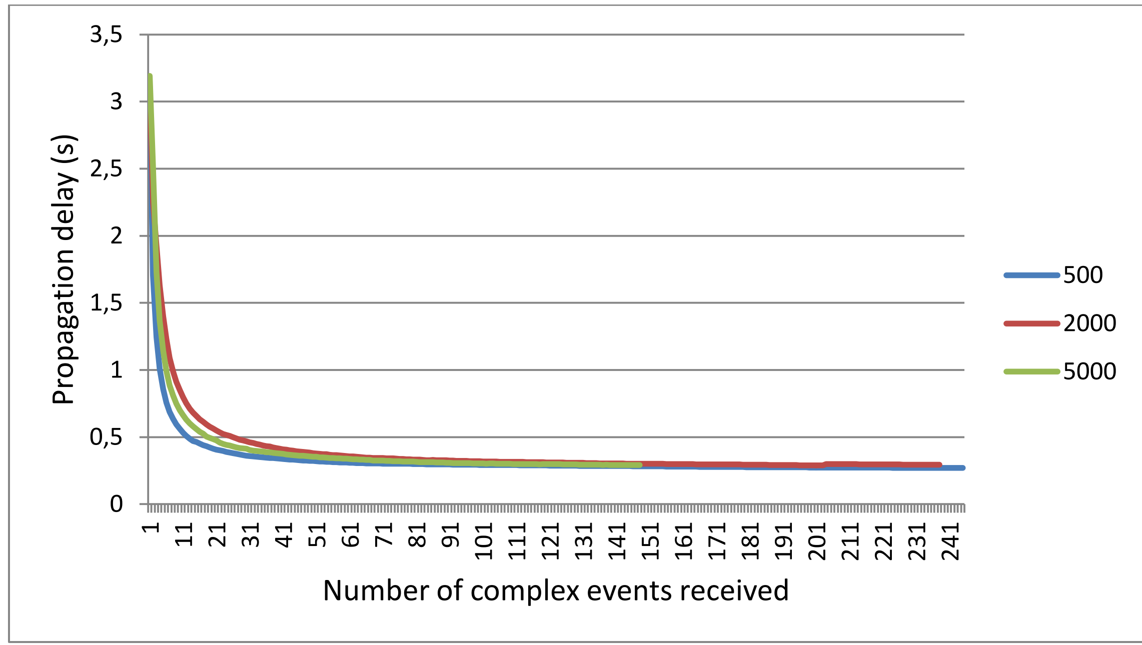

Figure 5.2. Average propagation delays for scenario 1, 30min test.

Figure 5.2 is almost identical to the five minute test figure 5.1. In both cases the propagation time reduces over time. Average values are still well above the minimum values with cycle times 2000 and 5000.

5.1.3 Scenario 1: Normal LINQ query, 24h test.

A twenty four hour test was done with scenario one to detect any changes happening in the system over a longer time. The measurement was done with cycle time of 2000.

Table 5.3 contains the measurement results and the figure 5.3 shows the average propagation delay plotted over the complex events.

Table 5.3. CEP measurement results for the 24h test of scenario 1.

Measured point	Results
	Cycle time = 2000
Events sent from FluidCirc simulator	5171854 units
Query passable events sent*	3216 units
Total number of complex events	3216 units
Average complex event rate per minute	2.24 complex events/minute
Minimum propagation delay	120.87 ms
Maximum propagation delay	3414.79 ms
Average propagation delay	264.81 ms
*Number of events sent from the event source that should pass the query defined for this scenario	

As shown on table 5.3 the number of query passable event sent and the total number of complex events received matches as in no events were missed. The number of sent events is significantly higher that other scenarios as the test lasted for 24 hours.

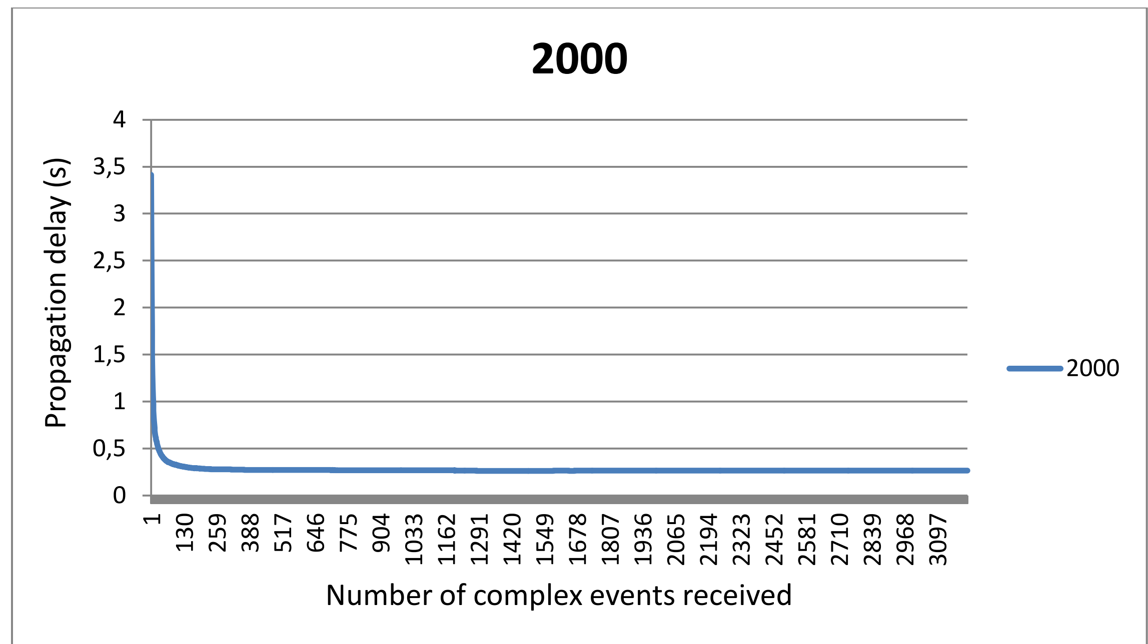

Figure 5.3. Average propagation delays for scenario 1, 24h test.

As seen from the figure 5.3 after the initial jump at the beginning the average propagation delay stays consistently the same for the entire test. A small drop in the

average propagation delay can be seen in between complex event 1291 and 1420 but the value quickly bounces back to the same line.

5.1.4 Scenario 2: Normal LINQ query, 5min test.

Scenario two was similar to scenario one but with considerably higher amount of events getting through the CEP query. Table 5.4 presents the measurement result for each simulator cycle time. Figure 5.4 depicts the average propagation times for complex events.

Table 5.4. CEP measurement results for 5 minute test of scenario 2.

Measured point	Result		
	Cycle time = 500	Cycle time = 2000	Cycle time = 5000
Events sent from FluidCirc simulator	24419 units	18821 units	7848 units
Query passable events sent*	24127 units	17668 units	7246 units
Total number of complex events	24127 units	17668units	7246 units
Average complex event rate per minute	557.53 complex events/minute	555.58 complex events/minute	555.31 complex events/minute
Minimum propagation delay	3886.58 ms	980.62 ms	884.12 ms
Maximum propagation delay	2300616.21 ms	1610338.88 ms	488431.98 ms
Average propagation delay	1153263.69 ms	811148.95 ms	252596.75 ms
*Number of events sent from the event source that should pass the query defined for this scenario			

Table 5.4 shows that query passable events are the same as total number of received complex events. Even with this amount of events no events are lost. Average complex event rate sent from the CEP stays the same for each cycle time. Propagation delays begin to increase from the beginning as more query passable events are arriving that the rate of complex event sent.

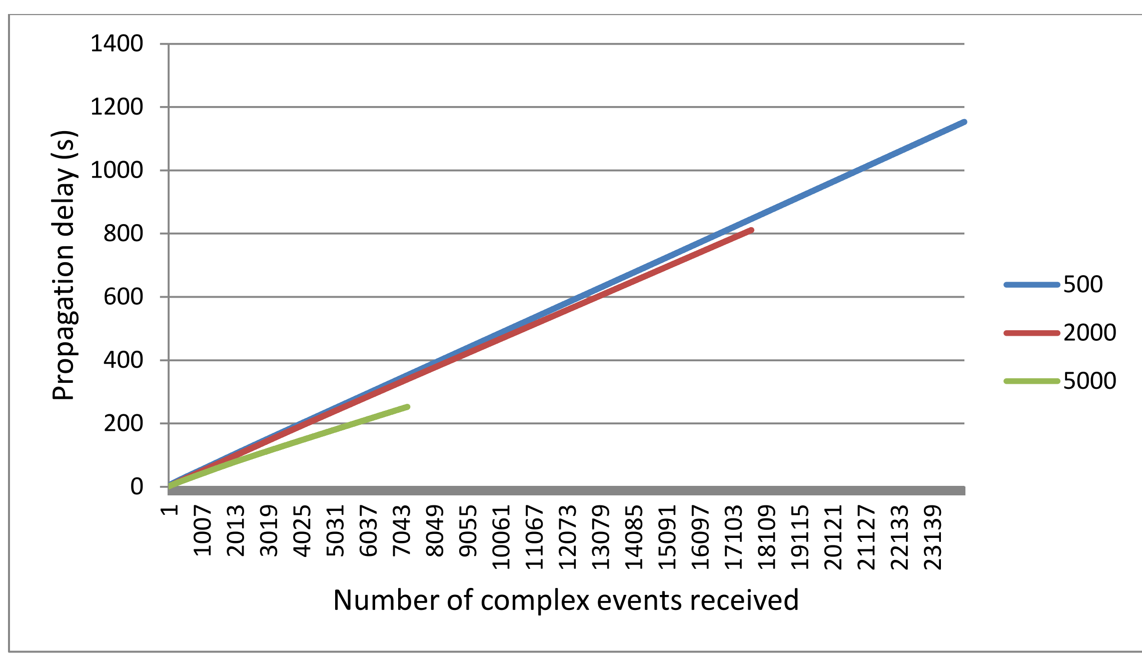

Figure 5.4. Average propagation delays for scenario 2, 5min test.

Figure 5.4 shows that the higher the rate of incoming messages the quicker the propagation delay increases. This happens due to events piling up on the input buffer as the CEP engine handles only around 555 complex events per minute. If the incoming amount of events exceeds the amount of event handled per minute the propagation time will increase as the amount of events increase. With the fastest rate at cycle time 500 the last message had a propagation delay of 38 minutes and 20 seconds which complies with the 24419 events sent and an average complex event rate of 557.53 complex events per minute.

5.1.5 Scenario 3: User defined function query, 5min test.

Scenario three used a UDF as part of the query. Table 5.5 presents the measured parameters for each simulator cycle time. Measured average propagation delays for each complex event are shown on figure 5.5. At this point of the measurements, the phase of restarting the CEP and configuring the query was not done in between measuring different cycle times to speed up the measuring process to save time.

Table 5.5. CEP measurement results for 5min test of scenario 3.

Measured point	**Result**		
	Cycle time = 500	**Cycle time = 2000**	**Cycle time = 5000**
Events sent from FluidCirc simulator	26170 units	18772 units	7763 units
Query passable events sent*	1234.4 units	880.8 units	358.7 units
Total number of complex events	1169 units	808 units	331 units
Average complex event rate	230.80 complex	164.25 complex	68.00 complex

per minute	events/minute	events/minute	events/minute
Minimum propagation delay	217.65 ms	226.89 ms	237.00 ms
Maximum propagation delay	3504.6844 ms	984.72 ms	1112.53 ms
Average propagation delay	466.73 ms	434.61 ms	495.38 ms
*Number of events sent from the event source that should pass the query defined for this scenario			

Query passable events sent is a rough estimate in this case as the query takes only the events from measuring station zero. The estimate was divided from the *flow change events sent* by twice the number of measuring stations as the UDF consumes two events in order to create one complex event. By taking into account the fact that the number of events sent by the simulated monitoring stations changes randomly. The total number of complex events fits near the estimate and taking into account the earlier scenario results it is assumed that no events were missed. The average propagation delay stays under 500ms which is almost 200ms higher compared to scenario one with a normal LINQ.

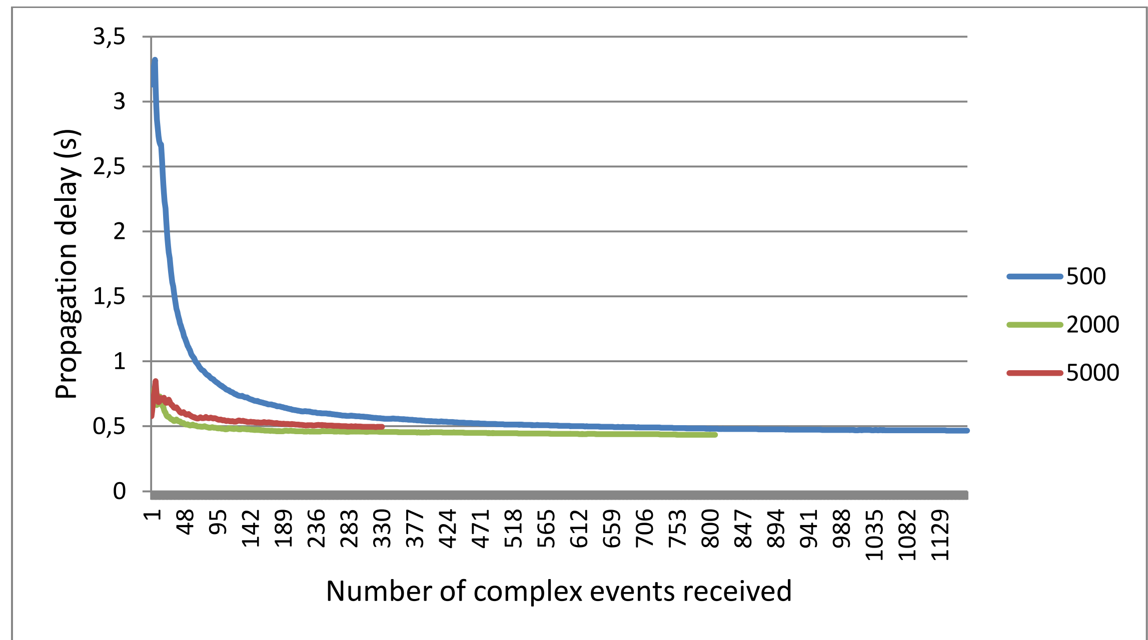

Figure 5.5. Average propagation delays for scenario 3, 5min test.

The distinct difference to earlier figure is that with cycle times 2000 and 5000 the average propagation time does not drop from 3 seconds during initial events. This may have a small affect on the average propagation delay value for 2000 and 5000 cycle time measurements compared to values of cycle time 500.

5.1.6 Scenario 4: Two normal LINQ queries, 5min test.

Scenario four combines the scenarios one and two to one CEP instance. Table 5.6 presents the measurement result for each simulator cycle time. Figure 5.6 depicts the aver-

age propagation times for complex events with each cycle time representing the combined amount of complex events from both queries.

Table 5.6. CEP measurement results for the 5min test of scenario 4.

Measured point	Result		
	Cycle time = 500	Cycle time = 2000	Cycle time = 5000
Events sent from FluidCirc simulator	25584 units	18831 units	7690 units
Query passable events sent*	24128+136 = 24264 units	17674+126 = 17800 units	7107+114 = 7221 units
Total number of complex events	24264 units	17800 units	7219 units
Average complex event rate per minute	555.53 complex events/minute	555.66 complex events/minute	563.25 complex events/minute
Minimum propagation delay	221.24 ms	125.67 ms	124.27 ms
Maximum propagation delay	2323976.59 ms	1624044.58 ms	474834.22 ms
Average propagation delay	1155041.83 ms	808224.30 ms	241165.67 ms
*Number of events sent from the event source that should pass the query defined for this scenario			

On table 5.6 query passable presents combined number of events to pass. For cycle time 500 the number of passable flow change events is 24128 and the number of temperature change events is 136 totalling 24264 that is the same as total number of complex events. With cycle time 5000 the query passable events value does not comply with the complex events received. Two events are missing.

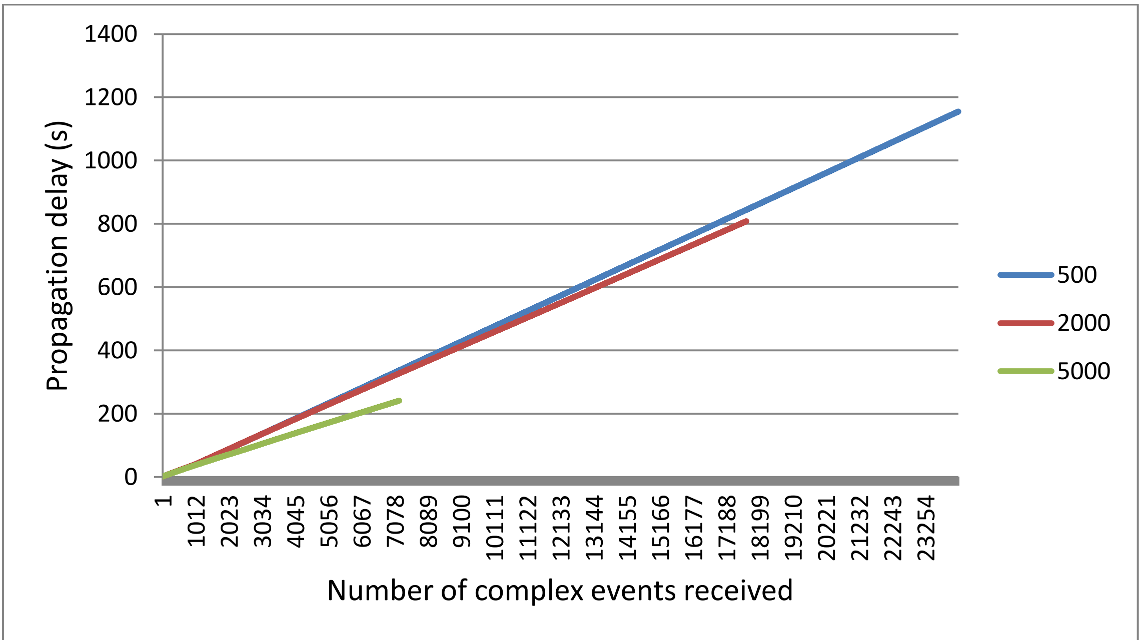

Figure 5.6. Average propagation delays for scenario 4, 5min test.

Figure 5.6 resembles the scenario two figure 5.4. In both scenarios the produced event rate was higher than the rate of produced complex events and the average complex event rate of around 555 complex events per minute, which was not affected by the amount of queries.

5.1.7 Scenario 5: Two user defined function queries, 5min test.

This scenario was implemented to test the effect of two simultaneously running UDF queries. Table 5.7 presents the CEP measurements for each simulation cycle time and the figure 5.7 depicts the average propagation delays for each complex event.

Table 5.7. CEP measurements for the 5min test of scenario 5.

Measured point	Result		
	Cycle time = 500	**Cycle time = 2000**	**Cycle time = 5000**
Events sent from FluidCirc simulator	25570 units	18499 units	7926 units
Query passable events sent*	1198.8+13 = 1211.8 units	868.1+12 = 880.1 units	364.95+12 = 376.95 units
Total number of complex events	1164 units	824 units	360units
Average complex event rate per minute	235.25 complex events/minute	169.75 complex events/minute	72.25 complex events/minute
Minimum propagation delay	224.76 ms	219.98 ms	234.69 ms
Maximum propagation delay	3272.75 ms	788.51 ms	845.54 ms
Average propagation delay	455.69 ms	439.36 ms	487.48 ms
*Number of events sent from the event source that should pass the query defined for this scenario			

As was in scenario three the query passable events sent values are rough estimates based on total amount of event send divided by twice the number of measurement stations because of UDF consuming two events and it is assumed that no events were missed. As before first cycle times average propagation delay is slightly affected by the initial delay when running the query the first time.

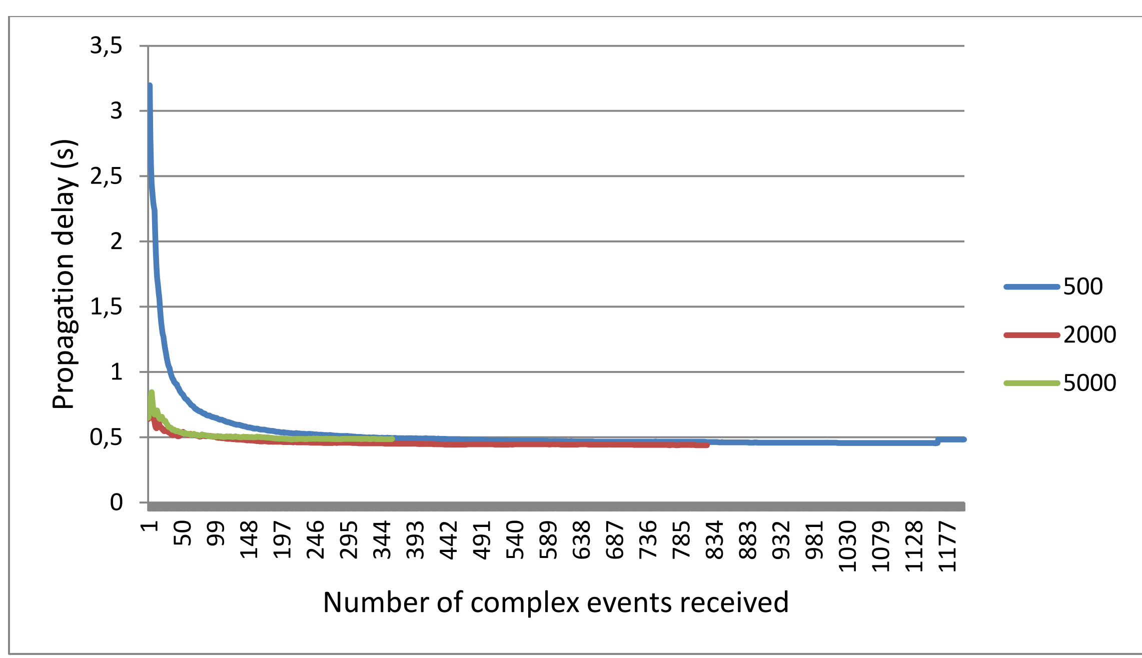

Figure 5.7. Average propagation delays for scenario 5, 5min test.

The same effect of not restarting CEP can be seen on figure 5.7. In all cases the propagation delay time settles quickly to average propagation delay. A small rise in the average propagation delay can be seen for cycle time 500 at the end.

5.1.8 Scenario 6: Additional normal LINQ query, 5min test.

Scenario six was made as a middle step to compare scenarios one and three. Table 5.8 presents the CEP measurements for each simulation cycle time and figure 5.8 depicts the average propagation delays as events accumulate.

Table 5.8. CEP measurements for the 5 minute test of scenario 6.

Measured point	**Result**		
	Cycle time = 500	**Cycle time = 2000**	**Cycle time = 5000**
Events sent from FluidCirc simulator	25348 units	18832 units	7960 units
Query passable events sent*	2394.9 units	1767.9 units	735.6 units
Total number of complex events	2374 units	1730 units	715 units
Average complex event rate per minute	475.25 complex events/minute	341.75 complex events/minute	144.25 complex events/minute
Minimum propagation delay	230.02 ms	235.77 ms	232.38 ms
Maximum propagation delay	4653.45 ms	2301.40 ms	2144.94 ms
Average propagation delay	950.11 ms	831.11 ms	856.94 ms
*Number of events sent from the event source that should pass the query defined for this scenario			

Also in this case the query passable is a rough estimate acquired by simply dividing the sent amount of flow events by twice the number of measurement stations. For example 23949 flow change events were sent with cycle time 500. As amount of sent events fluctuate between stations it is fair to say that the total number of complex events is close enough to estimate that no events were lost.

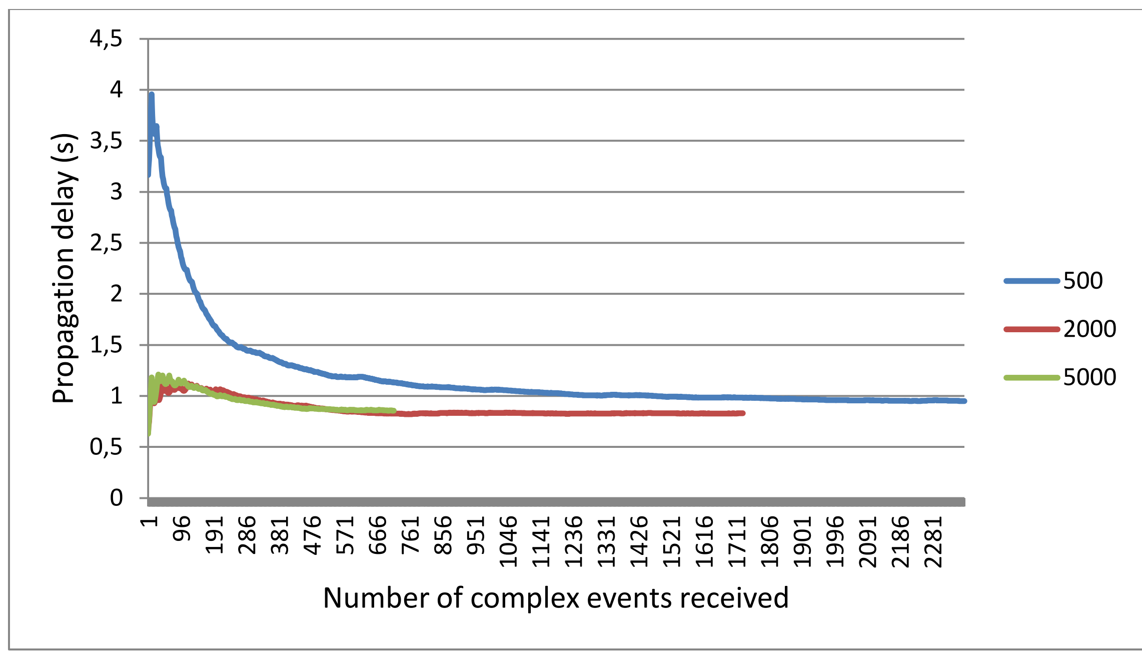

Figure 5.8. Average propagation delays for scenario 6, 5min test.

By not considering the initial leap in 500 cycle time case the average propagation delay values stay on both sides of 900ms. With cycle times 2000 and 5000 both get minimum propagation delays of 230ms and maximum propagations delays of over 2000ms. Even though these values resemble the values in scenario three the average propagation is almost 400ms higher in this case which is noticed by comparing figures 5.8 and 5.5.

5.1.9 Scenario 7: Additional two normal LINQ queries, 5min test.

This scenario was implemented to ease the comparison of scenarios four and five. The scenario is the same as scenario six but with two queries. Table 5.9 contains the CEP measurement values and figure 5.9 depicts the average propagation delays as events accumulate.

Table 5.9. CEP measurements for the 5min test of scenario 7.

Measured point	Result		
	Cycle time = 500	Cycle time = 2000	Cycle time = 5000
Events sent from FluidCirc simulator	25839 units	18761 units	7789 units

Query passable events sent*	2606 units	1869 units	745 units
Total number of complex events	2498 units	1773 units	707 units
Average complex event rate per minute	499.5 complex events/minute	354.50 complex events/minute	142.5 complex events/minute
Minimum propagation delay	222.68 ms	225.84 ms	233.19 ms
Maximum propagation delay	4687.92 ms	2523.20 ms	2182.31 ms
Average propagation delay	1080.22 ms	830.69 ms	834.65 ms
*Number of events sent from the event source that should pass the query defined for this scenario			

In table 5.9 the query passable events sent is an estimate as was for scenario six it is estimated that no events were lost. In this case it has both flow change and temperature change events combined. The important part to notice is that the average propagation delays are a little bit higher than scenario six at least for cycle time 500 and are almost double compared to scenario five.

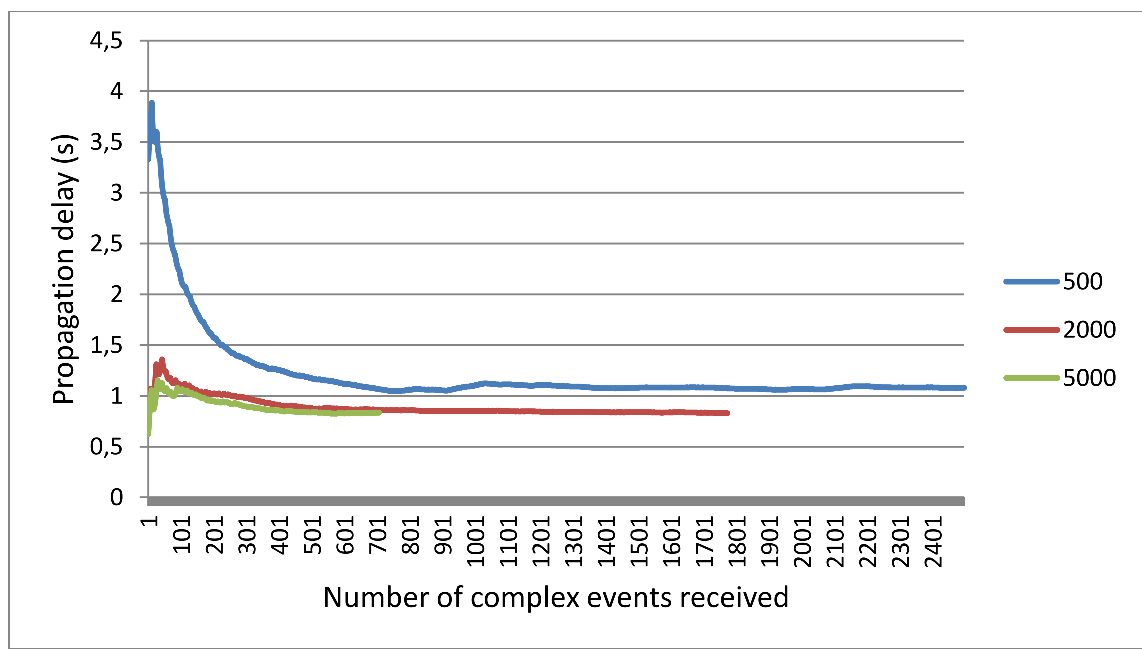

Figure 5.9. Average propagation delays for scenario 7, 5min test.

Figure 5.9 highlights more clearly the difference of averages between cycle time 500 and cycle times 2000 and 5000. Taking the initial peak into consideration the cap would be slightly smaller.

5.1.10 Scenario 8: Fastory UDF query, 30min test.

Scenario eight was performed to demonstrate the monitoring system performance on an actual manufacturing system using the Fastory line. Scenario eight query uses recursive UDFs where the UDF can be called multiple times. Table 5.10 contains the CEP meas-

urements and figure 5.10 depicts the average propagation delays for each event passing through the monitoring system.

Table 5.10 CEP measurements for the 30min test of scenario 8.

Measured point	Results
	Fastory
Events sent from Fastory manufacturing line	1688 units
Query passable events sent*	1688 units
Total number of complex events	1688 units
Average complex event rate per minute	57.5 complex events/minute
Minimum propagation delay	1689.77 ms
Maximum propagation delay	33024.7465 ms
Average propagation delay	8008.68 ms
*Number of events sent from the event source that should pass the query defined for this scenario	

In Table 5.10 the query passable events comply with the total number of events which means that no events were lost. As the query generated a new complex event for every event both the events sent and query passable events are the same. In this scenario the maximum propagation delay reaches 33 seconds even with average complex event rate under 60 events per minute. The high propagation delays are due to the timing problems described earlier in chapter 4.8.2. A combination of time drifting that increases in the long run of measurement and the delay attributed to the device.

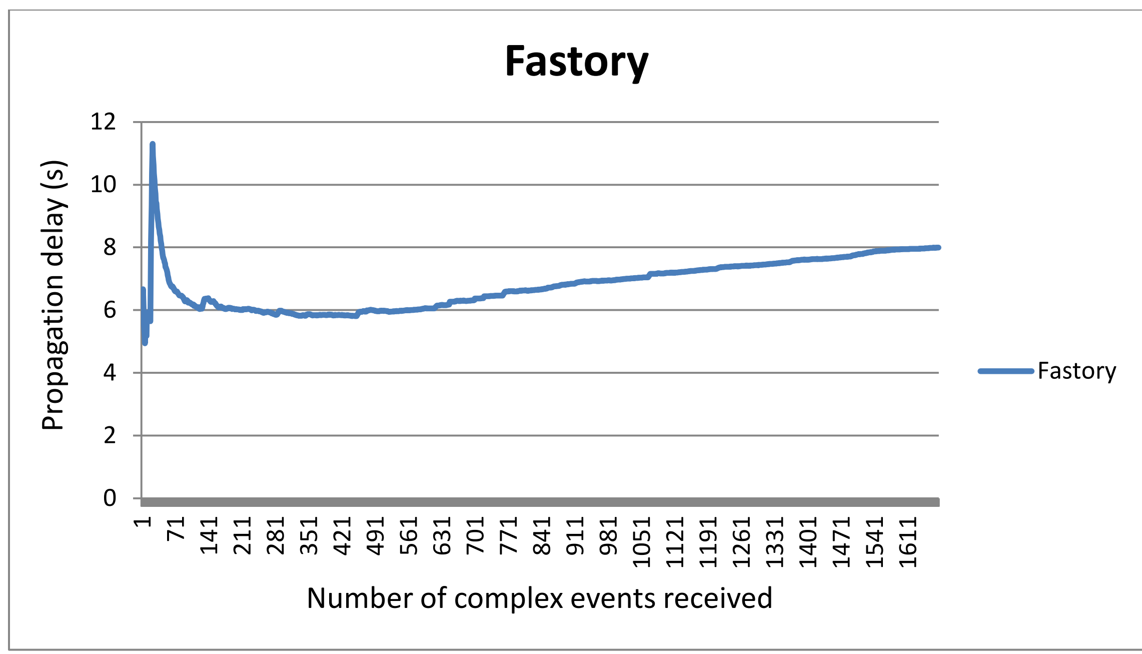

Figure 5.10. Average propagation delay for factory scenario, 30min test.

Figure 5.10 once again demonstrates a steep jump during the initial events with the average propagation delay quickly returning back to average of 6 seconds. Such jump cannot be attributed to initialization of CEP as in the earlier scenarios. The source for this initial jump is not known and would require further research. After around 600 complex events received the propagation delay starts to gradually rise reaching 8 seconds at the end of the 30 minute measurement.

5.2 Test results for cloud performance measurements

The cloud performance tests were performed according to the test parameters introduced in chapter 3.2 table 3.3. The cloud performance was measured by measuring processor time and committed bytes of the cloud computer. Tests were performed on scenarios 1,2,3,6 and 8 as they were the most interesting in the cloud performance point of view.

5.2.1 Scenario 1: Normal LINQ query, 5min test.

Scenario one tests the CEP performance with a normal LINQ for 5 minutes. Figure 5.11 shows the processor time in percents plotted over five minutes for each cycle time. Figure 5.11 shows the committed bytes over the same five minutes.

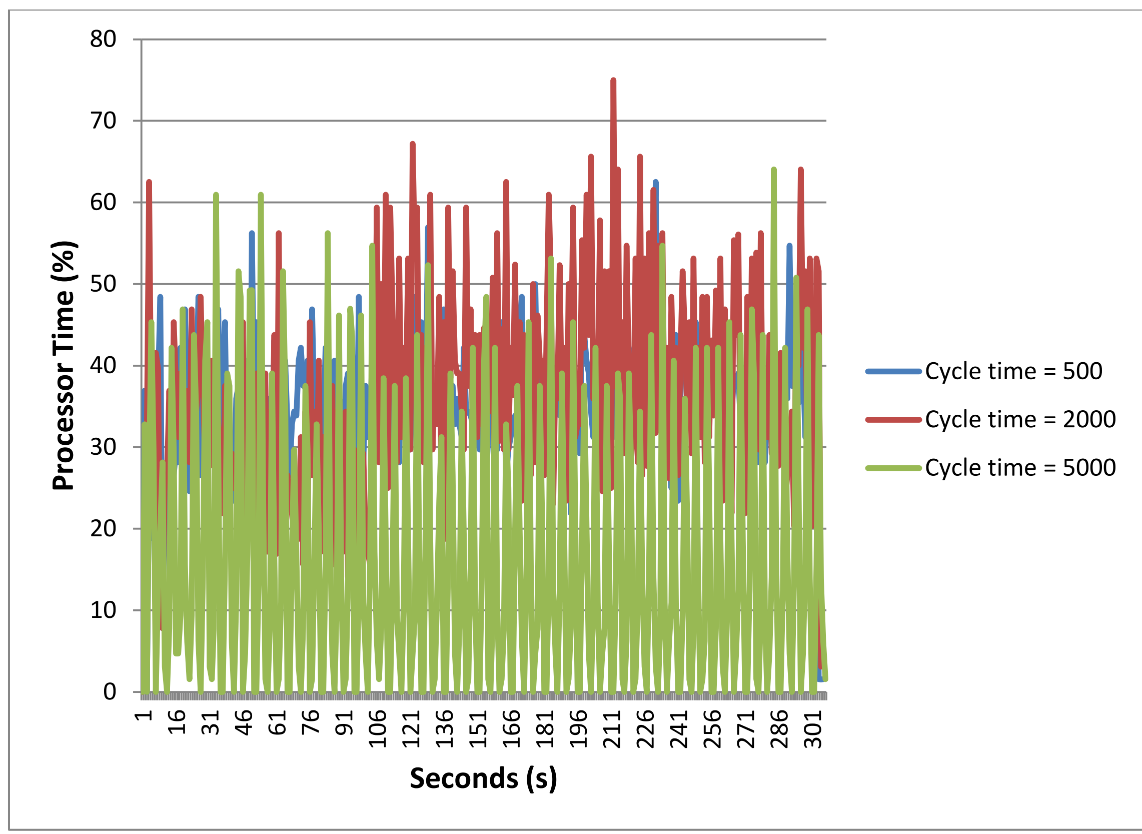

Figure 5.11. Processor time for scenario 1, 5min test.

Figure 5.10 depicts the processor time for each cycle time. Processor time for cycle times 500 and 2000 keep constantly around 30 to 50 percent while cycle time 5000 processor time keeps around the same values but drops constantly to zero.

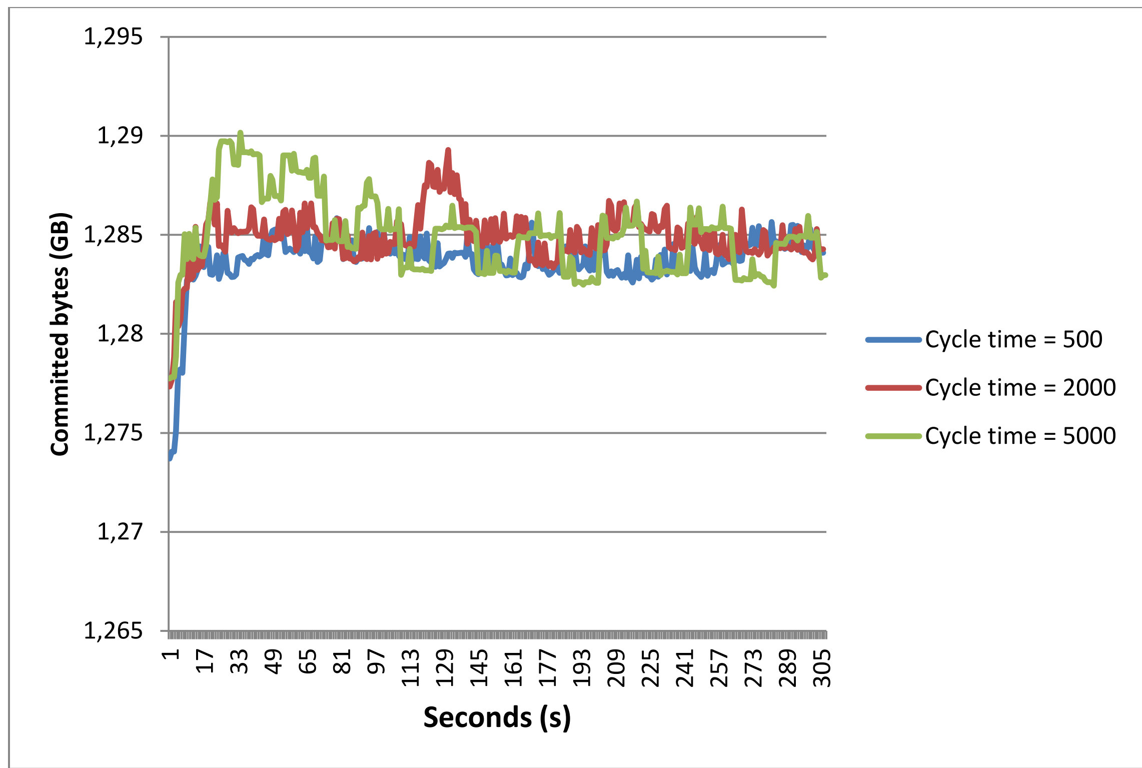

Figure 5.12. Committed bytes for scenario 1, 5min test.

Figure 5.11 describes the amount of committed bytes depicted in gigabytes while running the CEP on the cloud. For each cycle time the amount of committed bytes rises five to ten megabytes.

5.2.2 Scenario 1: Normal LINQ query, 24h test.

A 24 hour test was done for scenario one and the performance measurements done for the cloud can be seen on figures 5.11 and 5.12. Figure 5.11 contains the processor time in percents over the whole 24 hours.

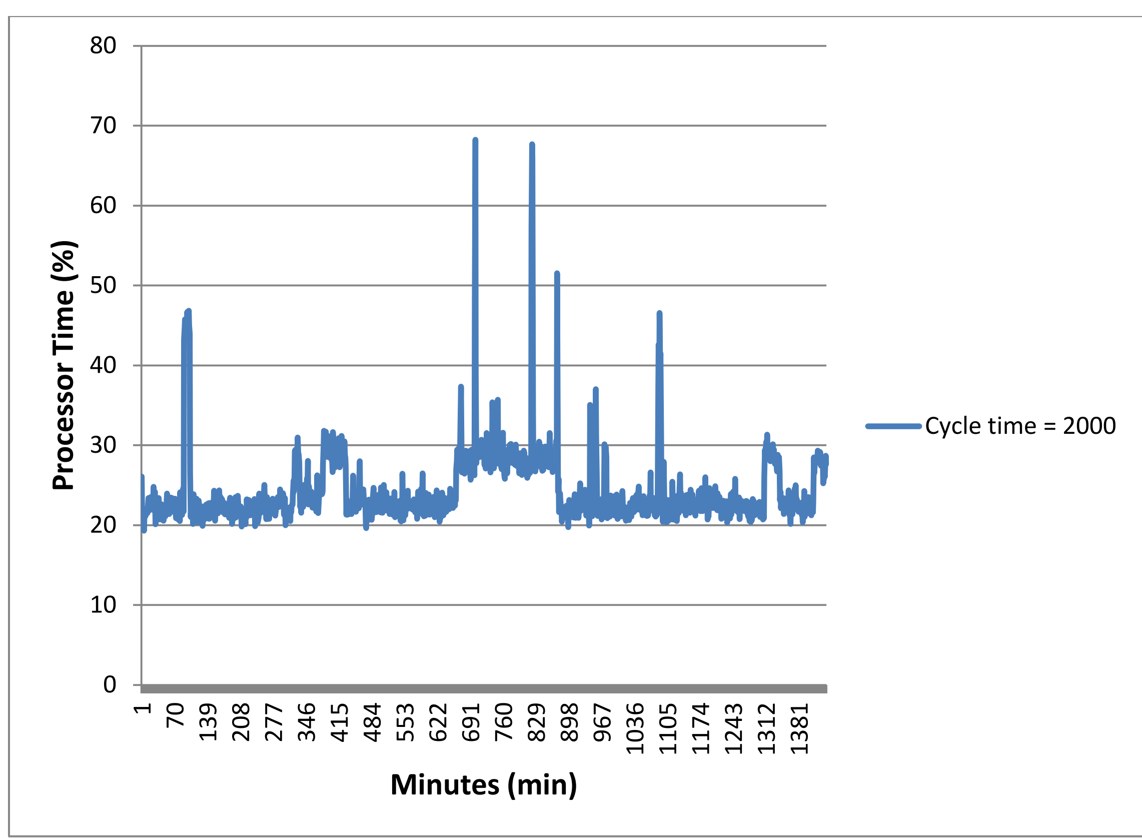

Figure 5.13. Processor time for scenario 1, 24h test.

As seen on figure 5.12 the processor time of the cloud stays around 20 to 30 percent the whole time except for jumps on 100, 700, 820, 870 and 1100 minutes.

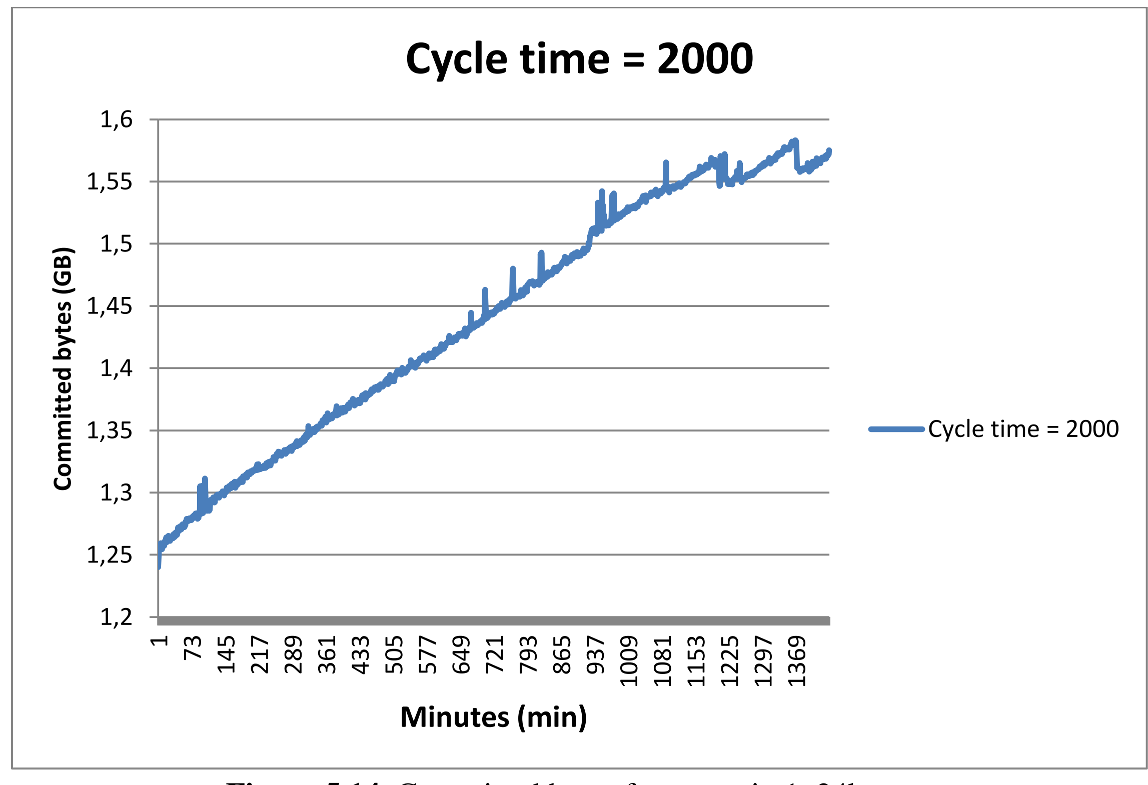

Figure 5.14. Committed bytes for scenario 1, 24h test.

Figure 5.13 illustrates the committed bytes in the cloud computer running the CEP. As seen from the figure the amount of committed bytes increases steadily as the test progresses starting from 1.24 GB up to 1.57 GB of committed bytes.

5.2.3 Scenario 2: Normal LINQ query, 5min test.

Scenario two was a similar to scenario one but with higher event through put. Figure 5.14 shows the processor time of the cloud processor while running the CEP. Figure 5.15 shows the committed bytes of the cloud memory.

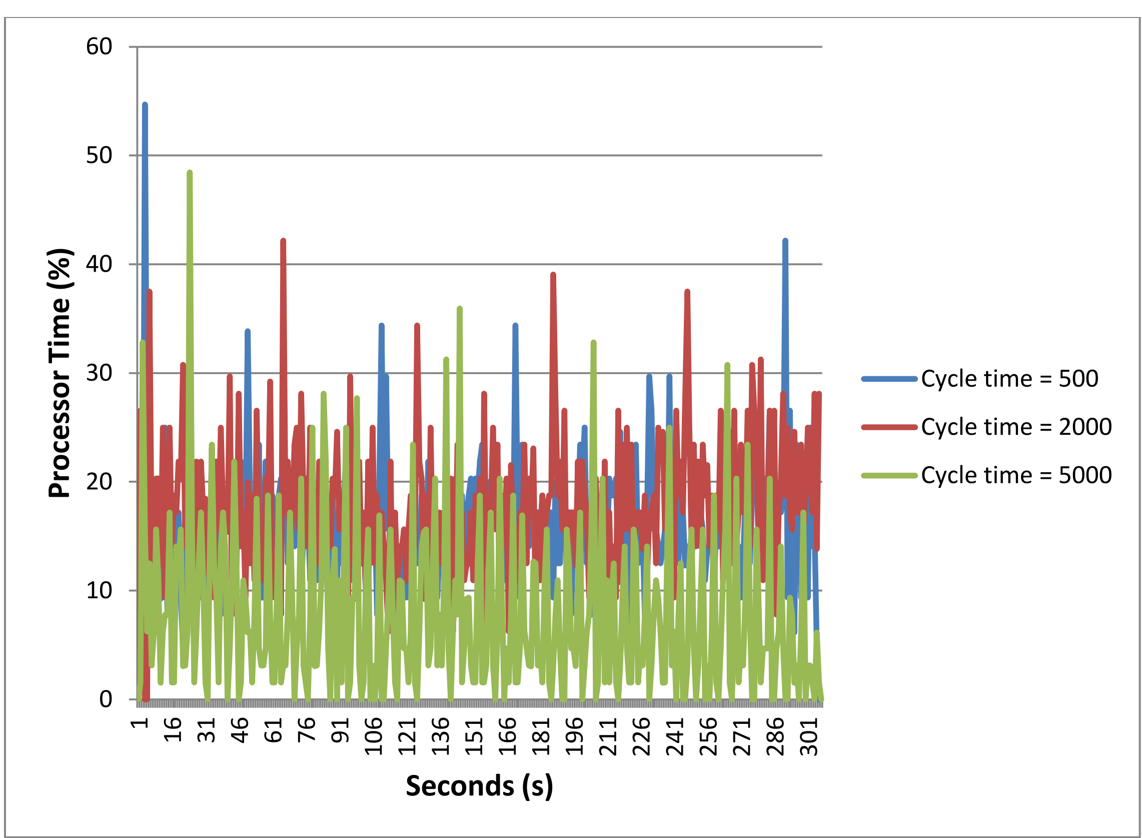

Figure 5.15. Processor time for scenario 2, 5min test.

Figure 5.14 shows that for cycle times 500 and 2000 the values stay consistently around 10 to 25 percent. As for cycle time 500 values keep in between 0 to 20 percent with some peaks exceeding 30 percent.

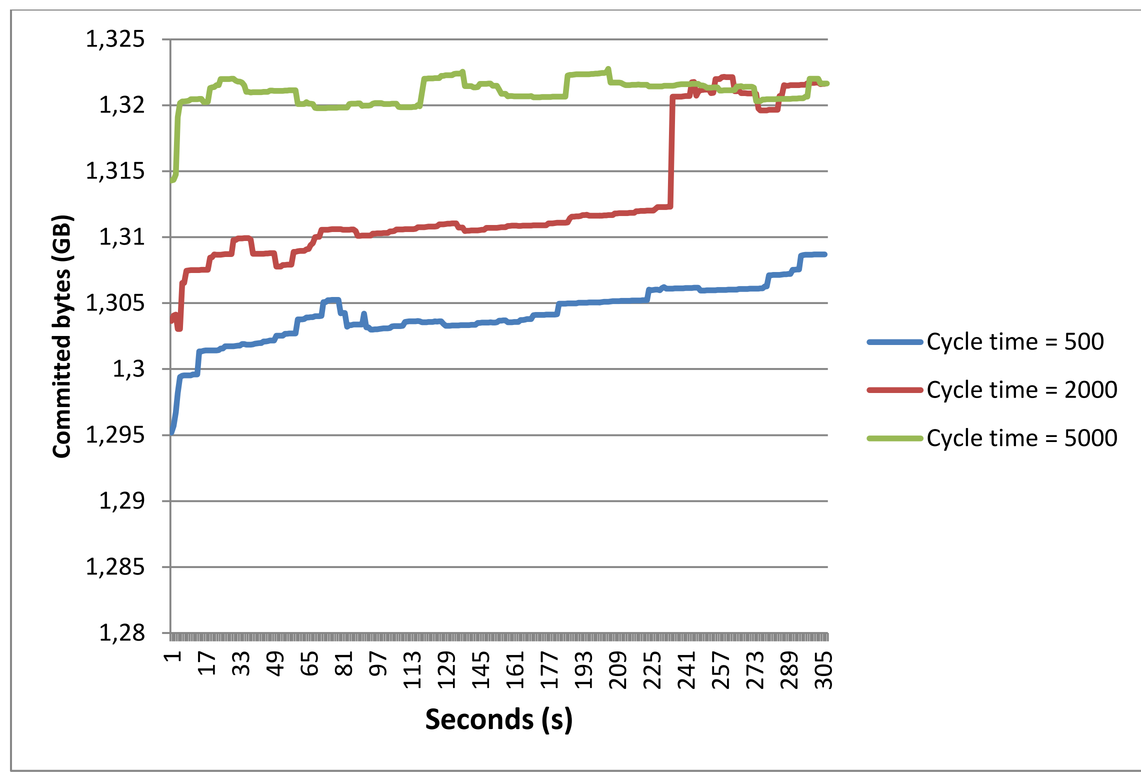

Figure 5.16. Committed bytes for scenario 2, 5min test.

The committed bytes vary between different cycle times with each rising from the previous as test progresses. Cycle times were measured in the order 500, 2000 and 5000. The committed bytes value for cycle time 2000 start close to where the 500 ended. The same can be seen with cycle times 2000 and 5000 where cycle time 5000 is five megabytes lower than when cycle time 2000 ended.

5.2.4 Scenario 3: User defined function query, 5min test.

Scenario three was a query utilizing UDF. The figure 5.17 shows the processor time of the cloud while the CEP is running. Figure 5.18 shows the amount of committed bytes of the cloud memory during the test.

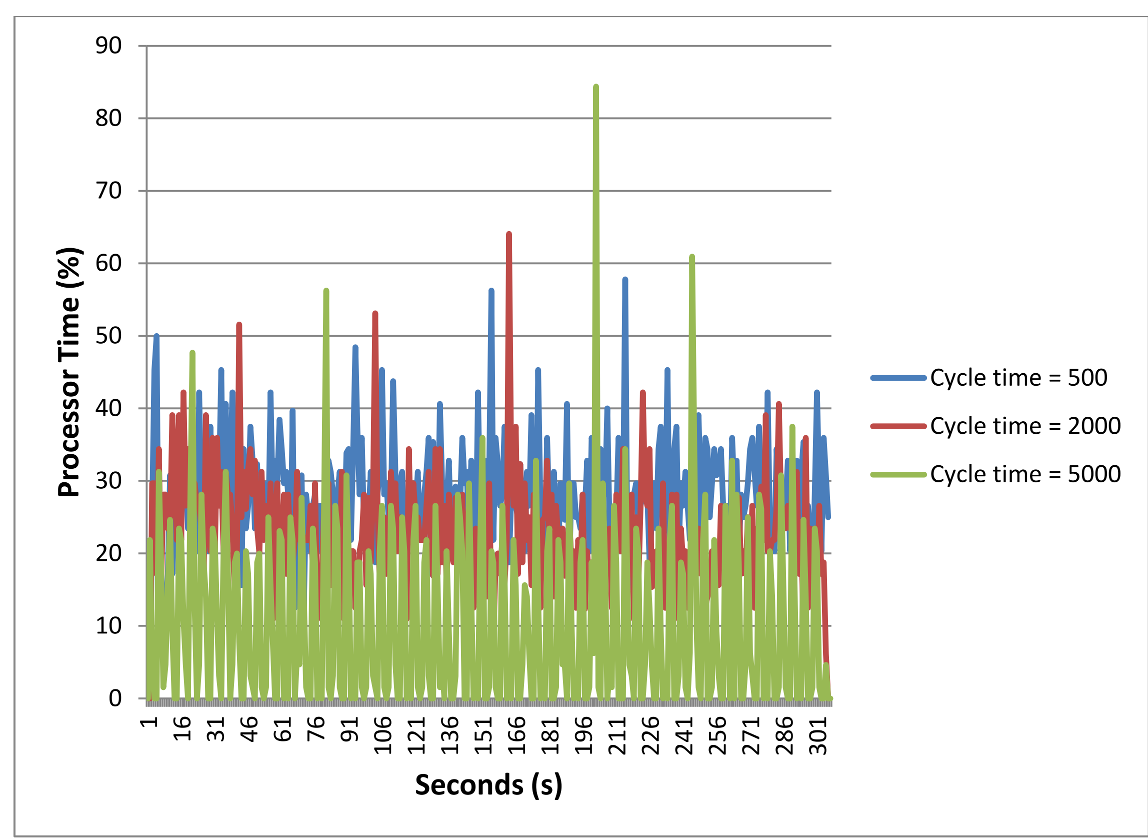

Figure 5.17. Processor time for scenario 3, 5min test.

In figure 5.17 values keep consistently under 40 percent processor time in all cases. Between 10 and 40 percent for cycle times 500 and 2000 while cycle time keeps between 0 and 30 percent. In each case there are occasional jumps exceeding these values. Like for cycle time 5000 jumping at 202 seconds to 84 percent processor time.

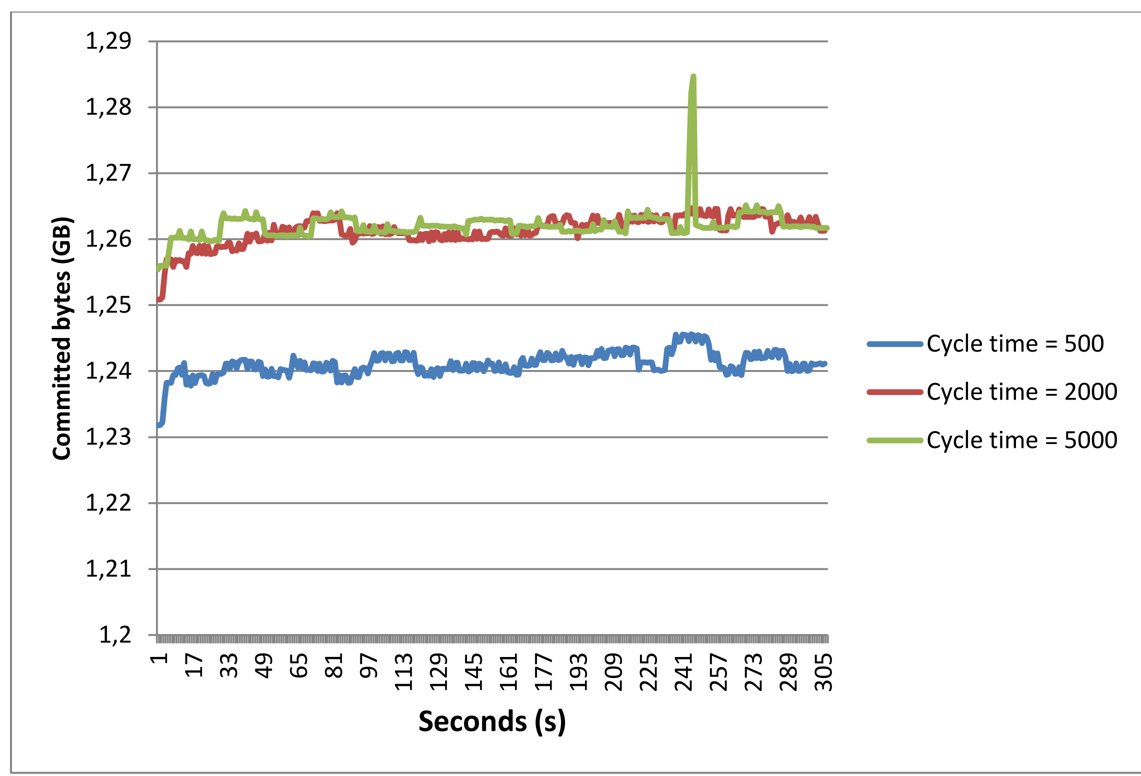

Figure 5.18. Committed bytes for scenario 3, 5min test.

The committed bytes can be seen in figure 5.18 with almost no variation between cycle times 2000 and 5000 with 5000 having a jump between seconds 241 and 257. Cycle time 500 measurements are noticeably under with 0.02 GB less committed bytes compared to cycle times 2000 and 5000.

5.2.5 Scenario 6: Additional normal LINQ query, 5min test.

Scenario 6 was an additional LINQ query to compare scenarios one and three. The figure 5.19 shows the processor time of the cloud while the CEP is running. Figure 5.20 shows the amount of committed bytes of the cloud memory during the test.

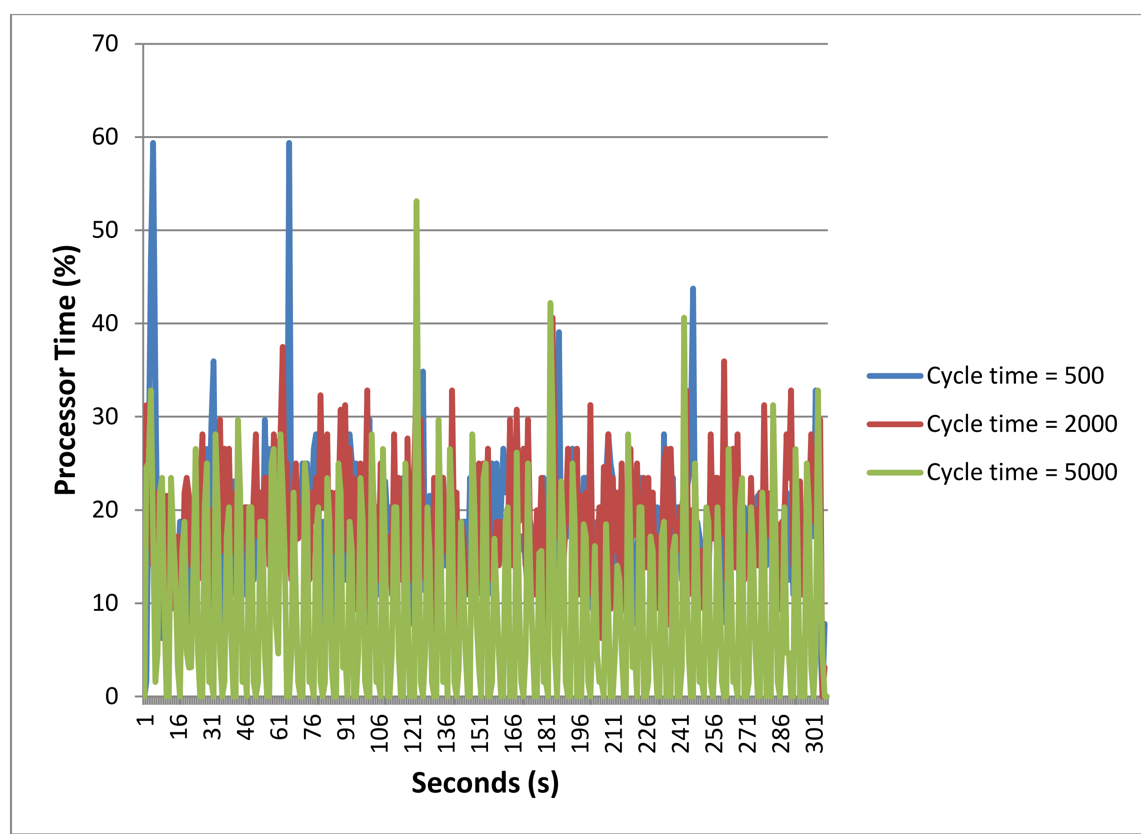

Figure 5.19. Processor time for scenario 6, 5min test.

In figure 5.19 processor time holds under 30 percent in each case with occasional jumps reaching from 35 to 60 percent. Cycle time 5000 often drops down to zero in this scenario also. The cycle times 500 and 2000 resemble each other with values between 10 and 30 percent.

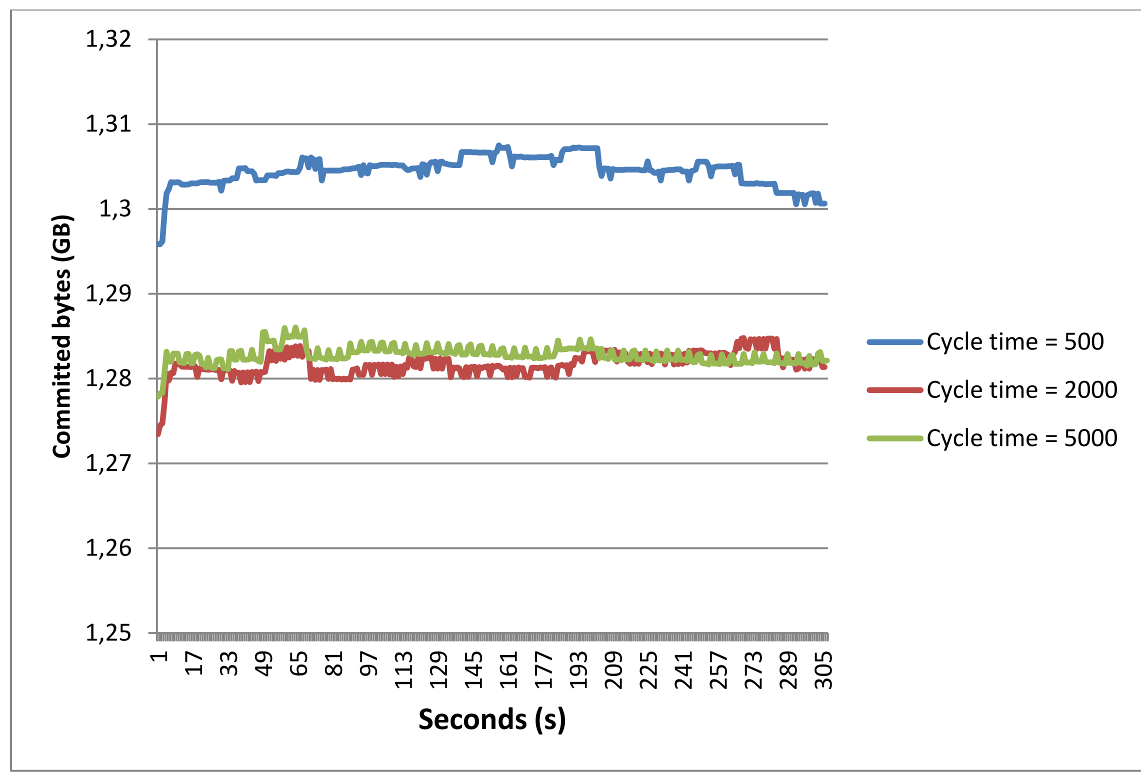

Figure 5.20. Committed bytes for scenario 6, 5min test.

As seen in figure 5.20 committed bytes for cycle times 2000 and 5000 resemble each other closely but cycle time 500 committed bytes are 0.02 GB higher, which is the reverse for the values at scenario three.

5.2.6 Scenario 8: Fastory UDF query. 30min test.

Scenario eight was performed to test the monitoring system in an actual manufacturing system and measure its performance. The figure 5.21 shows the processor time of the cloud while the CEP is running. Figure 5.22 shows the amount of committed bytes of the cloud memory during the test.

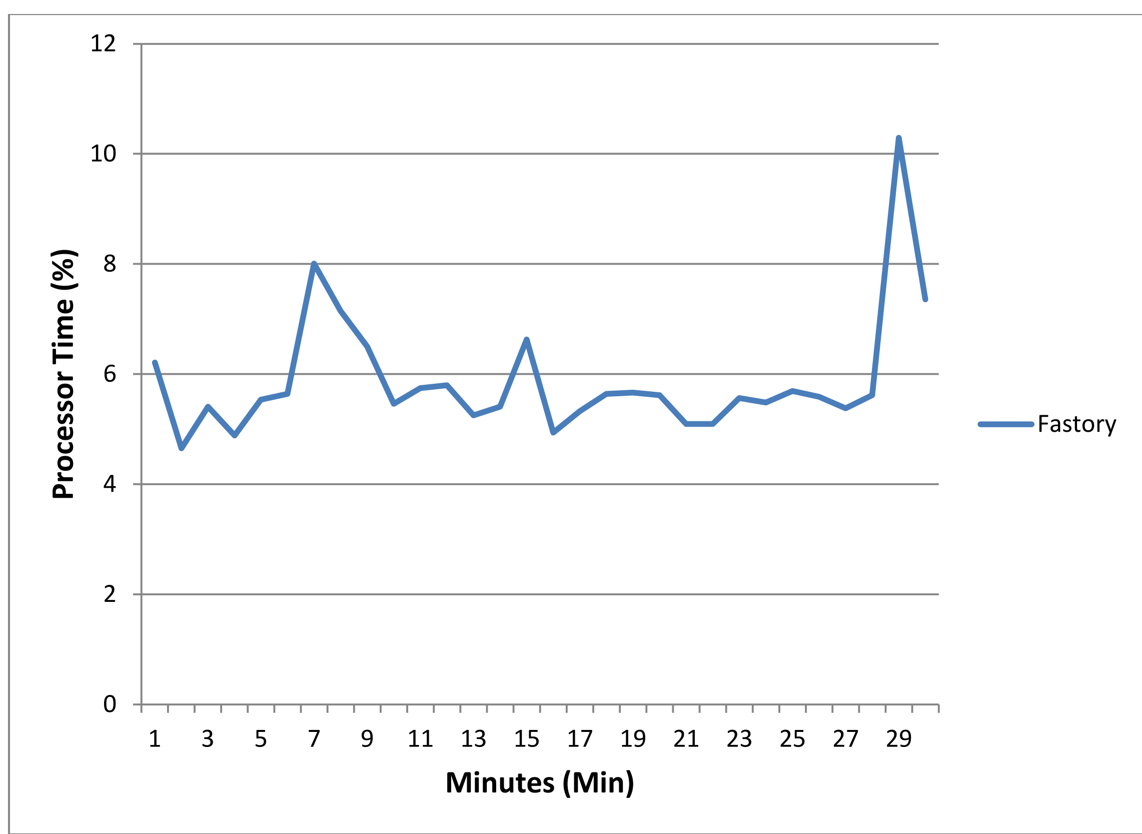

Figure 5.21 Processor time for scenario 8, 30min test.

Figure 5.21 shows the processor time during the factory measurement for each minute. The processor time holds near 6 percent with a few jumps to 8 and 10 percents. The measurement interval is different from other scenarios which results in a different looking figure. The amount of events arriving to the cloud was lower than for the other scenarios resulting in lower processor utilization.

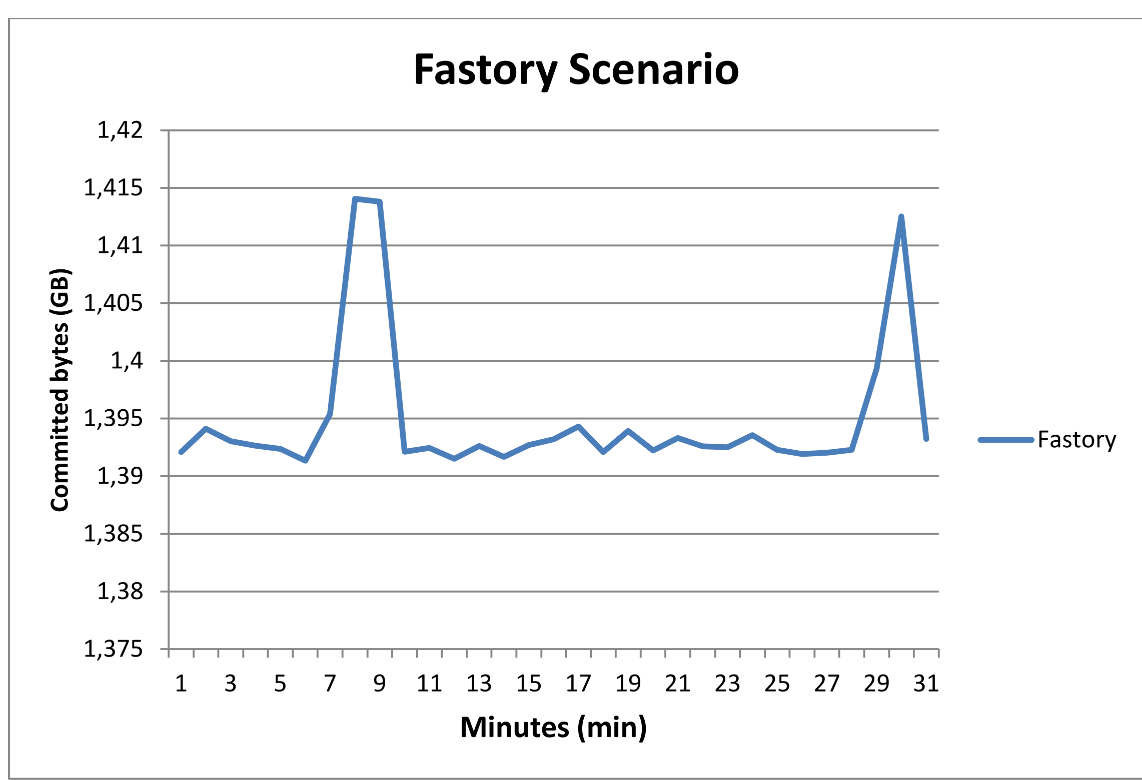

Figure 5.22 Committed bytes for scenario 6, 30min test.

Committed bytes for scenario 8 seen in figure 5.22 hold steady between 1.39 and 1.395 GB. Between minutes 7 and 9 the committed bytes jump to 1.41 GB which occurs again at 30minutes into the measurement.

5.3 Discussion of results

This chapter will discuss the results by comparing specific scenarios between each other to indentify system limitations and find possibilities for improvement. The discussion will concentrate on measurements described in sections 5.1 and 5.2. Discussion on the CEP and monitoring system will be in the next chapter. This section is divided into subsections by result findings.

5.3.1 Length of measurements

Three measurements was performed on scenario one for different lengths of time to detect if longer measurement times affect the results. It can be seen from figures 5.1, 5.2 and 5.3 that average propagation delay stays the same. The number of query passable events sent reduces on long measurements. This is a result of FluicCirc Simulator generating a large amount of temperatureChange events at the beginning as the measurement stations are initialized. After initialization the number of temperatureChange events reduced to a very small amount compared to the main flowChange events being generated. The random generation of events was described in chapter 4.6.2.

Cloud performance measurements for scenario one shows that CPU utilization stays around the same on both measurements. The committed bytes measurement show high memory usage on long measurements that cannot be seen on the 5 minute measurements. Memory usage is managed before reaching maximum level, but stays above 90% threshold. With the results obtained from scenario ones measurements it is assumed that the length of measurement does not affect measurement results.

5.3.2 Maximum event processing rate

Maximum event processing rate is reached in scenarios two and four as seen from figures 5.4 and 5.6. The average propagation rate keeps increasing as events are queued up to the buffer as described in 5.1.4. In both cases the average complex event rate per minute reaches around 555 events per minute. It can be assumed that 555 event per minute is the maximum rate that the CEP can process and generate events with this implementation and hardware.

5.3.3 Query initialization issue

An initial jump of the propagation delay can be seen from most of the average propagation delay figures. The jump occurs when the CEP instance is initialized for the first time. It disappears when the CEP is not restarted but only the query is changed as seen in scenario 3. It is not know why the initialization of the CEP results in a jump in the propagation delay and requires further investigation.

5.3.4 Small amount compared to large amount of incoming events

Comparing normal LINQ queries between scenarios one and two as in between small and large amount of incoming events is not possible as can be seen from figures 5.1 and 5.4. Scenario two's average complex event rate per minute is at its maximum and incoming events get queued up to the buffer increasing the propagation time. Scenario one will be compared to scenario six that has a higher amount of query passable events but the maximum event rate is not exceeded.

Both scenarios send around the same amount of events. Query passable events are 18 times higher on scenario 6. The difference can be clearly seen on figures 5.1 and 5.8 that average propagation time is higher on scenario six that processes larger amount of query passable data. The average time is roughly 3 times higher on scenario six. This cannot be seen from cloud performance measurements as CPU utilization is lower on scenario six while the memory usages are around the same. The number of processed events seems to affect the propagation time.

5.3.5 Normal LINQ query compared UDF query

Comparing normal LINQ to UDF is done between scenario three and scenario six as they have similar queries but different implementation. Average propagation delay is almost double for scenario six compared to scenario three. This can be attributed to the design of the UDF. UDF used in scenario three consumes two events in order to generate one complex event. Thus scenario three generates half the complex event compared to scenario six and half the complex event rate per minute. Both scenarios process roughly the same amount of events but scenario three generate less complex events and thus has a lower complex event rate. It can be assumed that generating complex events increases propagation time rather than just processing the query passable data. Future measurements would be needed between UDF and LINQ scenarios with same amount of generated complex events to gain information on which query is more efficient. Scenarios three and six implicates that query method does not matter but the amount of generated complex events.

5.3.6 Multiple queries compared to single queries

Scenarios containing one query were compared to scenarios with two queries. These scenarios were scenario six compared with scenario seven and scenario three compared with scenario five. Scenarios six and seven are normal LINQ queries and scenarios three and five are UDF queries. Measurements show that there is very little variation between the measurements and are almost the same. It can be seen from the results that additional query has very little affect on the CEP performance.

5.3.7 Recursive UDF

Recursive functions were used on scenario eight to demonstrate their functionality. This scenario is hard to compare with other scenarios as it was the only one measured with the Fastory use case. Also propagation times were affected by timing problems described in chapter 4.8.2. Recursive UDF generates one complex event for each event. Scenario three UDF could be replaced with recursive UDF not to lose any data. This would require an extra filter to the query as a complex event is generated for every event sent to the recursive UDF, which would quickly result in high complex event rate per minute. Recursive UDF can only be used with small amount of arriving events.

5.3.8 CPU and memory affecting CEP performance

Neither CPU utilization nor memory usage limit the CEP or become a bottleneck for event processing in scenario two, but still the maximum event processing rate is reached. Figure 5.15 shows that CPU utilization stays low and is event lower than for scenario one. A possible reason for reaching the maximum event processing rate is the moderate I/O performance of the cloud. It is possible that the I/O hardware cannot keep

up with handling large amounts of event in processing memory at the same time. Future measurements with higher I/O performance would be required for proof.

5.3.9 Lessons learnt

The following table 5.11 lists the lessons learnt from the results. These lessons were described in the subsections of this chapter.

Table 5.11. Improvements and lessons learnt.

Improvement and lessons learnt
-The length of measurement time does not affect measurement results.
-Maximum event processing rate was around 555 events per minute with the hardware that was tested.
-Memory usage reaches above 90% utilization with long measurements.
-The number of incoming events affects propagations time.
-The higher the amount of complex events generated the higher the propagation delay rises.
-Increasing the number of queries with one has little effect on CEP performance.
-A possible reason for reaching maximum event processing rate is the moderate I/O performance of the cloud.
-Recursive UDF's require low number of incoming events.

The most important findings were that the CEP can process and generate events the maximum amount of 555 events on average per minute and that this bottleneck is possibly due to the moderate I/O performance of the cloud computer. Recursive UDF's could be used in multiple different use cases but require that the number of events routed to it stays under the maximum event processing rate per minute.

6 CONCLUSION AND FUTURE

The main goal of this thesis was to develop a dynamic complex event processor. This chapter will evaluate the developed CEP, monitoring system and assess the result gained from the test scenario measurements. In addition some recommendations for future work will be given.

6.1 Implementation conclusions

Complex event processing has been used in the IT and business sector for a long time and only recently has it been introduced to the factory automation environment. The aim of this thesis was to develop a SOA-based monitoring system with a dynamic complex event processor capable of dynamically adding queries and the possibility to reuse generic user defined functions. This system was also used to gain performance measurements on the CEP and the cloud.

For developing a SOA-based monitoring system the CEP is a vital part of the system. CEP is required to handle the large amounts of events a modern manufacturing solution produces and to handle the logic the monitoring system needs in order to infer information from the stream of events. As the complex event processors are in most cases system specific they require a lot of development time. This problem of solution specific CEP was tackled by developing a CEP with StreamInsight that was generic and configurable for similar solutions that also enabled the use of CEP as a service. This goal was achieved by using un-typed adapters that allow interoperability between different systems. This way the CEP can adapt with the different SOA environments and requires less development time. StreamInsight LINQ query definition language allowed the possibility to develop user defined functions that can be used in the query template. This allowed development of recursive functions that are generic in nature and opened up the possibility to reuse queries and save development time. The dynamic SOA-ready CEP developed allows the CEP to be fully integrates to the SOA architecture as an autonomous and interoperable part of the system and enables the monitoring system to manage and define monitoring logic during runtime.

Both of the use case implementations developed for this thesis were fully functioning monitoring systems proving the concept of a dynamic SOA-based monitoring system. The important addition that the developed system brings to the current complex event processors used in the industrial monitoring environment is higher interoperability between systems and the possibility to define and reuse queries during runtime.

6.2 Result conclusions

Discussion of results in chapter 5.3 went through the measurement results described in chapters 5.1 and 5.2. The last section 5.3.8 described the main lessons learnt from these results. The combination of CEP performance measurement and cloud performance measurements gave a good overall view on how the software side and hardware side copes during system runtime. The different test scenarios resembling actual monitoring needs provided measurement data on how the system performs in different situations. This allowed to identify limitations of the system and gave clues on which variables affect the CEP performance. The measurement data and the lessons learnt can provide valuable information when developing similar CEP systems.

6.3 Future work

Many improvement areas were identified on the developed monitoring system. This section will describe recommendations for future work.

The main development for this thesis was the CEP that has many improvement areas. Many limitations were set on the CEP that could be fixed to improve its usability. The problem with bindings could be solved by adding a similar CodeDom functionality as with the event type and query template to define adapter bindings. More un-typed adapters could be added to allow also edge and interval events. DPWS support as in WS-Messaging could be added to the adapters in order for the DPWS hub to be made unnecessary. The amount of event type variables could be increased from five. Also more generic UDFs should be developed that target functions required in industrial monitoring solutions. In addition an important improvement would be to be able to add object to the CEP server without dynamically generating classes on CodeDom, as it opens up security issues. With CodeDom the user injects code straight into the class without any security checks.

Tests on the developed monitoring system should be ran on different hardware to compare how for example different event flow input and output processing capabilities would affect the number of complex event generated per minute. This would help to verify that the input and output processing capability is the source for the bottleneck identified in the results. This information would be important for the future developments of such systems. Also testing the system with more cycle times could bring new information. Future research into what affects the initial propagation time jumps on measurement scenarios would be required to prevent such behaviour from occurring.

7 REFERENCES

Amazon. 2013. Amazon Elastic Compute Cloud (Amazon EC2). [WWW]. [Accessed on 17.12.2013]. Available at: http://aws.amazon.com/ec2/.

Barr, J. 2010. Host Your Web Site in the Cloud: Amazon Web Services Made Easy. 392 p.

Bean, J. 2010. SOA and Web Services Interface Design: Principles, Techniques, and Standards. Morgan Kaufmann Publishers. 383 p.

Brandl, D. 2008. What is ISA-95? Industrial Best Practices of Manufacturing Information Technologies with ISA-95 Models. [WWW]. BR&L Consulting. [Accessed on 17.12.2013]. Available at: http://www.apsom.org/docs/T061_isa95-04.pdf.

Buyya, R., Yeo, C,. Venugopal, S., Broberg, J, & Brandic Ivona. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems. pp. 599-616.

Buyya, R., Broberg, J. & Goscinski, A. 2011. Cloud Computing: Principles and Paradigms. John Wiley & Sons. 664 p.

Cachapa, D., Colombo, A., Feike, M., Bepperling, A. 2007. An Approach for Integrating Real and Virtual Production Automation Devices Applying the Service-oriented Architecture Paradigm. Emerging Technologies and Factory Automation. IEEE. pp. 309-314.

Cachapa, D., Harrison, R. & Colombo, A. 2010a. Monitoring Functions as Service Composition in a SoA-based Industrial Environment. IECON 2010 - 36th Annual Conference on IEEE Industrial Electronics Society, pp. 1353-1358.

Cachapa, D., Harrison, R., Colombo, A. & Lee, L. 2010b. SoA-based production monitoring systems for energy efficiency: A case-study using Ford's POSMon system. IEEE International Conference on Industrial Technology. pp. 1077–1081.

Cachapa, D., Harrison, R., Colombo, A. 2010c. Improving Energy Efficiency in the Production Floor Using SoA-Based Monitoring Techniques. Emerging Trends in Technological Innovation. Springer Berlin Heidelberg. pp. 159-166.

Colombo, A., Karnouskos, S., Mendes, J. 2010. Factory of the Future: A Service-oriented System of Modular, Dynamic Reconfigurable and Collaborative Systems. Artificial Intelligence Techniques for Networked Manufacturing Enterprises Management. Springer London. pp. 459-481.

Daneels, A., Salter, W. 1999. What is SCADA?. International Conference on Accelerator and Large Experimental Physics Control Systems. pp. 339-343.

Delsing, J., Gustafson, J., Deventer, J. 2010. A service oriented architecture to enable a holistic system approach to large system maintenance information. Lulea University of Technology. pp. 8.

Du, R., Elbestavi, M. & Wu, S. 1995. Automated monitoring of manufacturing processes, part 1: monitoring methods. Journal of Engineering for Industry, 117/121.

European Commission, Information Society and Media . 2010. Monitoring and Control Today’s market and its evolution till 2020. Luxembourg Office for Official Publications of the European Communities. Pp. 226

Fan, Y., Huang, C., Wang, Y. & Zhang, L. 2005. Architecture and operational mechanisms of networked manufacturing integrated platform. International Journal of Production Research, 43(12), pp. 2615–2629.

Garcia, J., Lobov, A. & Lastra, J. 2011. OPC-UA and DPWS Interoperability for Factory Floor Monitoring using Complex Event Processing. Industrial Informatics (INDIN) 9th IEEE International Conference. pp. 205-211.

Galloway, B., Hancke, G. 2012. Introduction to Industrial Control Networks. Communications Surveys & Tutorials, IEEE. pp. 860-880.

Hall, R., Cervantes, H. 2004. Challenges in Building Service-Oriented Applications for OSGi. IEEE Communications Magazine. pp. 144-149.

IEEE Std C37.1™ -2007. IEEE Standard for SCADA and Automation Systems. New York 2007. pp. 146.

Inico. 2010. S1000 User Manual. [WWW]. [Accessed on 17.12.2013]. Available at: http://www.inicotech.com/doc/S1000%20User%20Manual.pdf.

ISA. 2010. ISA-95 web page. [WWW]. [Accessed on 17.12.2013]. Available at: http://www.isa-95.com

Jammes, F. & Smit, H. 2005. Service-Oriented Architectures for Devices - the SIRENA View. IEEE International Conference on Industrial Informatics (INDIN). pp. 140-147.

Jammes, F., Mensch, A., Smit, H. 2007. Service-Oriented Device Communications Using the devices Profile for Web Services. Advanced Information Networking and Applications Workshops. pp. 947-955.

Karnouskos, S., Colombo, A., Jammes, F., Delsing, J. & Bangemann, T. 2010. Towards an architecture for service-oriented process monitoring and control. IECON 2010 - 36th Annual Conference on IEEE Industrial Electronics Society. pp. 1385–1391.

Karnouskos, S., Colombo, A. 2011. Architecting the next generation of service-based SCADA/DCS system of systems. IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society. pp. 359-364.

Karnouskos, S., Colombo, A., Bangemann, T., Manninen, K., Camp, R., Tilly, M., Stluka, P., Jammes, F., Delsing, J. & Eliasson., Jens. 2012. A SOA-based architecture for empowering future collaborative cloud-based industrial automation. 38th Annual Conference of the IEEE Industrial Electronics Society IECON. pp. 5766-5772.

Luckham, D., Frasca, B. 1998. Complex Event Processing in Distributed Systems. Stanford University.

Luckham, D. 2002. The power of events: an introduction to complex event processing in distributed enterprise systems. Boston. Addison Wesley. 400 p.

Luckham, D. 2006. What's the Difference Between ESP and CEP?. [WWW]. [Accessed on 17.12.2013] Available at: http://www.complexevents.com/2006/08/01/what%E2%80%99s-the-difference-between-esp-and-cep/.

McClanahan, R. 2003. SCADA and IP Is Network Convergence Really Here?. IEEE Industry application magazine. Mar/Apr 2003. pp. 8.

McLeod S., Karnouskos S. D5.3 Engineering Toolkit for IMC-AESOP. 2013. IMC-AESOP. Project deliverable. Unpublished report. pp. 37.

Mendes, M., Bizarro, P. & Paulo, P. 2009. A Performance Study of Event Processing Systems. Performance Evaluation and Benchmarking. Springer Berlin Heidelberg. pp. 221-236.

MS-CEPM. 2012. Microsoft Complex Event Processing Engine Manageability Protocol. [WWW]. [Accessed on 16.12.2013]. Available at: http://msdn.microsoft.com/en-us/library/ee320526%28v=sql.105%29.aspx.

MS-WSPIG. 2012. Web Services Protocols Interoperability Guide. [WWW]. [Accessed on 10.12.2013]. Available at: http://msdn.microsoft.com/en-us/library/ms734776.aspx.

MS-SI. 2012. Microsoft StreamInsight 2.0. [WWW]. [Accessed on 16.12.2013]. Available at: http://technet.microsoft.com/en-us/library/hh750620%28v=sql.10%29.aspx

NCS. 2004. Supervisory Control and Data Acquisition (SCADA) Systems. US National communication systems office. Technical information bulletin 04-1. 64 p.

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L. & Zagorodnov, D. 2009. The Eucalyptus Open-Source Cloud-Computing System. Cluster Computing and the Grid. 9th IEEE/ACM International Symposium. pp. 124-131.

OASIS. 2006. Reference Model for Service Oriented Architecture 1.0. [WWW]. [Accessed on 16.12.2013]. Available at: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html.

OASIS. 2009a. Devices Profile for Web Services Version 1.1. [WWW]. [Accessed on 16.12.2013]. Available at: http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.pdf

OASIS. 2009b. Web Services Dynamic Discovery (WS-Discovery) Version 1.1. [WWW]. [Accessed on 16.12.2013]. Available at: http://docs.oasis-open.org/ws-dd/discovery/1.1/wsdd-discovery-1.1-spec.html

Priyantha, N., Kansal, A., Goraczko, M., Zhao, F. 2008. Tiny Web Services: Design and Implementation of Interoperable and Evolvable Sensor Networks. ACM/IEEE Conference on Embedded Networked Sensor Systems. pp. 253-266.

Ragavan, S. V., Kusnanto, I. K. & Ganapathy, V. 2012. Service Oriented Framework for Industrial Automation Systems. Procedia Engineering, 41(Iris). pp.716-723.

Roshen, W. 2009. SOA-Based Enterprise Integration: A Step-by-Step Guide to Services-Based Application Integration. McGraw-Hill/Osborne. 384 p.

Shodjai, P. 2006. Web Services and the Microsoft Platform. [WWW]. Microsoft Corporation. [Accessed on 17.12.2013]. Available at: http://msdn.microsoft.com/en-us/library/aa480728.aspx.

Sotomayor, B., Montero, R., Llorente, I. & Foster, I. 2009. Virtual Infrastructure Management in Private and Hybrid Clouds. IEEE Internet computing. pp. 14-22.

Stopper, M. & Gastermann, B. 2010. Applying SOA Concepts to Distributed Industrial Applications Using WCF Technology. IAENG Transactions on Engineering Technologies volume 5.

Thirumala, R., Rani, U., Gonsalves, A. 2006. An Event Notification Service based on XML Messaging on Different Transport Technologies. 9th International Conference on Information Technology (ICIT'06).

Vaquero, L., Rodero-merino, L., Caceres, J. & Lindner, M. A break in the clouds: Towards a cloud definition. ACM SIGCOMM Computer Communication Review. pp. 50-55.

Walzer, K., Rode, J., Wunsch, D. & Groch, M. 2008. Event-driven manufacturing: Unified management of primitive and complex events for manufacturing monitoring and control. IEEE International Workshop on Factory Communication Systems, pp.383–391.

W3C. 2001. Web Services Description Language (WSDL) 1.1. [WWW]. [Accessed on 17.12.2013]. Available at: http://www.w3.org/TR/wsdl.

W3C. 2003. Extensible Markup Language (XML). [WWW]. [Accessed on 17.12.2013]. Available at: http://www.w3.org/XML/.

W3C. 2004a. Web Services Architecture. [WWW]. [Accessed on 17.12.2013]. Available at: http://www.w3.org/TR/ws-arch/.

W3C. 2004b. Web Services Glossary. [WWW]. [Accessed on 17.12.2013]. Available at: http://www.w3.org/TR/ws-gloss/.

W3C. 2004c. Web Services Addressing (WS-Addressing). [WWW]. [Accessed on 17.12.2013]. Available at: http://www.w3.org/Submission/ws-addressing/.

W3C. 2006. Web Services Transfer (WS-Transfer). [WWW]. [Accessed on 17.12.2013]. Available at: http://www.w3.org/Submission/WS-Transfer/.

W3C. 2011. Web Services Eventing (WS-Eventing). [WWW]. [Accessed on 17.12.2013]. Available at: http://www.w3.org/TR/2011/CR-ws-eventing-20110428/.

W3C. 2012. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. [WWW]. [Accessed on 17.12.2013]. Available at: http://www.w3.org/TR/xmlschema11-1/.

Zang, C., Fan, Y. & Liu, R. 2008. Architecture, implementation and application of complex event processing in enterprise information systems based on RFID. Information Systems Frontiers Journal. Springer US.

Zeep, E., Bodek, A., Bohn, H., Golatowski, F. 2007. Lessons learned from implementing the
Devices Profile for Web Services. Inaugural IEEE International Conference on Digital Ecosystems and Technologies. pp. 229-232.

Zhang, P. 2010. Advanced Industrial Control Technology. William Andrew. 865 p.

APPENDIX 1: COMPARISON UDF CODE

```
static int comparisonStorage1;
static bool comparisonBool1 = false;
static DateTime comparisonTime;

public static bool comparisonIntWithName(string eventNameFromEvent, int dataFromEvent, string eventNameFromUser,
     int ComparisondataFromEvent, string ComparisoneventNameFromUser, string logic, int scanTimeSeconds)
{

    if(eventNameFromEvent == eventNameFromUser && comparisonBool1 == false)
    {
        comparisonStorage1 = dataFromEvent;
        comparisonBool1 = true;
        comparisonTime = DateTime.Now;
        comparisonTime = comparisonTime.AddSeconds(scanTimeSeconds);
    }

    if(comparisonBool1 == true && comparisonTime < DateTime.Now)
    {
        comparisonBool1 = false;
    }

    if (eventNameFromEvent == ComparisoneventNameFromUser)
    {
        switch (logic)
        {
            case "==":
                {
                    if (comparisonStorage1 == ComparisondataFromEvent)
                    {
                        comparisonBool1 = false;
                        return true;
                    }
                    else
                    {
                        return false;
                    }
                }
          case "<=":
              {
                  if (comparisonStorage1 <= ComparisondataFromEvent)
                  {
                      comparisonBool1 = false;
                      return true;
                  }
                  else
                  {
                      return false;
                  }
              }
          case ">=":
              {
                  if (comparisonStorage1 >= ComparisondataFromEvent)
                  {
                      comparisonBool1 = false;
                      return true;
                  }
                  else
                  {
                      return false;
                  }
              }
```

```
                case "<":
                    {
                        if (comparisonStorage1 < ComparisondataFromEvent)
                        {
                            comparisonBool1 = false;
                            return true;
                        }
                        else
                        {
                            return false;
                        }
                    }
                case ">":
                    {
                        if (comparisonStorage1 > ComparisondataFromEvent)
                        {
                            comparisonBool1 = false;
                            return true;
                        }
                        else
                        {
                            return false;
                        }
                    }
                default:
                    {
                        Console.WriteLine("Comparison returned default!");
                        return false;
                    }
            }
        }
        return false;
    }
```