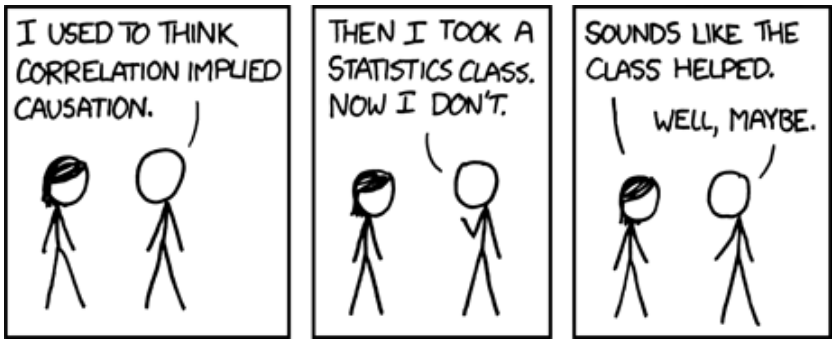
Software maintenance in a Data Distribution Service with Complex Event Processing

Master's Thesis

Final Version



Tom Pesman

Software maintenance in a Data Distribution Service with Complex Event Processing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Tom Pesman
born in Delft, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Logica
George Hintzenweg 89
Rotterdam, the Netherlands
www.logica.com

Cover picture: xkcd, ”Correlation”, http://xkcd.com/552/.

Software maintenance in a Data Distribution Service with Complex Event Processing

Author: Tom Pesman
Student id: 1128884
Email: tom@tnux.net

Abstract

This thesis covers the topic of software maintenance on a system which consists of a Data Distribution Service (DDS) and a Complex Event Processing (CEP) engine. Software maintenance on this system is hard to perform, because of the dependencies between the different components. This thesis answers the main research question: "To which extent do existing software maintenance principles apply to changing a running software system based on a Data Distribution Service with Complex Event Processing?". To answer this research question, an existing change request procedure is used as a basis to create a new change request procedure. A formalising method is added to have a formal way for the developer to analyse the impact of a change. This is needed because of the dependencies within the DDS/CEP system, it is easy to forget to change an important part of the system. The hardest part of the change request is the fact that the system is already running in a production environment, so if a mistake is made, data may be lost. To help with this complex problem a DDS monitoring tool is developed in this thesis, which visualises the structure of the DDS. This tool has more features to ease the maintenance of the system, such as highlighting edges in the graph with similar QoS settings. The case study is performed on a prototype of a system, to show this change request procedure is sufficient, which is verified with the tool.

Thesis Committee:

Chair:	Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor:	Dr. Phil. H. G. Gross, Faculty EEMCS, TU Delft
Committee member:	Dr. K. van der Meer, Faculty EEMCS, TU Delft
Company supervisor:	Ir. E. Essenius, Logica
	Drs. B. Vranken, Logica
	Mr. A.J. de Neef rm, Logica
External supervisor:	Chief Inspector E. de Jonge, Politie Groningen

Preface

First of all, let me Hans-Gerhard Gross (TU Delft) for his support during this thesis and to let me free in finding my own solutions. Second, let me thank Logica to give me the opportunity to let me work freely on my own project and to give me the full support to let this thesis become what it is now. I think that the relaxed atmosphere at Working Tomorrow (graduation department) contributed to the quality of this research. And I want to thank, for their support, the Working Tomorrow Software Architect Edwin and the Project leaders of Working Tomorrow: Bram and Bert. I also want to thank the initiators of this project Arnoud van Zuijlen (Logica), Hans Lammers (Logica), but the most important of all: Elle de Jonge. De Jonge started the idea of i-Catcher . Last but not least, I want to thank my reviewers: Nikki, Susan and Daan.

Tom Pesman
Delft, the Netherlands
June 11, 2010

Contents

List of Figures

Chapter 1

Introduction

A Data Distribution Service (DDS) is a specification of a middleware standard, which allows distributed nodes to communicate with each other using a publish/subscribe structure. Publishers send their data to specific topics, while the subscribers read data from those topics. The most important property of a DDS is the possibility to specify Quality of Service (QoS) settings. These QoS settings specify the performance or the behaviour of a publisher or subscriber. A subscriber knows what performance or behaviour it can expect from a publisher. DDS is used in event-driven architectures where many events are sent. These events need to be processed to give meaning to it. Therefore, a combination of DDS and Complex Event Processing (CEP) is common. CEP is a technique which executes queries on unordered streams of events, the result of such a query is a complex event. Queries can run for days and continuously produce complex events based on events received days apart.

Within a DDS and CEP system there are many dependencies which are not directly clear: topics are dependent on a data types assigned to them, while subscribers are dependent upon publishers, which are both connected to the same topic. The QoS properties create another way publishers and subscribers are dependent on each other, CEP queries are also dependent on events received from the DDS system.

Logica is commissioned by the police of the region of Groningen to build a proof of concept of the i-Catcher project. In this project the police is interested in using technology to let the police force be more effective; a security system with an underlying architecture which is used to send events of all sorts. These events come from sensors placed in the cities or even on the internet, for example; panic and aggression sensors, license plate scanners, but also Twitter crawlers. The events are then received by an event processing system which should detect interesting patterns for the police. In the preliminary literature survey a combination of Data Distribution Service (DDS) and Complex Event Processing (CEP) seems like a promising combination for the i-Catcher system. Therefore, in this thesis another proof of concept with DDS and CEP is built, to perform tests on during the case study.

When this i-Catcher software system with DDS and CEP is delivered to the customer it will be started and taken into a production environment. The police will become dependent on the system which makes downtime undesirable. The next phase of the software lifecycle starts: software maintenance. In this period the system will be adapted to the ever-changing

requirements of the police force. The policing, laws and crimes change, new insights on policing will be acquired, software platforms change, sensors will be added, etc. All sorts of changes are possible and the system should be modified accordingly. Because of the dependencies between the components in the DDS and CEP system, software maintenance is difficult. A developer has to determine which changes can be made online and which changes require downtime. What kind of impact such a change has on the system, is another problem that has to be dealt with. Therefore, the following research questions are formulated.

1.1 Research questions

The literature survey which was conducted preliminary to this thesis, concludes with the research questions, which are adjusted for this thesis. The main research question investigated in this thesis is:

RQ1 To which extent do existing software maintenance principles apply to changing a running software system based on a Data Distribution Service with Complex Event Processing?

This software security system is so valuable to the police that it is not allowed to have downtime. To answer the main research question, three research questions are formulated to support the main research question.

RQ2 To which extent are current software maintenance principles sufficient for a DDS environment?

RQ3 To which extent do subscribers, for example a complex event processing engine, influence the software maintenance principles?

RQ4 To which extent is it possible to replace components in a running DDS without downtime?

In the next section the approach to answer these research questions is explained.

1.2 Approach

To answer the research questions the following steps will be reported in this document. First the problems are investigated and defined, relevant topics are discussed and a software maintenance procedure will be proposed. Subsequently, a prototype of the i-Catcher system with DDS and CEP is designed and implemented. Thereafter, test cases are designed, these test cases are actually software change requests to modify the prototype. The test cases are, using the proposed maintenance procedure, tested to prove this procedure is correct for a DDS/CEP system. The results are discussed and finally conclusions are drawn.

1.3 Overview

This thesis starts with a chapter in which all background and related work are described in chapter 2. Chapter 3 covers the topics of DDS and CEP in more detail in the context of software maintenance. Subsequently in chapter 4 a process for changing a DDS/CEP system is introduced, this process includes a formalisation method to analyse the impact of a software change. A monitoring tool for the DDS system is also introduced in this chapter. In chapter 5 a case study is performed to show whether the process is correct. The tool is used to demonstrate the process is correct. Finally in chapter 6 conclusions are drawn and future work is proposed.

Chapter 2

Background and Related Work

This chapter covers all topics that are of importance to this research: Software maintenance, Data Distribution Service (DDS) and Complex Event Processing (CEP).

2.1 Software maintenance & software evolution

As software systems become larger and increasingly more complex, software engineers are facing difficulties maintaining large software projects [27]. This problem is covered in two research fields: software evolution and software maintenance. The definition of these two research fields:

- ”Software Maintenance is the correction of errors, and the implementation of modifications needed to allow an existing system to perform new tasks and to perform old ones under new conditions.” [11]
- ”Software Evolution is the dynamic behaviour of programming systems as they are maintained and enhanced over their life times.” [5]

Maintenance is the process that happens any time after a new development project is implemented. The software evolution process is examining the dynamic behaviour of systems as they change over time [24]. It is difficult to perform empirical research, as there need to be at least two points in time where data is collected. A project like this will be hard to find, as it is unknown in advance if it will last long enough. For this thesis a prototype of the i-Catcher system with DDS and CEP is built. Besides it being a small system, which causes changes to be too small to gather statistical data from, the timespan of the project is also too short to gather enough empirical data to draw conclusion. Therefore, this thesis will only cover the software maintenance of a DDS and CEP system.

But it is important for every software project to anticipate on the future maintenance work that has to be done on the software. Turver et al. [50] showed cost are estimated at 50% of the total lifecycle cost of software and this percentage is still increasing, while Erlikh et al. [14] estimated the cost of maintenance at 90% of the total cost. In [6] Bennett categorises software maintenance into four classes: adaptive, perfective, corrective and preventive. The classes adaptive and perfective describe the software changes in the software

environment and software changes according to new user requirements. Corrective and preventive contain fixing errors and to prevent problems which may occur in the future. The IEEE has adopted the first three categories in their software engineering glossary [46].

As software maintenance is one iteration of the software evolution process it is important to realise the long-term effects of maintenance performed on software. In 1980 Lehman [26] made some observations which are known as the "Laws of software evolution". These laws are amended over the years and the latest are listed in [27]:

1. Continuing Change
2. Increasing Complexity
3. Self Regulation
4. Conservation of Organisational Stability
5. Conservation of Familiarity
6. Continuing Growth
7. Declining Quality
8. Feedback System

Other research [18, 28, 55] points out that not all laws are observed in the investigated open source projects, but the overall understanding is that 1, 2 and 6 are valid laws. The other laws are difficult to prove and may or may not be true. The software maintenance process should take these effects into account, which can degrade the overall system quality.

Brooks stated the following in [8]: *All repairs tend to destroy structure, to increase the entropy and disorder of a system. Even the most skilful program maintenance only delays the program's subsidence into unfixable chaos, from which there has to be a ground-up redesign.* This would imply that all software projects will end in complete chaos when no redesign is done. This is an unproven statement, as the tested open source projects in [18, 55] showed.

Recent research focuses on software maintenance ontologies, in which a formal definition and structure is considered. For example, Ruiz [44] proposed a model for the maintenance process. Obrst [35] points out that software will be loosely coupled as the evolution of tightly coupled systems makes the software too complex. An example of a loosely coupled system is a Data Distribution Service.

2.1.1 Software change process

Bennett et al. [6] suggests a change mini-cycle from Yau et al. [56] to adapt existing software to new requirements. This cycle consists of the following phases:

1. Request for change
2. Planning phase

3. Change implementation
4. Verification and validation
5. Re-documentation

This process will be used in chapter 4 as a basis to propose a process to perform maintenance on a DDS and CEP system. In chapter 4 the tasks for every step in this process will be discussed.

2.2 Data Distribution Service

The Data-Centric Architecture is an architecture where sending and receiving of data plays a crucial role. A Data-Centric, or Database-Centric Architecture, often involves a central database containing the data, but a decentralised/distributed solution can also apply here. An example of a Data-Centric Architecture is the Data Distribution Service. This architecture implies loose coupling between components [22]. Distribution Service is standardised by the Object Management Group (OMG) and is implemented by several software vendors.

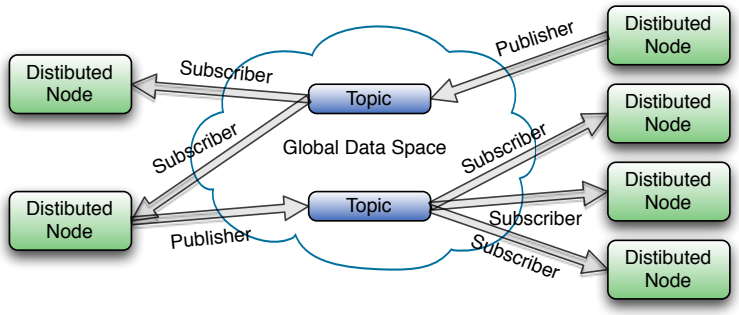


Figure 2.1: A Data Distribution Service overview

A DDS is a data-centric publish/subscribe (DCPS) middleware [37, 38], where publishers write their events to a specific topic and subscribers subscribe to those topics to receive the events, as seen in Figure 2.1. The collection of topics is called the Global Data Space, this is where all the data is stored and made available. The OMG DDS standard specifies a Quality of Service (QoS), this QoS applies to the topics, but also in general on a DDS application [10]. On each separate topic QoS properties can be set, which include properties to specify for example the reliability and durability. Therefore, publishers know to what specification they need to comply and the subscribers know what to expect. If the QoS agreement is violated the subscriber of the publisher is notified and can take precautions. The specification of the OMG DDS standard is named: 'Data Distribution Service for Real-time Systems (DDS)'. This title mentions the real-time performance part of the specification. Real-time performance is a key part of the specification and therefore, this system is suitable to be used in the aviation industry, military systems and can even be used for robotics. Every system can be connected to the network to publish its data. The clients

interested in certain data subscribe to that topic to receive the data. As this is a real-time publish subscribe middleware, the clients are sure of a guaranteed delivery of the data. If the data is not real-time; the clients are notified. Publishers and subscribers can be virtually anything, for example: a radar station, reconnaissance vehicle or a warship.

2.3 Event Processing

Events are the basis of the i-Catcher system, therefore it is important to have a good understanding of what the exact meaning of the word 'event' is in the context of this system. The definition in [31]:

> *An event is a notable thing that happens inside or outside your business. An event (business or system) may signify a problem or impending problem, an opportunity, a threshold, or a deviation.*

This definition describes when an event occurs, but there is a notion of location and time missing in this definition. Or in [29] in the context of event processing:

> *An object that represents, encodes, or records an event, generally for the purpose of computer processing.*

Both definitions are missing some important information, namely; an event happens on a certain moment in time and an event happens on a certain location. Therefore, this new definition combines these two parameters with the state change parameter into the following definition:

> *An event is a collection of three parameters: physical location or context of the event, time and the state change. State changes happen in the physical world or in a system and are detected by event generators.*

Event generators can be anything, for instance: a temperature sensors detecting changes in temperature or license plate scanners recognising the license plate of a passing car. This definition of an event will be used in this thesis, the two added components, time and location, will be used in the case study. Time is for the event processing required and location is required for the i-Catcher system (see section 5.1).

2.3.1 Event Stream Processing

Event Stream Processing (ESP) is a collection of techniques to support the processing of events. For example: styles of event processing and processing languages. In the next section the three types [31] of event processing are explained (see Figure 2.2): simple event processing, stream event processing, complex event processing. The last section covers event processing languages.

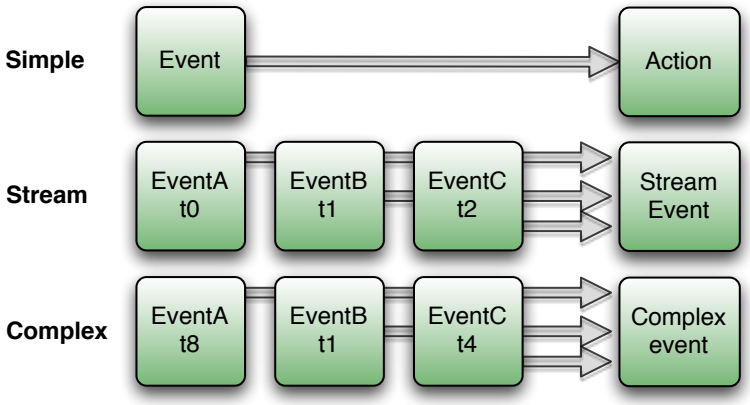


Figure 2.2: Differences between the event processing types.

Simple event processing

This is the most straightforward way of event processing; a single source places an event on the channel [9, 31]. The event processor receives the event and directly initiates the associated action. No event archive is present, the event processor only evaluates one event, therefore this way of processing events has no correlation detection. The process is monitored by investigating the output of the simple event processor.

Stream event processing

Stream event processing receives events from multiple sources and processes the events directly [31]. It is possible to evaluate relations between different events, because the events arrive in order of time. When an event is received it is forwarded to the information subscribers. The events arrive at the event processing in order and in a stream. When comparing complex event processing with stream event processing, the latter is actually a subset of the first. Complex event processing can process streams of events but stream event processing is unable to process complex event processing. This is because stream event processing can only process streams of events that are ordered, but complex event processing can process ordered and unordered sequences of events.

Complex event processing

This is the most advanced way of event processing, because it has access to various datasources. In [59] the datasource is an archive of events and the patterns are matched using this archive. Unlike in [31] the datasource is not an archive of events, but the archive is the data of the enterprise. The main idea is that a complex event will be generated out of multiple sources and is not just the result of only one event, but the result of multiple events. The events do not need to happen directly after each other, they can happen over a long period of time; days or even weeks. Unlike stream event processing the order in which the events arrive is unordered. The detection of those patterns [54] can be done by an Event Process-

ing Language (EPL). The event processing module can also generate new events that can be placed on the bus, so a complex event can also be a new event. After that this event can again be picked up by the event processing module.

2.3.2 Event Processing Language & Continuous Query Language

Continuous Query Language (CQL) is used for queering streams of events [2, 3, 19, 32]. This is a variant of the SQL, structured query language. SQL is used to query data from relational database management systems (RDBMS) in a structured way.

An example of a SQL query is:

```
Select * From Open Where start_price > 100
```

This query is on a table of an auction and returns only the items where the variable `start_price` is greater than 100. The example below is the same but this is a query on a stream. It selects from the continuous insert stream the items with a `start_price` greater than 100 and streams out the resulting output stream.

```
Select Istream(*) From Open[Now] Where start_price > 100
```

This is an example from [3] and demonstrates their implementation of CQL. Although SQL is a standard, it is rarely compatible between different vendors, but it creates some basis to work with. For CQL there does not exist a standard so implementations will vary. Another example that creates a complex event will look like this in Esper [45]:

```
select fraud.accountNumber as accntNum,
       withdraw.amount as amount
from FraudWarningEvent.win:time(30 sec) as fraud,
     WithdrawalEvent.win:time(30 sec) as withdraw
where fraud.accountNumber = withdraw.accountNumber
```

Here two types of events are combined into one complex event that suspects fraud. CQL is a powerful language, it is even possible to detect the absence of events. This can be achieved with a time window specified on a certain stream in combination of a negation function.

The internal structure of a Complex Event Processing Engine (CEPE) is different to regular RDBMS. When data arrives at a CEPE it is not stored on the disk, but the data is checked against the running queries. If a part of a query matches, the query is forked in memory and the engine waits until the rest of the query becomes true. If the query cannot become true anymore, because of a expired time window on the stream, the query is removed from the memory. If the query detects the desired pattern, the result will be returned to the query initiator.

To allow the execution of queries on a CEPE, the engine should be aware of the data types arriving. Most engines use a statement similar to the 'CREATE TABLE' found in RDBMS, in a CEPE this is called 'CREATE SCHEMA'. Beside the dependency of a query on the incoming data, this creates another dependency on the data type. If the developer decides to modify the data type of the DDS system, the CEP queries and the schema should be verified.

2.4 Summary

This chapter has covered various topics which are of importance to this thesis: software maintenance, DDS and CEP. In the next chapter software maintenance is explained for DDS and CEP. The relations of specific DDS and CEP components are described in great detail in relation to software maintenance.

Chapter 3

Software Maintenance with DDS and CEP

In the previous chapter all related work and background information of this thesis is discussed. In this chapter the topics Data Distribution Service and Complex Event Processing are considered in the context of software maintenance.

3.1 Data Distribution Service

Existing event-driven software lacks a way to sent data real-time in a distributed environment. The DDS standard is developed to solve this problem by defining a lightweight middleware protocol and by the addition of a Quality of Service (QoS). The QoS is part of the protocol and specifies on what contract a publisher sends its data and what the subscriber should expect on receiving the data. If this QoS is violated the publisher or subscriber is notified. This addition makes the DDS middleware suitable for use in realtime embedded systems, but also in systems where reliability is very important.

3.1.1 Components of a DDS

The focus of the thesis is the software maintenance within a DDS system, but first it is important to know what components are involved. Thereafter it is possible to conclude which components should be considered. As described before, a DDS consists of publishers and subscribers. Between those two there are topics which receive data from the publishers and subscribers receive data from the topics. So the definition of the data types sent and received is also important component. This is done via an Interface Description Language (IDL) to specify the data in a language and operating system independent way. The core of the DDS evolves around the existence of a Quality of Service (QoS). This is a sort of contract between the publishers and subscribers which should be compatible between these two to be able to communicate. These five components make up the parts of the DDS system that need to be designed to make the system behave as required.

- Publishers

- Subscribers
- Topics
- Data types
- QoS

In the next sections each of these components are discussed.

3.1.2 Publishers & Subscribers

Publishers and subscribers are the processes running in the system. The publishers will often be publishing sensor data which are connected to the process. These publishers can be quite small, e.g. a few classes. It is also possible that publishers use more than one topic to publish data on. On the other side the subscribers receive data from one or more topics. Depending on the task the subscriber is performing, the number of classes can vary from a few to many more. These two can also be combined into one process; a subscriber that reads data from multiple sources, performs a calculation and publishes the result to another topic. Such highly connected processes will increase the complexity of the system and will make the system harder to manage.

3.1.3 Topics

In a DDS application there can be many topics. A topic holds one specific type of object defined by the Interface Description Language (IDL). Publishers write to topics while the subscribers read from the topics. A topic contains samples which are instances of the objects published by the publishers. By default only the last instance of an object is saved, but when a field in the object is specified as a key then this field will act as a sort of primary key which are also found in relational databases to identify a single record. For example: an application has a topic named 'Vehicle_Positions'. Multiple vehicles publish their location to the topic and their license plate is used as a key value. When two vehicles publish their location, not the last sample of all samples will be saved, but the last sample of both vehicles. Other properties that control the timing behaviour of a topic are specified by the Quality of Service preferences.

When designing a distributed publish/subscribe system, topics play an important part as the structure of the network is determined by the topics. As a topic only contains one data type the creation of topics is determined by the creation of new data types, but multiple topics can also contain the same data type. The network complexity increases when the number of topics increase, therefore the developer should carefully consider whether to create or not to create a new topic. The programmer can decide to filter the messages on the subscriber side or to create a new topic with only the requested messages published on it. A filter requires only changes to the subscriber, adding topics requires changes on the publishers and subscribers.

3.1.4 Data types

Objects that can be sent with the DDS are described with an Interface Description Language (IDL) which is standardised by the Object Management Group (OMG). This standard describes the mappings from the IDL data types, like 'char', 'boolean' and 'string', to the language the programmer has decided to use. Most DDS vendors provide an IDL translator to Java, C, C++ and C#. When an IDL is created, the source code can be generated in the desired programming language (see figure: 3.1). The developer can program a publisher in C, which can be a good language for an embedded system. The subscriber does not have to be implemented in the same language, the only condition is that the generated code is both from the same version of the IDL file.

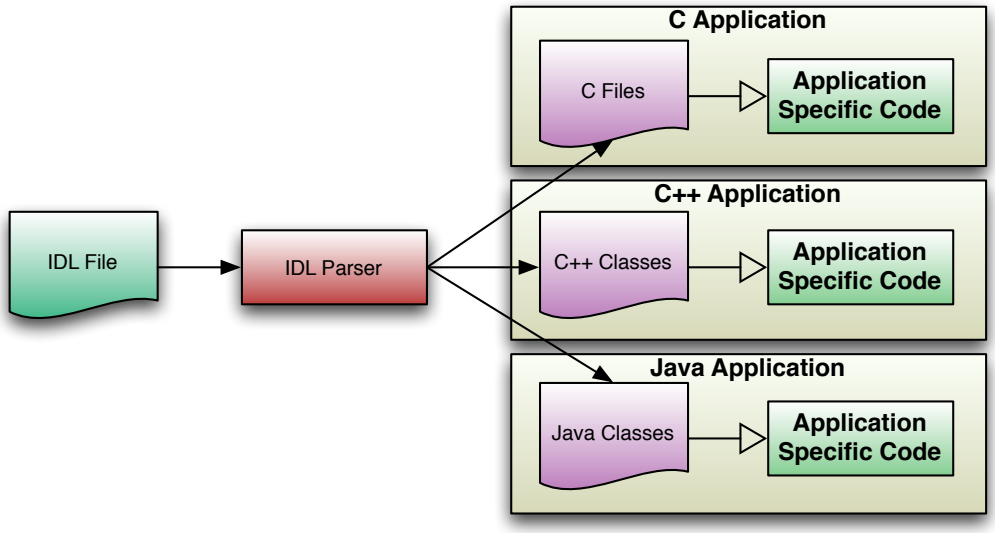


Figure 3.1: How the IDL is processed and integrated in an application.

When modifying a DDS, changes to the data types are inevitable. Here the programmer has two choices: first expanding an existing data type, second creating a new data type, this directly implies a new topic has to be created. In the long term this comes to the following trade-off: complex data types and less topics, or simple data types and many topics (see figure 3.2).

3.1.5 Quality of Service

DDS is a powerful middleware because of the existence of the Quality of Service (QoS) settings. These settings can be defined for each topic individually, but also application wide. Here all properties defined by the OMG DDS standard are discussed.

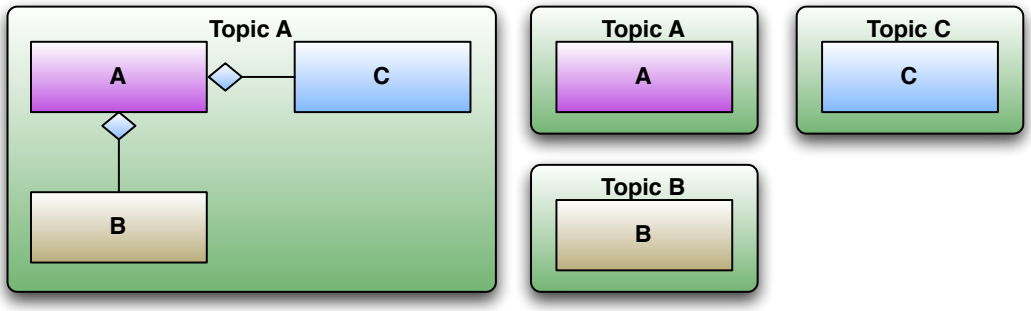


Figure 3.2: Complex data type on the left versus a simple data type in multiple topics.

Reliability

Samples can be send according to two different profiles: best-effort and reliable delivery. When sending messages over an unreliable communication channel, such as Wifi or 3G, messages can get lost. If a message is lost, with the best-effort profile, it will not be received by the subscriber. This method is the fastest and less resource intensive for the CPU and the network. Reliable data delivery profile, is slower and uses more resources, but when messages are lost, the lost message is repaired by the middleware. This property is actually a tradeoff between: speed, use of resources and reliability of the delivery of the messages.

History

This property specifies how many samples of an instance the DDS should save. When a new process connects to the DDS it will receive by default the last sample of an instance, but when a positive integer value is defined, for example 10 for a certain topic, the subscribed processes receive 10 samples per instance when available. The number of samples can be less, because of the fact that there were less samples published by the publisher or when the lifespan of a sample has expired.

Lifespan

The messages received by the DDS will be timestamped and when this timestamp exceeds the time defined by the lifespan property the message is discarded. After some time certain types of messages do not have any meaning or are not allowed to save longer than a certain time specified by the government due to privacy regulations.

Durability

This property specifies if the already published data is stored to deliver to newly connected subscribers. So if a subscriber gets disconnected due to some kind of failure the subscriber will be able to retrieve the missed samples when it reconnects. This will increase the system tolerance to failures, but the publisher has to save more samples in memory.

Liveliness

By activating the liveliness properties subscribers can detect the disconnection of the publishers. It is possible to detect failing publishers during runtime and the subscribers can act on the notification of the disconnection of a publisher.

Ownership and ownership strength

By specifying the ownership strength the DDS only delivers the samples from the publisher with the highest strength value, samples from other publishers are discarded. The detection of a failing publisher is specified by the liveliness property. The ownership strength property allows redundant publishers to cope with failing publishers.

Deadline

If a component is designed to publish data in a periodic way, it can be verified by this property. The subscribers are notified by the DDS that the publisher is missing deadlines and the subscriber can take measures to deal with this situation.

3.1.6 QoS dependencies

When designing an application using DDS it is important to consider, every QoS property for every topic. As can be read in the previous sections the QoS properties do not only make sure by what quality data is delivered within the DDS, but other behaviour like redundancy can be controlled. It is clear that the design of the QoS is an important part of the design process of a DDS.

The QoS part is already very complicated due to the large set of properties that can be defined, it becomes even more complicated because of the fact that publishers and subscribers of the same topic can have different QoS properties which have to be compatible with each other. For example, a deadline property of 500 milliseconds on a publisher is compatible with a subscriber with a deadline property greater or equal to 500 milliseconds. If the deadline property is less than 500 milliseconds, the publishers and subscribers are incompatible and will not be able to communicate with each other.

On the software maintenance side, when the system is changed and becomes larger, it is important to keep track of the QoS specifications. In this thesis a collection of QoS properties, which can be used for one or more topics, will be referred to as a QoS-tuple. A QoS-tuple can be used on multiple publishers and subscribers. The tuple has no relation to the data type. Due to this construction managing the QoS-tuples can be hard, because the programmer should keep track where which tuple is used (see figure 3.3). If a tuple needs to be changed for one topic, the programmer should check if the tuple is not used in other parts of the DDS system.

When the DDS system is running the QoS properties can be changed on the fly, the DDS system does not have to be stopped, but the publishers and subscribers must reload the QoS properties. If this is done a very good understanding of the relations between the

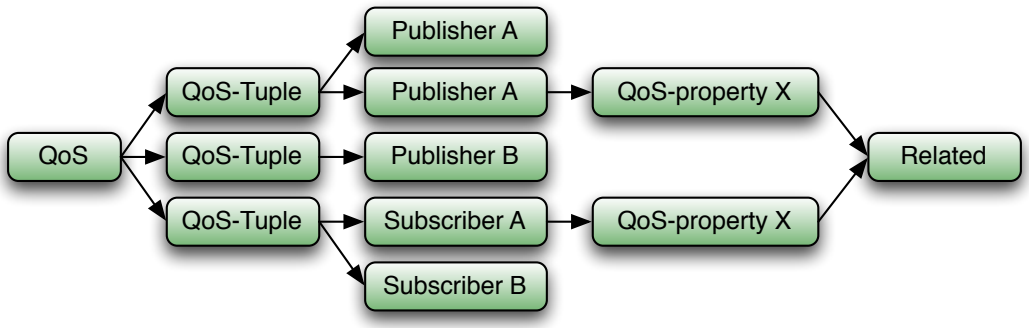


Figure 3.3: Overview of QoS dependencies.

topics, publishers, subscribers and the QoS-tuples is required. If relations are not clear a change to a QoS property can break the whole DDS system.

3.1.7 Software maintenance in a DDS

The maintenance process is about changes being made on a software system after a release of the program. It is likely that this system is running while components of the system are changed. At least the following actions will evolve the DDS while it is running:

- Publisher (add/change/remove)
- Subscriber (add/change/remove)
- Data type (add/change/remove)
- Change QoS properties
- Topics (add/change/remove)
- Failure (network/hardware/software components)
- Change in environment
  - New DDS version
  - New hardware (sensor/platform)

As stated in section 3.1.1, the key components of a DDS system are the publishers, subscribers, data types, QoS and the topics. All of them are included in this list. Failures will change this system unintended, while the 'Change in environment' are intended changes by the developer. The changes can either be initiated by the developer, by an external failure or the components can be constructed to change their behaviour autonomously. These changes have the following negative effects on the system qualities:

- Introduction of dead code

- Traceability becomes more difficult
- Change in system health
- Complexity of the DDS system increases

Dead code can be introduced by changing the structure of a DDS. DDS is an event-driven architecture, therefore subscribers will only react to events, when there are no publishers to a topic the subscriber can be classified as dead code. The same holds for a topic with only publishers, these publishers have no meaning to the system and can be omitted. When a DDS system is running, there is no way to see what the structure is. It is also unclear which path a message travels in the system, the only possibility is to view the output of the subscribers, this makes traceability hard for the developer. When the system structure changes it is difficult to anticipate on those changes as the structure is not visible. For example: if several publishers are removed from the system or are failing, the overall health of the system is unclear. As Lehman stated in [27] the complexity of a software system increases over time, this also holds for the DDS system, which makes the system harder to maintain. Although it is unclear how the complexity of a DDS should be measured, there are multiple problems contributing to this complexity: the structure of the DDS network, which consists of publishers, topics and subscribers. The relation of the data types to the publishers and subscribers, but also the relation of the QoS settings between the publishers and subscribers. All problems stated here, are harder to solve or do only exist because of the lacking of a proper visualisation tool. These are the main problems in a DDS system. The next section covers the addition of a Complex Event Processing Engine to the system and which consequences this has on software maintenance.

3.2 Complex Event Processing

To evaluate how the Complex Event Processing (CEP) is affected by the software maintenance process knowledge about the input and the output is needed. In a DDS/CEP system the input of the data comes from the DDS, so the CEP component is a subscriber from the DDS point of view (see figure 3.4).

A CEP engine does not save the data as a regular relational database, but the engine tries to match incoming data to a part of a query. If this succeeds the query is forked into a separate thread and waits for the rest of the query to become valid. If that happens the results are returned, otherwise the query stops. It is clear that this part of the system is very dependent on the data. On the user side, the system should continuously be adapted to new queries and existing queries should be updated. Such a system will not use a fixed set of queries, but a large set which can vary in size. These are the changes made to the CEP side, partially influenced by the DDS:

- Change of data type
- New data type (new subscriber)

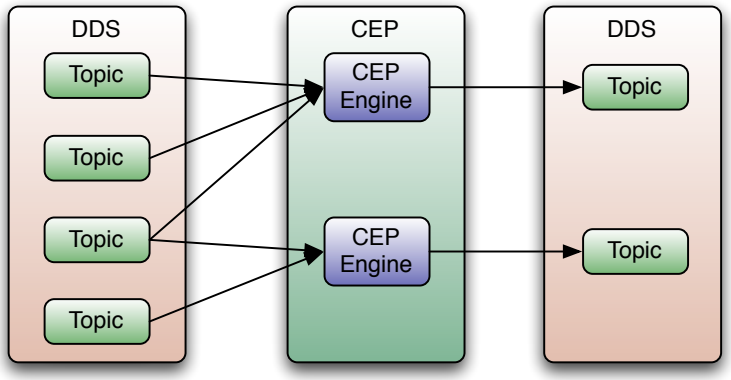


Figure 3.4: Overview how CEP is integrated in the DDS.

- Failure
- New query
- New CEP engine version

It is hard to state this list covers all the changes possible for this CEP. Note that only technical changes are listed and no managerial changes. Changes come from the DDS side of the system, so changes in that layer can influence the CEP layer. On the other side changes can also come from the CEP engine, a new version or a new CEP-query. These software maintenance changes have the following negative effects on the CEP part of the system:

- Long running queries need to be aborted on data type change
- CEP queries need to be validated after changing the data types
- Complexity increases when topics/data types are added
- Scalability, too many queries on one computing node

The biggest problem for maintenance of the CEP engine are the long running queries, as these queries cannot be stopped without data loss. For example; a query which spans multiple days, where a great part of the query is satisfied, is stopped due to maintenance. When the engine is started again, the data which satisfied a great part of the query is lost, because there is no data saved in this system. This complex event will not be detected, so important data is lost. This happens when data types are changed or added to the DDS and these data types are used in CEP queries. When data types are added the complexity increases of the CEP process, this is because another subscriber should added to the CEP engine to receive this data type. This addition of a subscriber or the addition of a CEP query to the CEP engine can cause the computing node, where the engine process is executed, a utilisation which is too high. A solution can be the addition of another node with a new CEP engine executed on it.

3.3 Summary

The specific maintenance properties of both DDS and CEP are clear now. The most important problem of maintenance in a DDS are the dependencies between all the components, but the lack of a proper visualisation tool is also important, as the developer is steering blind. The next chapter discusses the software maintenance process of DDS and CEP in great detail.

Chapter 4

Software Maintenance Process

In this chapter the two technologies, DDS and CEP, are brought together in the research field of software maintenance to analyse how the technologies influence the software maintenance process. These technologies have very specific properties that require attention. The following sections apply to a system that is already running in a production environment and is not allowed to have any downtime. If downtime is inevitable, it should be very short in the range of a couple of seconds. It is not allowed to lose data in the DDS, since it can have great consequences for the output of the system. The change cycle in [6] will be used as a basis to propose a method how a DDS with CEP should be modified. This specific cycle is chosen, because it is concise and to the point. The steps are as follows:

1. Request for change
2. Planning (change impact analysis)
3. Change implementation
4. Verification and validation
5. Re-documentation

In the next five sections the steps are discussed separately to illustrate how every step should be used to change a DDS/CEP system.

4.1 Step 1: Request for change

A request for change can be initiated by users of the system or by the developers themselves. In this step the request is formulated and formalised by the formulas in figure 4.1. This formalisation is used in the next step. To get a clear picture of the possible changes in a DDS/CEP environment, a list of atomic changes and a list of composite changes will be made (see figure 4.2). The atomic changes are changes that have no impact beyond the affected component (see [42]). The composite changes consist of changes that inflict a list of changes (see figure 4.1).

$$
\begin{aligned}
Data_{add} &\rightarrow \langle DataType_{add}, Topic_{add} \rangle \\
Data_{modify} &\rightarrow \langle DataType_{modify}, Topic_{modify} \rangle \\
Data_{delete} &\rightarrow \langle DataType_{delete}, Topic_{delete} \rangle \\
QoS_{add} &\rightarrow \langle QoSTuple_{add}, Topic_{modify} \rangle \\
QoS_{modify} &\rightarrow \langle QoSTuple_{modify}, Topic_{modify} \rangle \\
QoS_{delete} &\rightarrow \langle QoSTuple_{delete}, Topic_{modify} \rangle \\
Topic_{add} &\rightarrow \langle Publisher_{add}, Subscriber_{add} \rangle \\
Topic_{modify} &\rightarrow \langle Publisher_{modify}, Subscriber_{modify} \rangle \\
Topic_{delete} &\rightarrow \langle Publisher_{delete}, Subscriber_{delete} \rangle \\
Publisher_{add} &\rightarrow \langle Pub_{add}, [QoS_{add}] \rangle \\
Publisher_{modify} &\rightarrow \langle Pub_{modify}, [QoS_{modify}] \rangle \\
Publisher_{delete} &\rightarrow \langle Pub_{delete}, [QoS_{delete}] \rangle \\
Subscriber_{add} &\rightarrow \langle Sub_{add}, [QoS_{add}] \rangle \\
Subscriber_{modify} &\rightarrow \langle Sub_{modify}, [QoS_{modify}, CEPQuery_{modify}] \rangle \\
Subscriber_{delete} &\rightarrow \langle Sub_{delete}, [QoS_{delete}, CEPQuery_{delete}] \rangle \\
CEP_{add} &\rightarrow \langle CEPQuery_{add}, [Data_{add}] \rangle \\
CEP_{modify} &\rightarrow \langle CEPQuery_{modify} \rangle \\
CEP_{delete} &\rightarrow \langle CEPQuery_{delete}, [Subscriber_{delete}] \rangle
\end{aligned}
$$

Figure 4.1: Composite changes

4.2 Step 2: Planning

In this phase the developer tries to gain an understanding about what the impact will be on the running system, how much time there will be needed for this change and what components need to be modified. The most important step in this phase is the change impact analysis. Change impact analysis focuses on two areas, traceability and dependability [25]. Only dependability is covered here. In this part a dependency graph is needed to gain an understanding how the existing connections in the system are realised. The goals of this phase are not to break the existing system and also to prevent downtime due to the modifications.

4.2.1 Software change impact analysis

A DDS is a layered system (see figure 4.3), this implies that the changes made on a lower level are propagated to a higher level. Similar to the Ripple Propagation Graphs in [51], certain components *consists-of* other components, making changes have a certain impact. In figure 4.3 the lowest level is the data type, so if the data type is modified, the topics that

$DataType_{add}$ → Add new data type
$DataType_{modify}$ → Modify existing data type
$DataType_{delete}$ → Delete data type
$QoSTuple_{add}$ → Add new QoS tuple
$QoSTuple_{modify}$ → Modify existing QoS tuple
$QoSTuple_{delete}$ → Delete QoS tuple
Pub_{add} → Add new publisher
Pub_{modify} → Modify existing publisher
Pub_{delete} → Delete publisher
Sub_{add} → Add new subscriber
Sub_{modify} → Modify existing subscriber
Sub_{delete} → Delete subscriber
$CEPQuery_{add}$ → Add new CEP query
$CEPQuery_{modify}$ → Modify existing CEP query
$CEPQuery_{delete}$ → Delete CEP query

Figure 4.2: Atomic changes

use this data type need to use the new data type, therefore all publishers and subscribers need to be updated. So a single change results in a chain of events. If the programmer decides to add a new publisher, the only change will be the addition of the new publisher, all other parts are untouched. One exception to this are the QoS properties, if the developer changes the QoS settings, the settings need to match on both sides, on the publisher's and subscriber's side.

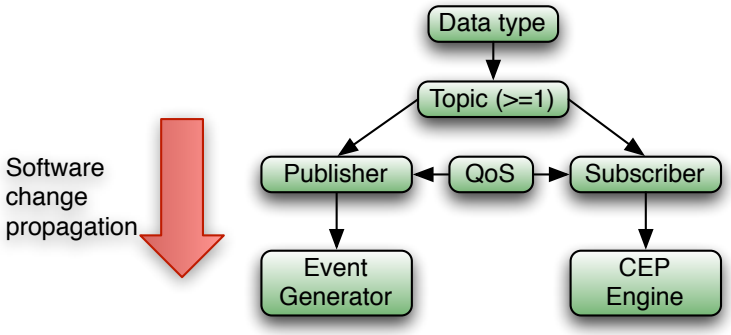


Figure 4.3: Propagation of changes in a DDS/CEP environment.

This dependency problem is actually the same as the software dependency graphs used for change impact analysis [41], the only difference is that the dependency graphs represent the calls within a single piece of software and this dependency graph will represent the flow of data. As this is a data-centric architecture this is not surprising. The directed graph should be used a bit differently than the call graphs. Vertices represent processes or topics and the edges are the publishers and subscribers (see figure 4.4). When, for example, a topic is changed (this implies change to the data type), all processes connected to this vertex are affected, so the direction of the graph should not be used. The increase of the change problem is called the ripple effect [50, 57]. This makes a change to a vertex complicated, as the incoming and outgoing edges are affected and therefore, the vertices connected to them. Figure 4.3 shows what components are affected (the scope), the directed graph shows which processes are affected. A tool which visualises this is very desirable for a developer.

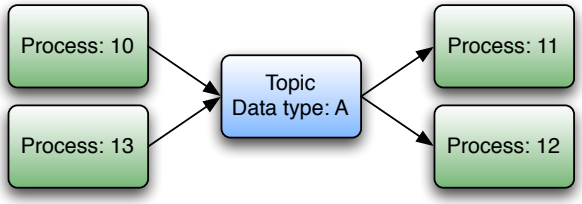


Figure 4.4: DDS network as a directed graph.

4.2.2 Dependencies of changes

Using the formalisation from step one and the dependency graph from this step, the developer can now see which components are affected. The only part that is not directly clear from the formalisation are the 'Topic' rules in 4.1. This rule locates publishers and subscribers of the topics involved with the same data types or QoS settings. The composite changes can be reduced to atomic changes 4.1, which are the components the developer has to alter. Now the developer has a list of DDS and CEP components that are impacted by the change. For each of these components the normal code refactoring and maintenance of the source code apply.

4.3 Step 3: Change implementation

Using the formalisation built in step one and the atomic changes found in step two it is now possible to implement the planned changes. So the only changes that can be done to a DDS and CEP system are the atomic changes from 4.2. Every category of changes will be described below.

4.3.1 Publishers and subscribers

Publishers and subscribers are the easiest modifications that can be made in a DDS environment. Publishers can be started at anytime without making changes to any other module. Optionally QoS properties have to be specified or changed. If the goal is to replace a specific publisher, for example; the sensor is old or not reliable enough and this information is important to the system, the QoS strength property can be used. Set the QoS settings to a lower level and add the new publisher with a higher strength value and the values of the old sensor will not be used in the system. Now the old publisher can be removed from the system. This is a seamless transition from an old publisher to a new publisher. This way of replacing publishers online, can only be done if the old publisher is started with the correct QoS properties, because these properties cannot be changed online.

It is also easy to add new subscribers to a DDS as data is published to a specific topic, a subscriber can subscribe to this topic and receive the data. This will not interfere with existing components of the system. If the data identifies an unique object, like a license plate which needs to be looked-up in the stolen vehicles database, there should only be one occurrence of the data object in the system. If such a subscriber is replaced in the DDS it can be started next to the already running subscriber and subsequently the old subscriber can be terminated. The most flexible part of the DDS is the relatively easy way of managing the publishers and subscribers, when possible the developer should preferably modify these two components instead of the data types of the QoS properties.

4.3.2 Data types

This is the most difficult modification that can be made to a running DDS system. When modifying a data type, all topics which use the data type are affected and all subscribers and publishers to those topics. To change the data type online, the following solution can be used, which is derived from Service-Oriented architecture [23] and may also apply to DDS systems (see figure: 4.5).

Change of data type online: (Versioning)

1. Create new version of data type (V2)
2. Change subscribers/publishers to use the new topic and data type
3. Start new subscribers
4. Start new publishers
5. Stop old publishers
6. Stop old subscribers

In a situation where some downtime is allowed, all publishers and subscribers of the affected topics of the data type change, should be stopped (see figure: 4.6). Thereafter, all publishers and subscribers can be started again. This prevents the existence of multiple

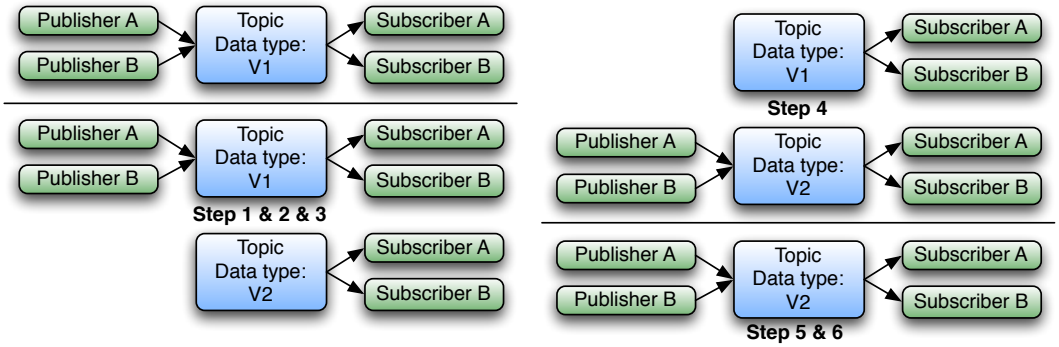


Figure 4.5: From the initial situation to the new situation with versioning.

versions of the data type on the same topic. This can be a problem when the publisher sends out data that is not expected by the subscriber.

Change of data type offline:

1. change data type
2. change publishers/subscribers
3. stop all publishers/subscribers related to data type
4. start all publishers/subscribers related to data type

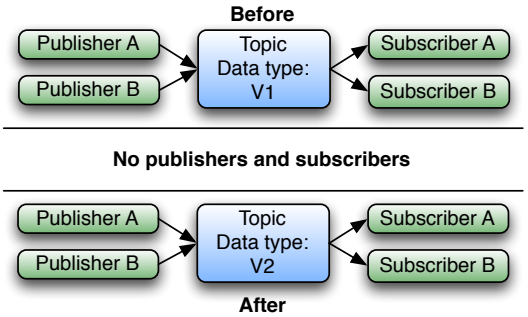


Figure 4.6: From the initial situation to the new situation without versioning.

4.3.3 QoS

Some QoS properties assigned to publishers and subscribers can be changed online. Although this is very easy for the developer, the changes should be considered very carefully to prevent incompatible QoS property combinations on publishers and subscribers. For example: a publisher has a deadline property set at 500 milliseconds and a subscriber has

its deadline property set at 600 milliseconds. If the new settings should be set at 100 milliseconds for the publisher and the subscriber. It is not allowed to set the subscriber at 100 milliseconds first and leave the publisher at 500 milliseconds, this combination is incompatible. The solution is to set the new deadline properties to the publisher first.

This is an example of a QoS property that takes an 'Integer' as a value, there are also properties that work with a 'String' value. For example, the reliability property can have the value BEST_EFFORT or RELIABLE. There is only one combination of this property impossible, a publisher with the BEST_EFFORT property set and the subscriber requesting RELIABLE. This is impossible, the publisher cannot guarantee delivery of messages as the subscriber requests it. This is one example of a property that cannot be changed online.

Change QoS property online:

1. Determine a valid sequence to set the properties
2. Push new QoS to publisher/subscriber

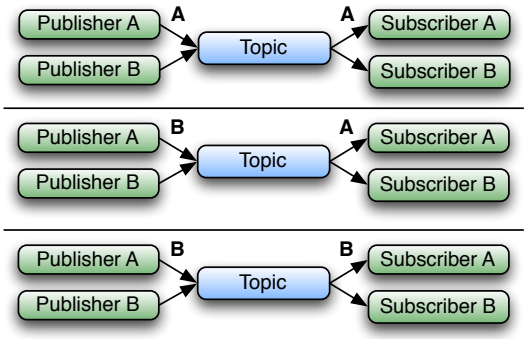


Figure 4.7: Change QoS properties online.

Change QoS property offline:

1. Determine the impact of the changes
2. Prepare and apply the changes
3. Restart the processes involved

4.3.4 CEP

The CEP engine is implemented with subscribers to retrieve the data from the DDS system. If changes need to be applied to the process itself, then the only option is to stop the process. Unless the CEP engine is implemented in a way that allows online modification of the queries. Then the long running queries do not need to be aborted. If the maintenance is of a more structural kind, change of data type, then it is required to stop and start the engine.

If a change to a data type is made and an important CEP query is dependent on the old data type, it is also possible to let the old CEP implementation co-exist with the new

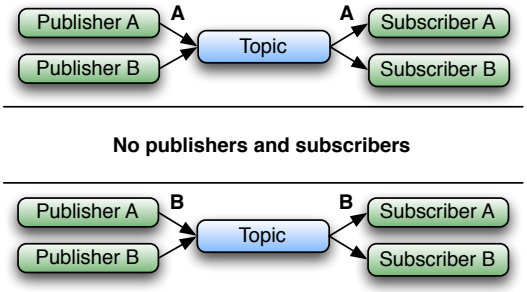


Figure 4.8: Change QoS properties offline.

CEP process which uses the updated data type. The old CEP implementation should be kept so the longest running query can finish. During this period the output of the two CEP engines is the same. This can be avoided if the old engine is not allowed to start new forks of queries, it is only allowed to finish already running queries.

4.4 Step 4: Verification and validation

After performing the changes to the system, itshould be verified and validated against the proposed changes to see if the modification was successful. It can only be done thorough with a tool which observes the system as it is. Enabling the developer to compare the system with the design. When a process is running it is difficult to locate the process where it exists and how it is performing. Therefore, a separate monitoring service is desirable, to bring the status messages together into one central place.

4.5 Step 5: Re-documentation

This is an obvious step to take, but not less important. The developer should document the changes in a log and update the system design documents. This makes the next change easier to perform because it is clear where the description about the running software system is found. Anomalies found during the update should also be logged.

4.6 DDS Monitor

Most of these steps are more difficult than needed, because of the lack of visualisation of a running DDS system. There is no way to imagine what this system looks like when it is running. All sorts of problems can happen; modifying QoS settings which are not all known to the developer, forgotten publishers and more. To solve this, a good understanding of the inner workings of the DDS is needed. DDS knows for itself where the data is, otherwise the whole DDS system could not function, to do this it keeps track of publishers and subscribers. What actually happens is that there is a built-in topic where the all the

publishers and subscribers write data to containing information on what topic they are connected to with which QoS properties. So, if this information is used, the DDS network can be visualised by creating a graph with process id's and topics as nodes with the connections between them as edges. This is one of the contributions of this thesis, which should ease the maintenance of a DDS for the software developer.

4.6.1 Requirements

This tool should help the developer maintaining a DDS system, but also support the case study to perform the experiments. Therefore, the following requirements are constructed:

- Visualise processes, topics, publishers and subscribers as a graph
- Display QoS information per publisher/subscriber
- Display data type usage in topics
- Highlight where a specific data type is used
- Highlight edges with same QoS settings

The requirements come from the steps for the change process, in those steps some extra help from a visualisation tool is desired. Especially because of the fact that it is unclear what the structure is when the system is running.

4.6.2 Results

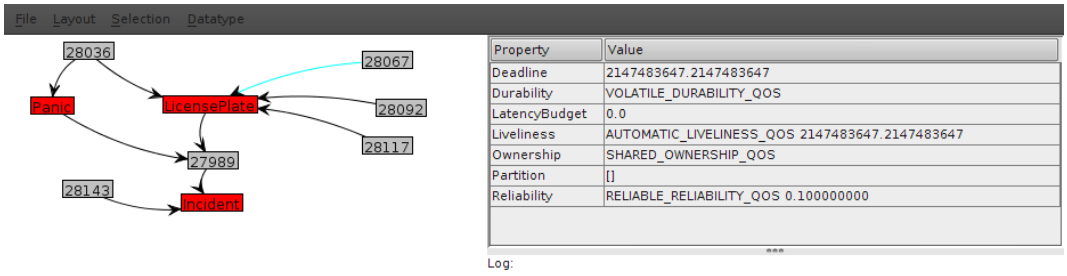


Figure 4.9: Screenshot of the monitoring tool.

With this tool it is now possible to see (figure 4.9) the structure of the DDS. This figure shows a DDS network with seven processes and five topics. It shows a rather small network, the graph already looks complicated, but not as complicated before, because the developer only had console output on various nodes in the network and therefore no complete picture.

Changes in structure are now visible, a developer can now make a better prediction on what the consequences of his changes are. Changes to the data types are harder to visualise, but if the developer selects the data type from the menu, all the topics, publishers and subscribers will be highlighted, to enable the developer to see what the consequences of a

data type change will be. Another function to help a developer with changing a running system is the QoS selector. This function highlights all edges with the same QoS settings as the one the developer selects. In the normal selection mode the developer can select the edges to see the individual QoS information or when a topic is selected to see the data type associated to that topic.

Dead code in a software program are lines of code that are unreachable, as there are no paths to that part of the program. This is the same in a DDS, when a subscriber is connected to a topic with no publishers, this is dead code, because the subscriber will never act on incoming data. The same holds for publishers, who publish to a topic which is never read. This type of dead code can now be found visually, another type of dead code is a publisher and subscriber who are connected both to the same topic, but no data is sent by the publisher. In the tool the publishers and subscribers will look connected and active, but actually both the publisher and subscriber can be removed from the network.

Traceability is improved by use of the visualisation, it is possible to see what might have happened. But, when the DDS system is very large and contains cycles, the visualisation is a tool, not the solution for the traceability. Change in system health is hard to monitor from a system point of view. It is possible to see the structure change of the system, but it is not possible to see the impact of the changes in the behaviour of the publishers and subscribers.

This tool enables developers and researchers to see what is actually happening within the DDS system. Also QoS information is accessible, so impact of changes can now be calculated more easily. In the case study this tool will be used to see the situation before and after the change. First the developer can see the actual situation, thereafter the developer can use the tool to se if his changes have the proposed effect on the DDS.

4.7 Summary

Now the process to perform changes to a DDS/CEP system are clear, but is the process correct and will it perform as expected in the case study? Below is a small summary listed of the steps with the related methods, tools and procedures for each step.

1. Request for change: formalisation method, composite changes
2. Planning: atomic changes, dependency graph, ripple effect, tool
3. Change implementation: atomic changes, tool
4. Verification and validation: atomic changes, composite changes, tool
5. Re-documentation: system design, change log, tool

In the next chapter the correctness of the procedure is demonstrated by means of a case study. The tool will be used for verification of the test results.

Chapter 5

Case study

The design and implementation of the case study are described in great detail in this chapter and in appendix B. This is because there is not much literature on how to design and implement DDS-like systems. The design and the way the components are implemented are also of importance, because this can affect the software maintenance process. As this is a qualitative case study [12, 17, 30, 58], as the case study contains only one case, the details of this specific case can affect the results. The goal of this case study is to validate the software maintenance process introduced in chapter 4. The tool created in this same chapter is used to visualise and validate the results of the tests.

5.1 i-Catcher

In this chapter a case study is performed on a prototype of a system which is developed by Logica and commissioned by the police of the region of Groningen. Relevant topics are covered here to understand this i-Catcher system. The original design uses an Enterprise Service Bus (ESB) instead of the DDS used in this case study. The prototype with DDS and CEP, used in this case study, is designed and implemented especially for this case study.

The aim of this project is to let the police work proactively and more efficient. To realise this idea, a central communication network is created where events are sent. These events can come from many different sources, for example: sensors, Twitter and license plate scanners. These event generators connect and send their events to the central system. These events do not have much value on their own, but when the events are combined this information can be very valuable to the police. When all this information is combined it is possible to create knowledge about the current situation in the city. This process is called situational awareness [7, 21].

5.1.1 Control room

The control room is the central link in the communication between the police officers on the streets and the incident reporters which can either be civilians or police officers themselves. When these incidents are accepted by the intake process, which is handled by the teleservice operators, the incidents are prioritised and forwarded to the dispatch process. Prioritisation

Figure 5.1: i-Catcher logo

is done by a set of standardised procedures. When lives are in danger a prio-1 is issued and when there is less priority required prio-2 and prio-3 are used. The dispatcher prioritises the incidents again, based on the capacity and priority a police unit is deployed. Within this process there is also the possibility to scale-up the incident. This happens when the incident is not a routine police incident, the incident will be classified as a GRIP (coordinated regional incident management). In contrast to the prio classifications the GRIP numbering of importance is completely opposite; GRIP 1 is the lowest priority while GRIP 4 is the highest.

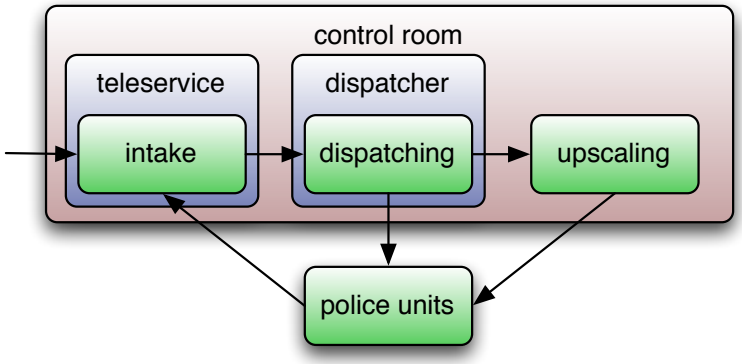


Figure 5.2: Overview of the control room process.

The only input of the control room are the phone calls from the civilians and from police officers. The i-Catcher system adds a third possibility here: the digital alerts. Preferably, these alerts will be forwarded directly to the nearest available police unit. By bypassing the dispatcher and the teleservice operators, a lot of time can be saved before a police unit is deployed, the dispatcher will act as a supervisor for the digital alerts.

5.1.2 Network enabled capability

The i-Catcher project has great similarities with the Network Enabled Capability (NEC) project of the United Kingdom Ministry of Defence. The British army noticed they where generating an increasingly amount of data from various sources. Sources are for example: data from fixed or aircraft mounted radar installations, tanks and reconnaissance vehicles. This project investigates how to get the right information at the right time in the right place.

A similar project is initiated by the United States army and is called: Network-Centric Warfare (NCW) or Network-Centric Operations (NCO). NEC, NCW and NCO share the same common goal [16] which is the sharing of data by using a robust network, thereby creating a shared situational awareness (see next section). Shared situational awareness should increase the speed of command which results in an effectiveness of the forces. Thereby, all forces use the same Common Operational Picture (COP) of the battlefield to enable every unit to see all relevant information.

The way the policing works is similar to the army. There are incidents to which immediate reaction is required and in the long run a higher goal should be achieved. The army and the police both have information at their disposal. By using these sources of information they want to achieve their goals. They both work in a hierarchical organisation which is geographically distributed. The gathering of the information is also geographically distributed. This means the input to the system is not performed on one location, but on multiple locations which can be stationary or moving. Thereby, the soldiers in the field are not interested in the information that is not relevant to them, only the regional information is of importance.

5.1.3 Situational awareness

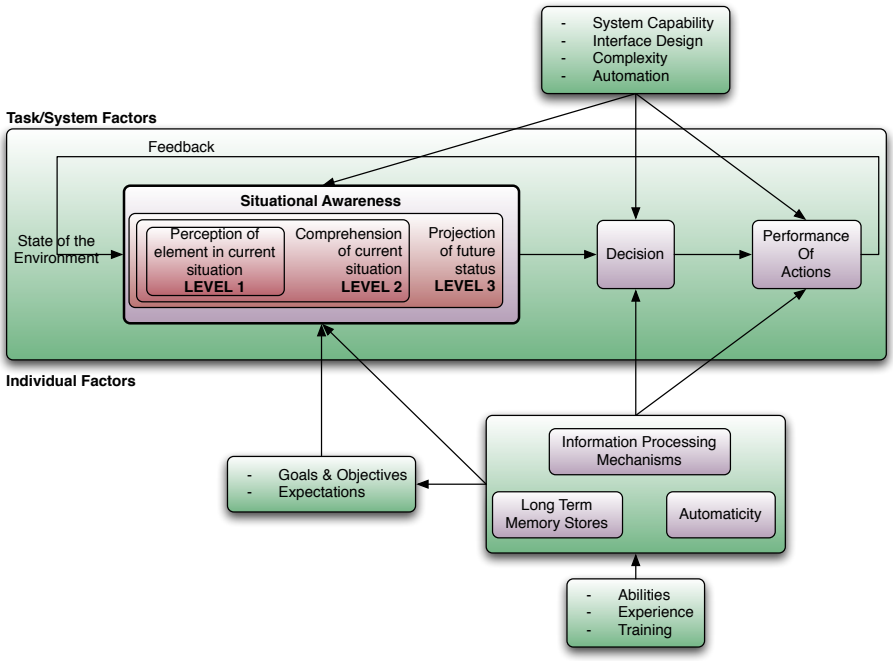


Figure 5.3: Overview of the Situational Awareness process in [13].

Situational awareness (SA) is a research domain where the perception of the environment and decision making is studied. What is happening now, what will happen in the near future and how to react on this information. SA is used in aviation, air traffic control, power

plants, military command and control, public safety (Fire and Police) and even for operating a car [7, 13, 21].

There are three levels of SA [13]; perception, comprehension, projection. The first level of SA is the most basic one, is understanding the environment. Distinguish people, houses, roads, events and their current states in their environment. The states can be a location, action or condition. The second level combines the information of the first level into a higher level of abstraction. This is done by recognising patterns and interpretation of those patterns to understand how this will influence the objectives and goals of the system. See figure 5.3 for the complete overview of the SA process. These objectives and goals are used to let the system make decisions to influence the environment to achieve these objectives and goals. The last level, projection, calculates what the results are when certain actions are taken. What actions have the best results to accomplish the objectives or goals. This level looks into the future to calculate what is expected to happen by making certain decisions.

The i-Catcher system is a kind of SA system, as the first two levels will be used. In the perception level, the system receives events of the environment. These messages are used in the second level, comprehension, to detect patterns within these events that can be recognised as a higher abstraction level of knowledge. Level three will not be considered for the i-Catcher system. As mentioned before this process is used in the military control structure named: command and control (c2). This is the control structure which is used by armies all over the world. This control structure can be supported by the Network Enabled Capability (NEC) platform.

5.2 Background

Chief inspector Elle de Jonge leads the Research and Innovation department at the police district Groningen. With his department de Jonge is responsible for the project 'Slimmer Veilig'. This project is divided in three sub projects: Blue Light Car, PDA Alert and nodal orientation. The goal of the Blue Light Car project is to design the next generation of police vehicles. Equipped with Automatic Number Plate Recognition (ANPR) and streaming video directly to the control room. PDA Alert is an internationally known project where the policemen patrolling the streets get information based on their current location. Nodal orientation is a larger national project where data of multiple license plate scanners are combined to fight criminals who operate on a national level. Besides these projects de Jonge does research on applying new technologies to the current police work. To make the work of the police easier and more efficient.

In 2009 de Jonge supervised the Master thesis of Tom Wassink of the University of Groningen. This research focused on the consequences of an autonomous alert system to process e-alerts. He investigated how e-alerts should be integrated into the current workflow of the control room. He discovered that e-alerts can save valuable time in the control room. This valuable time can be saved on both prio-1 and prio-2 incidents reported to the control room. Another way of improving the reaction time of the dispatcher is by automating his task and change his role from a controller to a supervisor.

The police in the Netherlands has adopted the concept of 'Intelligence Led Policing'

(Informatie Gestuurde Politie). This concept originates from the United Kingdom [1], the Kent police force developed this concept in a period where their budget was cut and they faced an increase in burglary and automobile theft. The police force also knew that a small number of people were responsible for a high percentage of these crimes. To deal with this situation the Kent police force had their current intelligence gathering capabilities analysed and this analysis revealed that their intelligence capability was passive, poorly resourced, lacking management direction and produced little usable results for the uniformed patrols and detectives. Therefore, the Kent police force started with a project to improve the intelligence capability. Every police unit was equipped with an intelligence unit and these units where integrated into the decision process. In the first year the IT systems where upgraded and in the next three years the police could cut the crime rates by 24 percent.

The police assumes that 'Autonomous Alerting' (Autonoom Alarmeren) can reduce the reaction time to a 'PRIO 1 incident' (Prioriteit 1 melding). A priority 1 incident is the highest level of incidents, at this level people's lives are in danger or there is a chance of serious damage to goods or properties. Right now, the reaction time from the reception of the incident and the reaction by the control room can take up to one minute. These 60 seconds could mean the difference between life and death when someone has a heart attack for example.

The Dutch police force only receives just over a million incidents per year, that is a small part of the estimated five million of the actual crimes committed [4]. Of those five million crimes there are more than four million eyewitnesses, but just over one hundred thousand crimes are solved directly (red-handed). To increase the number of red-handed catches the police need to receive more reports of incidents and receive the incidents on time. Receiving incidents on time is important, because when crimes have to be investigated the number of crimes solved is low and the costs of those investigations are very high compared to the red-handed incidents. Increasing the number of reported incidents can be done by increasing the participation of the citizens and by reporting incidents autonomously. When this is accomplished it is believed that the effectiveness of the police can be improved.

Together with Logica, the i-Catcher system is developed and is now tested in the city of Groningen. This system is a proof of concept built by Logica to demonstrate the police that this system is capable of 'Autonomous Alerting' and 'Intelligence Led Policing'. Currently, the inputs of this system are the so called license plate scanners (ANPR) and the database of stolen cars. The system receives events from the license plate scanners and these license plates are compared with the database of stolen vehicles. When a stolen vehicle is detected the nearest police car is notified. This is an example of 'Autonomous Alerting'. The law in the Netherlands prohibits that data is saved due to privacy regulations, therefore the i-Catcher system is not allowed to store data. It is only allowed to directly alert the police if an incident is detected, so the police can arrest the persons involved red-handed.

In the future the i-Catcher system will receive many events from different sources. For example: cameras, microphones, sensors (gas, temperature, earthquake) and even Twitter crawlers. All these input sources generate events that are sent to a central system and can be received by other components. By processing these events, the police want to achieve their goal of reducing the reaction time and act preventive. At this moment Elle de Jonge is working on a new project called: Community Crime Catch (3C). 3C is a concept in which

abnormalities in the public space are detected in an early stage. The police or other parties can also alerted in an early stage, without the need of a control room.

5.3 Design

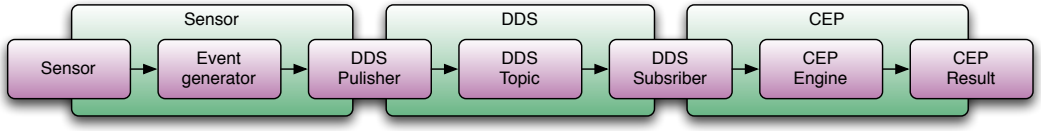


Figure 5.4: Flow of events through the system

The design of the i-Catcher system consists of multiple parts: data types, publishers, topics, subscribers, QoS properties and the CEP engine. An implementation of the DDS standard and the CEP engine has to be chosen, all these decisions are described in great detail in appendix B. This is done because details in this implementation can change the results of the case study. This is the general structure of the system: sensors will send their data using a publisher to a topic and the subscriber receives the data and gives the data to the CEP engine (figure 5.4).

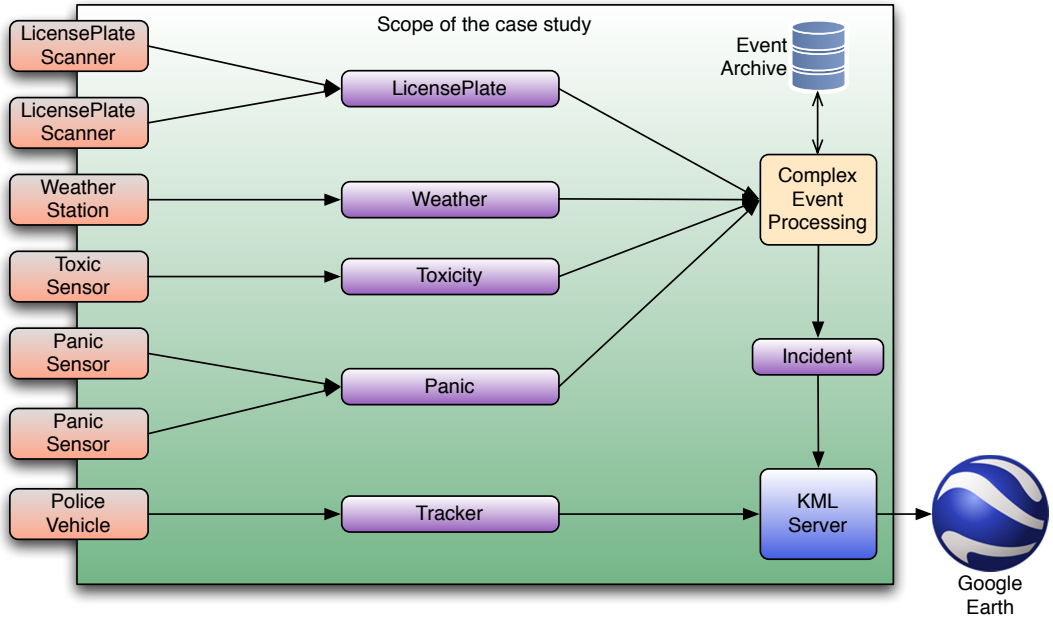


Figure 5.5: Overview of the system design, with an optional kml viewer: Google Earth [20].

The overall system design will look like figure 5.5. On the left side there are publishers, there are two 'LicensePlateScanners', but these can easily be increased by starting more instances of this process, there is at least one instance of every publisher. In the middle

there are topics which receive data from the publishers on the left. The 'Complex Event Processing' engine on the right receives all the messages from the topics. If a complex event is detected by the engine the result is published to the 'incident' topic. To visualise data from the CEP engine a small web server is built which outputs the data to a Keyhole Markup Language (KML) file. This file contains GPS locations of the events detected by the CEP engine.

5.4 Tests

The following tests are identified based on the different composite changes:

Test 1 Publisher addition

Test 2 Subscriber addition

Test 3 Data type modification

Test 4 QoS modification

Test 5 CEP add query

These tests are chosen because they cover all the categories of atomic changes (Publisher/Subscriber/DataType/QoSTuple). This demonstrates the completeness of the whole chain of dependencies. Starting with the high-level change requests from the users or customers, working down to the level of atomic changes using the proposed software change process.

5.5 Case Study

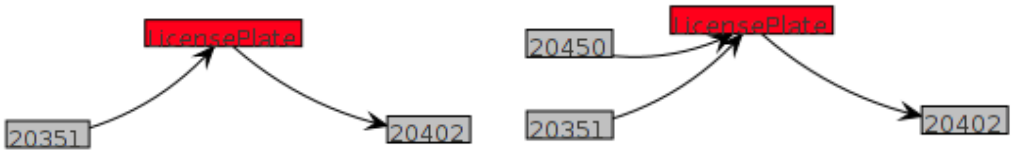
In this case study the tests defined in section 5.4 are used to perform the change steps and formalisation of the change requests of section 4. The goal of the test is to show whether the change steps and the formalisation suffices.

5.5.1 Test 1: Addition of a publisher

In this test a change request is handled that will add a publisher. This is part of the regular maintenance of a DDS system.

Step 1: Request for change

An additional license plate scanner is placed in the city and should be integrated into the system. The license plate scanner will use an already known data type and publish its information on an already existing topic with the QoS set to an existing QoS tuple. Formalisation of the change request: $Publisher_{add} \rightarrow \langle Pub_{add}, [QoS_{add}] \rangle$. This formalisation says: request to add a publisher, this results in an addition of a publisher and optionally an addition of a QoS tuple.



(a) Before (b) After

Figure 5.6: Test 1

Step 2: Plan

The tool is used for the impact analysis to see what the structure of the current system is (see figure: 5.6). In this change an existing data type and topic will be used. Using the formalisation of step 1, it is clear that the only change to the system is the addition of the publisher. QoS is the same as the other publisher of the same topic. The atomic changes are: $Pub_{add} \rightarrow$ Add new publisher and $QoSTuple_{add} \rightarrow$ Add new QoS tuple.

Step 3: Change implementation

Assuming the hardware is already placed and accessible to the developer, the only action that has to be taken is starting up the publisher software on the hardware. The DDS connects to the network and publishes its messages to the topic.

Step 4: Verification and validation

The process started in step 3, it is new possible to see the existence of a new publisher for the existing topic. To validate the output, a known license plate should be sent by the publisher and a subscriber should check if this license plate is published on this topic.

Step 5: Re-documentation

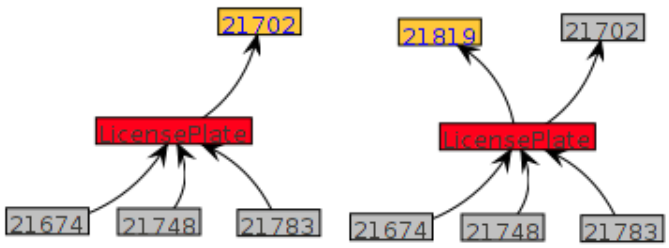
Add an entry describing the change to the system change log. It is important to have a document or database describing what the physical location of the devices is.

Results

The publisher started without interfering with the running system. This system is performing as expected without any downtime. The change process is sufficient for adding a publisher to the system.

5.5.2 Test 2: Subscriber addition

This test simulates a change request which required to add a new subscriber to an existing topic. This is standard software maintenance on a DDS system.



(a) Before (b) After

Figure 5.7: Test 2

Step 1: Request for change

A new subscriber needs to be added which calculates statistics from the license plate data. There are already publishers writing data to this topic and there are also subscribers receiving the data from this topic. Formalisation of this change is as follows: $Subscriber_{add} \rightarrow \langle Sub_{add}, [QoS_{add}] \rangle$. This results in an addition of a subscriber and optionally an addition of a QoS tuple.

Step 2: Plan

The tool is used for the impact analysis to see what the structure of the current system is (see figure: 5.7). In this change a existing data type and topic will be used. The atomic changes are: $Sub_{add} \rightarrow$ Add new subscriber and $QoSTuple_{add} \rightarrow$ Add new QoS tuple.

Step 3: Change implementation

Assuming the hardware is already placed and accessible to the developer, the only action that has to be taken is starting up the subscriber software on the hardware. The DDS connects to the network and receive its messages from the topic.

Step 4: Verification and validation

The process is started in step 3, and it is now possible to see the existence of a new subscriber for the existing topic. To validate the output, a known license plate should be sent by a publisher and the subscriber should be verified whether this license plate is received by the subscriber.

Step 5: Re-documentation

Add to the system change log an entry describing the change. A document of database describing where the physical location of the devices are located is important to have.

Results

Subscriber started without interfering with the running system. This system is performing as expected without any downtime. The change process is sufficient for adding a subscriber to the system.

5.5.3 Test 3: Data type modification

In this change request the most important part of the system is changed, the data type. In the i-Catcher system this is a realistic change, as this system should not violate the Dutch laws.

Step 1: Request for change

A data type has been used for sometime now and additional information should be added to the data type. For example, the Dutch law requires to save some sort of probability value to the license plate scanners data type. This change request is rather complicated, as the composite data type takes several iterations to come to the atomic changes. The main data change is as follows:
$Data_{modify} \rightarrow \langle DataType_{modify}, Topic_{modify} \rangle$.

And using the following rules:
$Topic_{modify} \rightarrow \langle Publisher_{modify}, Subscriber_{modify} \rangle$,
$Publisher_{modify} \rightarrow \langle Pub_{modify}, [QoS_{modify}] \rangle$ and
$Subscriber_{modify} \rightarrow \langle Sub_{modify}, [QoS_{modify}, CEPQuery_{modify}] \rangle$.

To rewrite it to this:
$Data_{modify} \rightarrow \langle DataType_{modify}, \langle \langle Pub_{modify}, [QoS_{modify}] \rangle,$
$\langle Sub_{modify}, [QoS_{modify}, CEPQuery_{modify}] \rangle \rangle$

With an optional composite rule:
$QoS_{modify} \rightarrow \langle QoSTuple_{modify}, Topic_{modify} \rangle$

Step 2: Plan

From the expanded rule of step 1 the following components will be changed:
$DataType_{modify} \rightarrow$ Modify existing data type
$Pub_{modify} \rightarrow$ Modify existing publisher
$Sub_{modify} \rightarrow$ Modify existing subscriber

And from the :
$QoSTuple_{modify} \rightarrow$ Modify existing QoS tuple
$CEPQuery_{modify} \rightarrow$ Modify existing CEP query

Now the tool can be started to identify the components involved (see figure: 5.8), select the data type that will be changed. All edges (publishers and subscribers) will be highlighted and the corresponding process should be modified. CEP queries that use this data type are affected and need to be restarted.

Step 3: Change implementation

The previous steps have inventoried the required changes, now the developer can apply these changes. The most difficult step is the reactivation of the services with no or minimal downtime.

Step 4: Verification and validation

This step should validate whether the structure of the DDS is the same as before the changes, because this modification should not change the structure of the DDS. The developer could start a subscriber on a topic with the changed data type to see the data type works as expected.

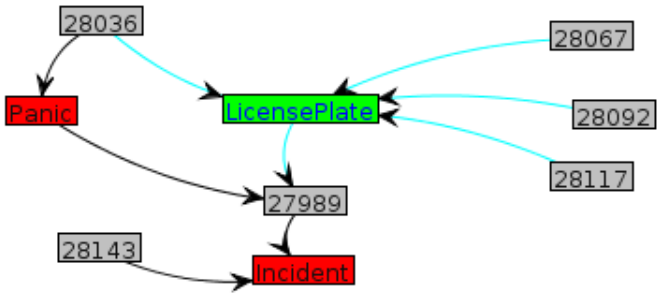


Figure 5.8: Test 3 impact analysis.

Step 5: Re-documentation

The current structure of the data type should be logged and also the topics were the changes where applied. This information can be used on subsequent changes to data types.

Results

The process for changing the data type suffices, but the most difficult part is the reactivation of the services. It is clear that this change is not preferable as this changes almost every part in the whole DDS/CEP environment. If a developer can avoid changing the data type, he should.

5.5.4 Test 4: QoS modification

QoS properties are subject to change if technology improves or technology performs worse under certain conditions. This is standard maintenance of a DDS system.

Step 1: Request for change

The connection of the police cars with the DDS network has significantly improved, therefore the QoS setting of 'BEST_EFFORT' can be changed to 'RELIABLE'. The following formalisation can be derived: $QoS_{modify} \rightarrow \langle QoSTuple_{modify}, Topic_{modify} \rangle$
$Topic_{modify} \rightarrow \langle Publisher_{modify}, Subscriber_{modify} \rangle$
$Publisher_{modify} \rightarrow \langle Pub_{modify}, [QoS_{modify}] \rangle$ and
$Subscriber_{modify} \rightarrow \langle Sub_{modify}, [QoS_{modify}, CEPQuery_{modify}]] \rangle$.

This is expanded to:
$QoS_{modify} \rightarrow \langle QoSTuple_{modify}, \langle \langle Pub_{modify}, [QoS_{modify}] \rangle,$
$\langle Sub_{modify}, [QoS_{modify}, CEPQuery_{modify}]] \rangle \rangle$

Step 2: Plan

The formalisation from step 1 reduces to 3 atomic changes. These changes apply to the QoS settings of the publishers and subscribers of the topics involved. $Pub_{modify} \rightarrow$ Modify existing publisher
$Sub_{modify} \rightarrow$ Modify existing subscriber
$QoSTuple_{modify} \rightarrow$ Modify existing QoS tuple

And optionally:
$CEPQuery_{modify} \rightarrow$ Modify existing CEP query

This QoS change cannot be performed online, therefore the publishers and subscribers have to be stopped and started again. But the sequence this happens in is important, because the combination of a publisher with 'BEST_EFFORT' is incompatible with a subscriber with 'RELIABLE'. So the publishers of the topic should be updated first with the new setting (see figure: 5.9), after that the subscriber should be restarted with the new setting.

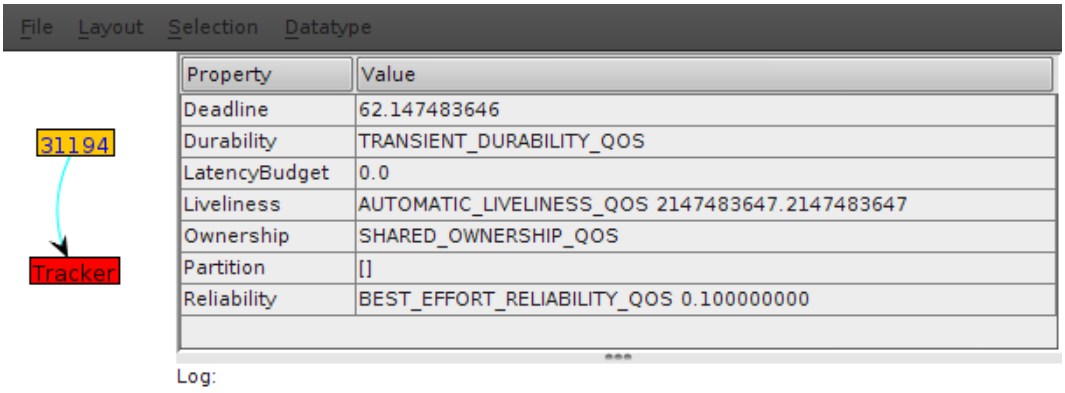


Figure 5.9: Test 4 impact analysis.

Step 3: Change implementation

Changing the QoS property is easy, after that all publishers have to be restarted. When that process is finished, which may take some time, the subscribers should be restarted.

Step 4: Verification and validation

The process of restarting all publishers can be rather time consuming, but the developer should monitor whether all the publishers come back online and the graph looks the same as before. The developer can also verify whether the QoS properties are correct for the affected edges in the tool.

Step 5: Re-documentation

In the log the developer can state why the change is performed and what processes were affected. This can help solve problems detected later on.

Results

The developer has a clear picture of what has to be done in the change process if he follows the steps of the process. The tool proves itself very useful in inspecting the current status of the DDS system.

5.5.5 Test 5: CEP add query

Part of the regular maintenance of the i-Catcher system, will be the addition of new CEP-queries. The Police will think of new possibilities as they discover the power of the Complex Event Processing.

Step 1: Request for change

User or customer requests a new complex event to be added. Query: ”Detect if a car passes the same license plate scanner more than 5 times within 30 minutes.”. This is a realistic complex event from chief officer Elle de Jonge which can be of interest to the police. The actual values of ’5 times’ and ’30 minutres’ may differ, but the general idea remains the same. The following formalisation applies:
$CEP_{add} \rightarrow \langle CEPQuery_{add}, [Data_{add}] \rangle$
We assume that no data type needs to be added. If there is a need to add a data type, this is similar to ’Test 3’ and these steps have to be done here as well.

Step 2: Plan

The formalisation from step 1 produces only one atomic modification.
$CEPQuery_{add} \rightarrow$ Add new CEP query
Using the tool it is possible to see (figure: 5.10) which CEP process is suitable to accommodate the query due to the subscribers of the process.

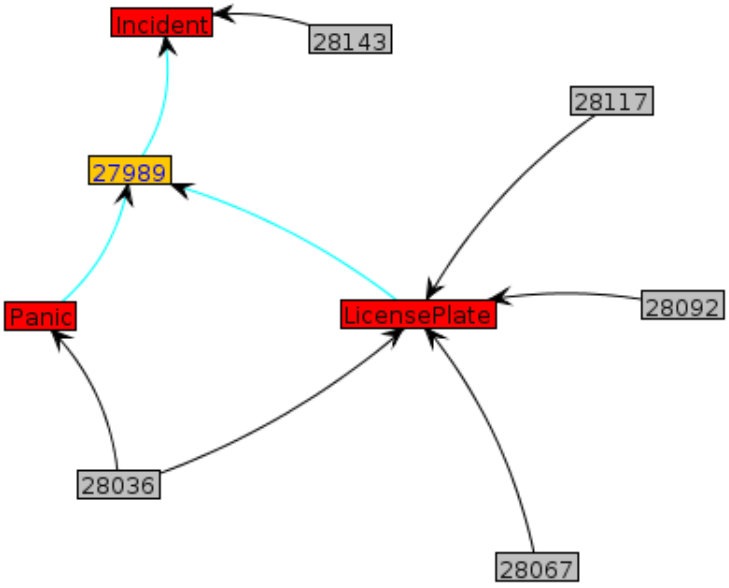


Figure 5.10: Test 5 impact analysis.

Step 3: Change implementation

Because of the single atomic change, the modification is rather small. The developer just has to write the complex event query and the query should be started on the complex event processing engine. Although, it is not advisable to create this query in the production environment. A sandbox environment should exist to let the developer create and test the query.

Step 4: Verification and validation

Because there are no changes to the structure of the DDS the usage of the tool is not needed. To test the CEP-query the developer should inject 5 times the same license plate into the system within 30 minutes to validate this license plate is detected.

Step 5: Re-documentation

For CEP queries it is important to log in which process the query is running and what the maximum running time is of the query. This information can be of use when data types are replaced.

Results

The addition is a high-level modification, therefore the impact on the system is rather small. The hardest part of this modification is step 4, verification and validation, because these

queries can be complex to produce. A sandbox is required for de developer to experiment with the query before taking the query into the production environment.

5.6 Results

Test 1 and test 2 where very easy to perform, this is due to the DDS itself, because a DDS allows publishers and subscriber to be removed or added very easily. The procedure helps the developer to pay attention to the QoS settings. The QoS settings are important, even for this simple change, because with the wrong QoS settings the subscribers will not receive any data. This change was so easy it could be performed online, without interfering with the existing components. Test 3 was the hardest to do, this can be seen in step 1 and 2, in which there are many atomic changes inferred from the change request. Therefore, this change is performed offline. The tool was a great help in this test, all the related components to the change were highlighted, which makes it easy to see the formalisation in the graph. The use of the structured method helps the developer, but as this is a complicated change, the change is still very hard to make. This change effects every level in the DDS/CEP system. Test 4 was also performed offline as this was a change to a QoS property which was not allowed to be changed online. The tool highlighted all the edges with the same QoS properties, to let the developer know which processes need to be adjusted to the new settings. Test 5 is at the highest level, so no changes to the lower parts of the DDS need to be adjusted. The tool can be used to see which CEP engine has the proper subscribers to the data types which are involved in the query. Depending on the implementation the queries could be added online to the engine. This implementation of the CEP engine is not that advanced in the case study, so the online addition of a query was not possible, but this should be possible. Testing of the newly added query is impossible without interfering with the production environment. This is the largest problem of the CEP engine.

The change process did not miss any atomic step, all the atomic changes where present. The change cycle is just five steps and does not contain any redundant or irrelevant tasks. So there is no need for the developer to skip steps. The steps are directly related to the work the developer should be doing, so the process is not distracting the developer of the core of his tasks, applying the change with minimal impact. Minimal impact does not imply all changes can be performed online and without downtime. On the contrary, with some changes downtime is inevitable, changes to data types and to QoS settings are impossible to perform online in a DDS/CEP environment. These five tests cover all of the categories of the composite changes, therefore these tests should be sufficient to show the formalisation is working. The tool was very important in these tests, the current situation could be assessed, the impact of a changes could be analysed and in the end the result could be verified with the expected results.

The purpose of these test cases is to show the completeness of the procedure of changing a DDS/CEP system. The change steps from [6] are sufficient with the additional formalisation of the change request. When modifying the data type it is hard to keep the system running while applying the changes.

As this case study is performed in a single restricted environment, the results of the

case study apply to systems equal to this environment. Systems similar to this environment are DDS systems, small implementations but also larger systems. This is untested, but the atomic changes remain the same and the size of the system has no influence on the atomic changes. The monitoring tool is not tested in large environments, it is possible that this tool has performance issues with such environments. During the tests the monitoring tool proved itself very important, it made the validation possible which was impossible otherwise. The tool is not only useful in the case study, it can also help developers in the implementation or testing phase of building a DDS system.

5.7 Discussion

The change process for a DDS/CEP system has been tested. This is the first test to a single system and to have more confidence in the process, formalisation method and the tool, more testing is required. The most obvious test are performed, but many more are imaginable.

The case study is designed and implemented in a structured way, not in a way that is standardised. UML diagrams are not sufficient for DDS systems. This will be a problem when designing larger systems, the process will become unclear and to complex. First structure was designed based on the data types, subsequently publishers and subscribers where build. In a real project this process will be different.

The formalisation step proves to be accurate when it comes to the reduction into the atomic changes. Although it is a formalisation, the developer still has great responsibility when it comes down to select the affected processes and because the processes are in a DDS it is difficult to match a process in the tool onto a real process. The tool clears the image for the developer, so changes can be made with all the information available to the developer. The tool will not make the change it, only facilitates the developer with the required information.

A problem still remaining is an existence of a test environment such as a sandbox. Complex event queries can be hard to create, but thereafter very hard to test. Mainly timing is difficult here, but the existence of many sources can make the testing very hard.

5.8 Threats to validity

In the following sections the four different validities for case study research are discussed.

5.8.1 Construct validity

This validity verifies if the right operational measures are taken during the case study. First the purpose of the case study should be clear. As stated in the introduction of the case study the purpose is to validate the change process proposed in chapter 4. This case study takes the input for this process, a change request, and subsequently the change is performed as described in the process. The result of a test should be the system with the changes applied. This result should be reached by only changing the required atomic changes and possibly an optional atomic change.

5.8.2 Internal validity

DDS is a standard so the differences between implementations should not affect the results, although the focus of this thesis was not to verify the implementations of the different DDS vendors, there might be different opinions of implementing this standard. Another internal validity concern is the question what leads to the desired results, the proposed procedure or is it the developer conducting the test.

5.8.3 External validity

Another threat to validity is the external validity, which defines to what domain the results can be generalised. In this case study a DDS/CEP environment in which the maintenance procedure is tested. Therefore, the generalisation will not be broader than this environment, although the size of this environment can differ. The size should not influence the results, because the change procedure will still be the same for smaller problems.

5.8.4 Reliability

The case study is completely documented and the complete setup is described in appendix B. This enables developers to reproduce the results of the case study. The complete implementation documentation, including the QoS properties used in the case study, are required to reproduce the results, because minor differences in these properties can change the results completely.

Chapter 6

Summary, Conclusions and Future Work

This chapter first gives a summary and an overview of the contributions of this thesis. Finally, conclusions are drawn and ideas for future work will be discussed.

6.1 Summary and Contributions

Performing research on a topic, which is rather new, is more complicated as literature from related and similar topics have to be mapped on the new problem. The maintenance of DDS/CEP systems is complicated due to the dependencies between the components. During this analysis the need for a visualising tool was clear. Especially in the case study this tool proved itself essential for the analysis of the current situation and the verification and validation after the change. Structuring the change request with five clear steps made the change process easier for the developer. By formalising the request it is difficult to make mistakes in the change impact analysis.

This thesis has several contributions to science, a change process that matches a DDS/-CEP system, a formalisation method for a change request and a tool to visualise a DDS and other properties. This change process is based on a general five step change process, but every step is customised to fit the changes needed in a DDS/CEP environment. The first step has a new formalisation method, which translates a user change request to all required and optional atomic changes that should be performed to satisfy the change request. During the subsequent steps the tool introduced is essential, it visualises the current situation and in the last steps the changes can be verified. The tool also shows the current QoS properties and highlight specific parts of the graph which satisfy a certain condition.

6.2 Conclusions and Future Work

The steps from [6] are valid for a DDS environment, because these steps are rather generic, additional tooling and the formalisation of the change method where required to make these steps work. In the end the steps where sufficient to complete the change requests from the

case study. The change impact analysis is very complicated in a DDS/CEP system, because there are many dependencies. The formalisation of the change request creates a structured method for the developer to analyse what the impact is of a pending change. The CEP engine is greatly affected by changes to the data types in the DDS. Such a change has a large impact, because the queries depend on the data types and when these are changed the queries need to be aborted. Therefore, a developer should prevent changes to the data type in a DDS/CEP environment. When performing software maintenance, it is hard to prevent downtime in a DDS, although in some cases it will be possible to make changes to a running DDS system. Some QoS settings can be changed online, but not all. A change to a property that cannot be changed online, means downtime. When the developer is well prepared, the downtime can be short, as the services only have to be restarted. Changes in data types have the most impact on the system, multiple topics and its publishers and subscribers are affected, this also includes the CEP engine. When using the steps of the change process and the formalisation method the developer can make changes in a structured way. The tool proved itself during the case study as a great help to make the invisible, visible. This DDS system is running when the system is changed, this creates problems with some changes. Changes to data types and to some QoS properties cannot be performed when the system is online. If these parts are changed, only these parts should be stopped and started again. This will have an impact to a directly connected CEP engine. If a developer has to modify components, he should avoid changing components connected to the CEP engine, as results of long running queries will get lost when the CEP engine has to be restarted.

As a DDS has some remarkable features, these features require special attention in the design process. Research on the design process with special attention to structure and QoS properties is desirable. UML is not sufficient to use as a modelling language, because UML expects the system being modelled to be hierarchical. A DDS is not hierarchal at all, it is a flat system: just topics, publishers and subscribers. This can cause problems in large systems with many topics. QoS properties, which are very important for DDS systems, have no place in the UML models. Consistency within the QoS specifications can improve the maintainability of a DDS. Another problem is the lack of tooling, this problem was partially solved with the monitoring tool developed in this thesis, but this tool is not production ready.

This thesis covers the maintenance part of the software lifecycle, the next step is to improve this formalisation onto a lower level. Research on the software evolution becomes interesting also as the process of maintenance is more clear now. During the implementation of the case study it appears that there are many repeating structures. These structures are, for example, a combination of a certain type of publisher and QoS settings serving a specific purpose. When standard software design patterns are extracted, development can be more structured, quality can be improved and mistakes can be avoided. These patterns can ease later software maintenance on the system and can prevent downtime due to QoS settings which cannot be changed online.

Testing and debugging of a DDS is very hard, as there is only the output of the publishers. The tool introduced in this thesis can help debugging, but only to a certain level. The same problem exists with CEP, testing is difficult, mainly because timing is involved. It is doable to trigger the pattern which matches the query, but due to the large variables of input, there can be many false-positives.

Bibliography

[1] Richard Anderson. Intelligence-Led Policing: A British Perspective. *Intelligence Led Policing: International Perspectives on Policing in the 21st Century*, 1997.

[2] A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. *Lecture Notes in Computer Science*, pages 1–19, 2004.

[3] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[4] J.V. Baardewijk and P.V. Os. Meer heterdaadkracht:Aanhoudend in de buurt. 2007.

[5] LA Belady and MM Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[6] K.H. Bennett and V.T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the conference on The future of Software engineering*, page 87. ACM, 2000.

[7] A. Blandford and BL William Wong. Situation awareness in emergency medical dispatch. *International Journal of Human-Computer Studies*, 61(4):421–452, 2004.

[8] F.P. Brooks. *The Mythical Man Month: Essays on Software Engineering, 2/e*. Pearson Education India.

[9] K.M. Chandy. Sense and respond systems. In *CMG-CONFERENCE-*, volume 1, page 59. Citeseer, 2005.

[10] A. Corsaro, L. Querzoni, S. Scipioni, S.T. Piergiovanni, and A. Virgillito. Quality of Service in Publish/Subscribe Middleware. *Chapter in Global Data Management*, 5, 2006.

[11] J. Dvorak. Conceptual entropy and its effect on class hierarchies. *COMPUTER,*, pages 59–63, 1994.

[12] K.M. Eisenhardt. Building theories from case study research. *Academy of management review*, 14(4):532–550, 1989.

[13] M.R. Endsley. Toward a theory of situation awareness in dynamic systems. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 37(1):32–64, 1995.

[14] L. Erlikh and R. Technol. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[15] EsperTech Inc. EsperTech website, March 2010. `http://esper.codehaus.org`.

[16] J. Garstka. Implementation of Network Centric Warfare. *Transformation Trends*, 28, 2004.

[17] J. Gerring. *Case study research: principles and practices*. Cambridge Univ Pr, 2007.

[18] M.W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142. Citeseer, 2000.

[19] L. Golab and M.T.
”Ozsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.

[20] Google. Google Earth website, March 2010. `http://earth.google.com/intl/index.html/`.

[21] J.C. Gorman, N.J. Cooke, and J.L. Winner. Measuring team situation awareness in decentralized command and control environments. *Ergonomics*, 49(12-13):1312–1325, 2006.

[22] R. Joshi. Closer to the edge. *Electronics Systems and Software*, 5(2):14–18, 2007.

[23] N. Josuttis. SOA in Practice. 2007.

[24] CF Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, 1999.

[25] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318. IEEE Computer Society Washington, DC, USA, 2003.

[26] MM Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[27] M.M. Lehman and J.F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.

[28] MM Lehman, JF Ramil, PD Wernick, DE Perry, and WM Turski. Metrics and laws of software evolution-the nineties view. In *metrics*, page 20. Published by the IEEE Computer Society, 1997.

[29] David Luckham and Roy Schulte. Event Processing Glossary - Version 1.1, 2008.

[30] C. Marshall and G.B. Rossman. *Designing qualitative research*. Sage Publications, Inc, 2006.

[31] B. Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2006.

[32] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*. Citeseer, 2003.

[33] G. Muhl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer Verl. Berlin [etc.], 2006.

[34] Object Computing. OpenDDS Object Computing website, March 2010. `http://www.opendds.org`.

[35] L. Obrst. Ontologies for semantically interoperable systems. In *Proceedings of the twelfth international conference on Information and knowledge management*, page 369. ACM, 2003.

[36] Oracle. Oracle CEP website, March 2010. `http://www.oracle.com/technologies/soa/complex-event-processing.html`.

[37] G. Pardo-Castellote. OMG Data-Distribution Service (DDS): Architectural Overview, 2004.

[38] G. Pardo-Castellote, B. Farabaugh, and R. Warren. An introduction to dds and data-centric communications. In *Proceeding of the 23rd International Conference on Distributed Computing Systems Workshops*, 2003.

[39] PrismTech. PrismTech OpenSplice website, March 2010. `http://www.opensplice.com`.

[40] Progress Software Corporation. Progress Software CEP website, March 2010. `http://web.progress.com/en/Product-Capabilities/complex-event-processing.html`.

[41] V. Rajlich. A model for change propagation based on graph rewriting. In *icsm*, page 84. Published by the IEEE Computer Society, 1997.

[42] X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. *ACM SIGPLAN Notices*, 39(10):432–448, 2004.

[43] RTI. Real-time innovations website, March 2010. http://rti.com.

[44] F. Ruiz, A. Vizcaino, M. Piattini, and F. García. An ontology for the management of software maintenance projects. *International Journal of Software Engineering and Knowledge Engineering*, 14(3):323–349, 2004.

[45] N.P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 4. ACM, 2009.

[46] A. September. IEEE Standard Glossary of Software Engineering Terminology. 1990.

[47] StreamBase Systems, Inc. StreamBase website, March 2010. http://www.streambase.com.

[48] Sybase Inc. Sybase Aleri CEP website, March 2010. http://www.aleri.com.

[49] TIBCO Software Inc. Tibco CEP website, March 2010. http://www.tibco.com/software/complex-event-processing/businessevents/default.jsp.

[50] R.J. Turver and M. Munro. An early impact analysis technique for software maintenance. *Journal of Software Maintenance: Research and Practice*, 6(1):35–52, 1994.

[51] R.J. Turver and M. Munro. An early impact analysis technique for software maintenance. *Journal of Software Maintenance: Research and Practice*, 6(1):35–52, 2006.

[52] Twin Oaks Computing. CoreDX DDS Twin Oaks Computing website, March 2010. http://www.twinoakscomputing.com.

[53] UC4 Software GmbH. UC4 Software CEP website, March 2010. http://www.uc4.com/en/products/uc4-cep/complex-event-processing/.

[54] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, page 418. ACM, 2006.

[55] G. Xie, J. Chen, and I. Neamtiu. Towards a better understanding of software evolution: An empirical study on open source software. 2009.

[56] SS Yau, JS Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC'78*, pages 60–65, 1978.

[57] SS Yau, RA Nicholl, J.J.P. Tsai, and S.S. Liu. An integrated life-cycle model for software maintenance. *IEEE Transactions on Software Engineering*, 14(8):1128–1144, 1988.

[58] R. Yin. *Case study research: Design and methods*. Sage Pubns, 2008.

[59] C. Zang and Y. Fan. Complex event processing in enterprise information systems based on RFID. *Enterprise Information Systems*, 1(1):3–23, 2007.

Appendix A

Glossary

In this appendix an overview of frequently used terms and abbreviations is presented.

ANPR: Automatic Number Plate Recognition

CEP: Complex Event Processing

CEPE: Complex Event Processing Engine

COP: Common Operational Picture

CPU: Central Processing Unit

CQL: Continuous Query Language

DCPS: Data-Centric Publish/Subscribe

DDS: Data Distribution Service

DDSI: Data Distribution Service Interoperability Protocol

EPL: Event Processing Language

ESB: Enterprise Service Bus

ESP: Event Stream Processing

GPS: Global Positioning System

IDL: Interface Description Language

IEEE: Institute of Electrical and Electronics Engineers

KML: Keyhole Markup Language

NCO: Network-Centric Operations

NCW: Network-Centric Warfare

NEC: Network Enabled Capabilities

OEM: Original Equipment Manufacturer

OMG: Object Management Group

QOS: Quality of Service

RDBMS: Relational Database Management System

RTI: Real-Time Innovations

SA: Situational Awareness

SOA: Service-Oriented Architecture

SQL: Structured Query Language

UML: Unified Modelling Language

Appendix B

Design and implementation of the i-Catcher prototype

The next sections describe in great detail how the case study is implemented in such a way that a developer should be able to reproduce the case study.

B.1 Selection of a DDS

Where should a selection of a DDS be based on? In the research questions the main focus is on the software maintenance in a software system with CEP and DDS. As DDS is a standard, the initial focus of the selection is the compliance to the standard and the support for the various platforms. The best known vendors of the DDS middleware are: OpenSplice by PrismTech [43], RTI Data Distribution by Real-Time Innovations [39], OpenDDS by Object Computing [34] and CoreDX DDS by Twin Oaks Computing [52]. In the next sections the various properties of the different vendors are compared.

B.1.1 Documentation

The availability of documentation is important to implement the DDS system correctly. A basic level of documentation is too small to use for such a complex system. OpenDDS and CoreDX DDS deliver one PDF file or just a JavaDoc output of the Java API, which is not sufficient. OpenSplice and RTI DDS deliver a complete library of their software with complex implementation examples.

B.1.2 Languages

Due to the use of an Interface Description Language (IDL) multiple can be used in one DDS system. All the DDS implementations contain support for the C++ and Java programming language. OpenSplice and RTI have support for the C# language, and all except OpenDDS have support for the C language.

Properties	OpenSplice	RTI DDS	OpenDDS	CoreDX DDS
Documentation	Good	Good	Basic	Basic
Languages	C/C++/C#/Java	C/C++/C#/Java	C++/Java	C/C++/Java
License	Open/closed	Closed	Open	Closed
DDS version	1.2	1.2	*1.2	*1.2
DDSI version	2.1	2.1	-	2.1
Windows	x86	x86/x64	x86	x86/x64
Linux	x86	x86/x64	x86/x64	x86/x64
Mobile	-	Windows mobile	-	-
Embedded, RTOS	INTEGRITY, VxWorks	INTEGRITY, LynxOS, QNX, VxWorks, WinCE	QNX	QNX, VxWorks

Table B.1: Comparison between different DDS products
* = not fully implemented

B.1.3 License

OpenSplice offers an open source community edition, which does not contain all features. If these features are needed a closed source license is required. RTI and CoreDX are closed source and both require a license to be used. OpenSplice has an open source community version. OpenDDS is an open source implementation and is free to use and modify.

B.1.4 OMG DDS version

It is important to know which version of the OMG DDS standard is implemented. All implementations use OMG DDS version 1.2, but do not implement all the required and optional features. OpenDDS and CoreDX do not fully comply to the DDS 1.2 standard, OpenSplice and RTI DDS fully comply to the standard.

B.1.5 OMG DDSI version

OMG has also standardised the Data Distribution Service Interoperability (DDSI) protocol. This protocol standardises the way DDS implementations communicate over the network. Now different DDS implementations can be used together in one system. This specification prevents the so-called vendor lock-in. Only OpenDDS does not implement this protocol, therefore OpenDDS applications will only be able to communicate with each other. OpenSplice, RTI and CoreDX can communicate with each other, this was presented at the OMG meeting in March 2009 in Washington.

B.1.6 Platform support

When selecting a DDS vendor platform support is important, because this decision can restrict the possible implementations. When a specific platform is used within an organi-

sation and the DDS does not support this platform, the platform or the DDS used should be changed. RTI does support the most diverse range of platforms. Not only the choice of platforms is restricting, also the support for the platform architecture, as the intel architecture is in a transition phase from 32 bit to 64 bit, not all vendors support the 64 bit operating systems. The development environment is running on a 64 bit architecture.

B.1.7 DDS Decision

When adding up all negative and positive properties, two DDS implementations stand out: OpenSplice and RTI DDS. Both implement the full properties of the DDS standard and have full support for DDSI. The open source licence of OpenSplice is interesting, but this version is a restricted version with less functionality than the closed licensed versions. Both vendors are on the same level except for the availability of a 64 bit version. As the development environment uses a 64 bit operating system obviously RTI DDS is the best choice.

B.2 Modelling IDL data types

The data types are the core of the DDS system (see figure 4.3) and this needs special attention to setup. In section 2.3 an event has at least the following properties: time, location and the actual state change. A DDS maintains a time for each message, because the QoS properties use this information to determine a message is still relevant.

```
1  //@copy-java import icatcher.idl.locatable.Locatable;
2  //@copy-java import icatcher.idl.locatable.LocatableTypeCode;
3  //@copy-java import icatcher.idl.locatable.LocatableTypeSupport;
4
5  #include "idl/Locatable.idl"
6
7  struct Incident {
8      long        incident_id;//@key
9      long        priority;
10     string        description;
11     Locatable     location;//@resolve-name false
12 };
```

```
1  struct Locatable {
2      double      latitude;    // latitude
3      double      longitude;     // longitude
4      double    altitude;    // altitude
5  };
```

DDS has no built-in support for locations, so a data type 'locatable' is created with GPS coordinates and altitude. This data type can be used by all other data types specified in the DDS. The state change can be just one value, but more complex combinations of values can be imaginable. The UML representation of the incident data type will look like figure B.1.

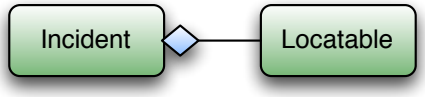


Figure B.1: Relation between data types modelled in UML.

B.3 Modelling QoS properties

In figure 5.5 multiple topics are visible in purple. These topics have different properties, therefore the QoS has to be designed individually for each topic. In section 3.1.5 the QoS settings are discussed and based on this information the table C.1 is created. For some data types send by publishers the last value will only be relevant or the values within a certain timespan. All these properties have to be analysed and the developer should keep in mind that not all QoS properties can be adjusted online. There can be a big overlap between the QoS settings per topic, these settings can be joined together to make the system easier to manage.

B.4 Modelling publishers and subscribers

The class diagram of the publishers and subscribers is described in figure B.2. The wrapper classes are abstract Java classes. The source code of these wrappers can be found in the appendix D.1 and E.1. These abstract classes are implemented for every unique data type, for example the 'LicensePlate' data type, see appendix: D.2 and E.2. Because the subscriber receives data, a listener is needed to get the data to the program itself (see appendix E.3).

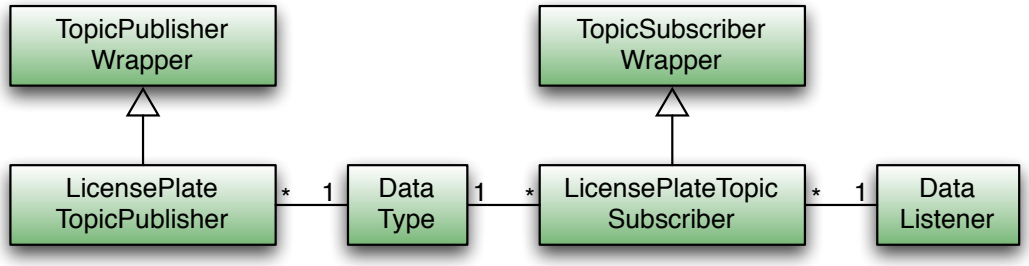


Figure B.2: UML class diagram of the publishers and subscribers.

B.5 Event Processing

The decision of the complex event processing engine depends on several properties. These properties are listed in the table B.2, in the sections thereafter the properties are discussed in more detail. There are a limited amount of software vendors in de CEP market: Oracle [36], EsperTech [15], Sybase Aleri [48], Streambase Systems [47], Tibco [49], Progress

Software [40] and UC4 Software [53]. The last three vendors do not offer a development version of their CEP engine, therefore only the other vendors are considered. Sybase Aleri have had an OEM license agreement with RTI Software, so this is a known combination of software. Esper is a well known open source CEP engine which is written in Java. Finally, the software from Oracle and Streambase are enterprise solutions. In the table B.2 these four software tools are compared.

Property	Esper	Oracle	Sybase	Streambase
CQL Schema	DDS classes	Manual	Manual	Manual
Documentation	Good	Medium	Good	Good
Graphical Editor	No	Yes	Yes	Yes
Integration DDS	Java API	Java API	Java API	Java API
License	GPL v2	Closed	Closed	Closed

Table B.2: Comparison between different CEP engines

B.5.1 CQL Schema

How is the schema created in the CEP engine? The engine is a kind of database, therefore it needs to know what data it should expect and how it will store the data. Two ways of defining schemas can be distinguished; by using CQL statements similar to 'CREATE SCHEMA'. This has the disadvantage that the code created in the DDS part cannot be reused and is separately maintained. This can cause inconsistency between the DDS and CEP engine. The second way is by reusing the classes generated by the DDS part of the project. The IDL files are used to generate the Java classes, which are used to be sent over the DDS topics. On the subscriber side of the topic there should be the CEP engine to process the events. Esper is the only engine which can build the schema from existing DDS classes. This saves a lot of time for the programmer, because there is only one system that needs to be maintained and not two systems which need to be synchronised with each other.

B.5.2 Documentation

Documentation is important to use the software the way it is designed. This will also decrease the development time, because less time is spent on searching for answers. All software vendors provide sufficient documentation to allow a prosperous development process.

B.5.3 Graphical Editor

A graphical editor can be very useful to understand the flow of data to the CEP engine. Such an editor can also help with the programming of the CQL and the schemas. It is not mandatory, but can reduce development time.

B.5.4 Integration DDS

How well and how easy is it to hookup the engine to the DDS system? All products provide a Java API to connect the systems together. For all systems an adapter can be programmed which acts as a DDS 'datareader'. The data received by the adapter is thereafter transferred to the API provided by the software vendors. All these adapters take only 2 classes per DDS topic and one main class in Java.

B.5.5 License

Sybase Aleri prohibits the use of CEP for the purpose of national security, therefore it is not allowed to be used in the context of the i-Catcher system. The Esper project is released under the GNU General Public License version 2. This implies the source code is available to the public. Oracle and Streambase allow the use of their licensed software for the purpose of national security. The source code of these products are closed, therefore not available to the public.

B.5.6 CEP engine decision

To answer the research questions all the CEP engines suffice. Sybase Aleri is not an option, because the license prohibits the use of their software for the purpose of national security. When an engine takes much effort to integrate into the DDS system a lot of time can be wasted, therefore the level of integration into the DDS part of the project is the decisive factor. Esper is the engine which can be easily integrated into the DDS environment, because of the usage of the DDS classes instead of building a schema file. Therefore, Esper will be used as a CEP engine in this case study.

B.6 Setup

The design and the implementation described in the previous sections are used for this case study. The DDS implementation used is the release 4.4d of Real-Time Innovations, together with the Esper CEP engine version 3.4.0. The implementation is realised on an Ubuntu 10.04 and 9.10 with Eclipse as the integrated development environment (IDE). Both the i-Catcher system and the DDS monitoring tool are implemented in and running on Java 1.6.

Appendix C

QOS table

See next page for QoS table.

		LicensePlate	Weather	Sensor	Tracker
Reliability					
	Best effort, Reliable	Reliable	Best effort	Reliable	Best effort
	Max blocking time	-	-	-	-
History					
	Keep last, Keep all	Keep last	Keep last	Keep last	Keep last
	Depth	100	6	10	10
Lifespan					
	Duration	10 min	60 min	60 min	10 min
Durability					
	Volatile, Transient local, Transient, Persistence	Transient	Transient	Transient	Transient
	Direct communication	true	true	true	true
Liveliness					
	Automatic, manual by participant, manual by topic	Automatic	Automatic	Automatic	Automatic
	Duration	-	-	-	-
Ownership					
	Shared, Exclusive	Shared	Exclusive	Shared	Shared
Strength					
	Value	-	multiple	-	-
Deadline					
	Period	-	15 min	5 min	1 min

Table C.1: QoS properties used in the case study

Appendix D

Publisher source code

Listing D.1: An abstract Java class for the publishers.

```
1  package icatcher.wrapper.publisher;
2
3  import icatcher.profiles.Util;
4
5  import com.rti.dds.domain.DomainParticipant;
6  import com.rti.dds.domain.DomainParticipantFactory;
7  import com.rti.dds.infrastructure.RETCODE_ERROR;
8  import com.rti.dds.infrastructure.StatusKind;
9  import com.rti.dds.publication.DataWriter;
10 import com.rti.dds.publication.DataWriterListener;
11 import com.rti.dds.publication.LivelinessLostStatus;
12 import com.rti.dds.publication.OfferedDeadlineMissedStatus;
13 import com.rti.dds.publication.OfferedIncompatibleQosStatus;
14 import com.rti.dds.publication.PublicationMatchedStatus;
15 import com.rti.dds.publication.Publisher;
16 import com.rti.dds.publication.PublisherListener;
17 import com.rti.dds.publication.ReliableReaderActivityChangedStatus;
18 import com.rti.dds.publication.ReliableWriterCacheChangedStatus;
19 import com.rti.dds.topic.InconsistentTopicStatus;
20 import com.rti.dds.topic.Topic;
21 import com.rti.dds.topic.TopicListener;
22
23 /**
24  * Wrapper for the publisher
25  * @author tom
26  */
27 abstract class TopicPublisherWrapper implements PublisherListener,
      DataWriterListener, TopicListener{
28
29   private DomainParticipant participant;
30   private DataWriter datawriter;
31   private int domainid;
32   private String topicname;
33   private String qosname;
34
35   abstract void registerType(DomainParticipant participant);
```

```
36    abstract String getTypeName();
37    abstract void writeDataToTopic(Object data, DataWriter datawriter);
38
39    public TopicPublisherWrapper(int domainid, String topicname, String
          qosname){
40      this.domainid = domainid;
41      this.topicname = topicname;
42      this.qosname = qosname;
43
44      // load QoS profiles
45      Util.setProfile();
46
47      // start the
48      participant = getParticipant();
49      if (participant == null) System.err.println("! Unable to create DDS
            domain participant");
50
51      registerType(participant);
52
53      Topic topic = getTopic();
54      if (topic == null) System.err.println("! Unable to create topic " +
            topicname);
55
56      Publisher publisher = getPublisher();
57      if (publisher == null) {
58        System.err.println("! Unable to create DDS Publisher");
59        throw new RuntimeException("Publisher creation failed");
60      }
61
62      System.out.println(this.getClass().getName()+": Creating the data
            writer...");
63      datawriter = getDataWriter(publisher, topic);
64      if (datawriter == null) System.err.println("! Unable to create DDS
            data writer\n");
65    }
66
67    /**
68     * Write object to this topic
69     * @param data
70     */
71    public void writeData(Object data){
72      try {
73        writeDataToTopic(data,datawriter);
74      } catch (RETCODE_ERROR e) {
75        System.err.println("! Write error " + e.getClass().toString() + ":
              " + e.getMessage());
76      }
77    }
78
79    /**
80     * Close this topic
81     */
82    public void closeTopic(){
83      if(participant != null) {
```

```
84          participant.delete_contained_entities();
85          DomainParticipantFactory.TheParticipantFactory.delete_participant(
                participant);
86        }
87        DomainParticipantFactory.finalize_instance();
88      }
89
90      // Helper methods
91      public DomainParticipant getParticipant(){
92        if(participant==null)
93          participant = DomainParticipantFactory.get_instance().
                create_participant(domainid, DomainParticipantFactory.
                PARTICIPANT_QOS_DEFAULT, null, StatusKind.STATUS_MASK_NONE);
94        return participant;
95      }
96
97      private Topic getTopic(){
98        return participant.create_topic(topicname, getTypeName(),
              DomainParticipant.TOPIC_QOS_DEFAULT, this, StatusKind.
              STATUS_MASK_NONE);
99      }
100
101     private Publisher getPublisher(){
102       return participant.create_publisher(DomainParticipant.
              PUBLISHER_QOS_DEFAULT, this, StatusKind.STATUS_MASK_NONE);
103     }
104
105     private DataWriter getDataWriter(Publisher publisher, Topic topic){
106       return (DataWriter) publisher.create_datawriter_with_profile(topic, "
              iCatcherLib", qosname, this, StatusKind.STATUS_MASK_NONE);
107     }
108
109     private void outputStatus(String status){
110       System.out.println(status);
111     }
112
113     public void on_liveliness_lost(DataWriter arg0, LivelinessLostStatus
            arg1) {
114       outputStatus("liveliness lost");
115     }
116     public void on_offered_deadline_missed(DataWriter arg0,
            OfferedDeadlineMissedStatus arg1) {
117       outputStatus("offered deadline missed");
118     }
119     public void on_offered_incompatible_qos(DataWriter arg0,
            OfferedIncompatibleQosStatus arg1) {
120       outputStatus("offered incompatible QoS");
121     }
122     public void on_publication_matched(DataWriter arg0,
            PublicationMatchedStatus arg1) {
123       outputStatus("publication matched");
124     }
125     public void on_reliable_reader_activity_changed(DataWriter arg0,
            ReliableReaderActivityChangedStatus arg1) {
```

```
126       outputStatus("reliable reader activity changed");
127     }
128     public void on_reliable_writer_cache_changed(DataWriter arg0,
            ReliableWriterCacheChangedStatus arg1) {
129       outputStatus("reliable writer cache changed");
130     }
131     public void on_inconsistent_topic(Topic arg0, InconsistentTopicStatus
            arg1) {
132       outputStatus("inconsistent topic");
133     }
134   }
```

Listing D.2: An implementation of the abstract publisher Java class for the LicensePlate data type.

```
1  package icatcher.wrapper.publisher;
2
3  import icatcher.idl.licenseplate.LicensePlate;
4  import icatcher.idl.licenseplate.LicensePlateDataWriter;
5  import icatcher.idl.licenseplate.LicensePlateTypeSupport;
6
7  import com.rti.dds.domain.DomainParticipant;
8  import com.rti.dds.infrastructure.InstanceHandle_t;
9  import com.rti.dds.publication.DataWriter;
10
11 public class LicensePlateTopicPublisher extends TopicPublisherWrapper {
12
13   public LicensePlateTopicPublisher(int domainid, String topicname,
         String qosname) {
14     super(domainid, topicname, qosname);
15   }
16
17   @Override
18   void registerType(DomainParticipant participant) {
19     LicensePlateTypeSupport.register_type(participant,
           LicensePlateTypeSupport.get_type_name());
20   }
21
22   @Override
23   String getTypeName() {
24     return LicensePlateTypeSupport.get_type_name();
25   }
26
27   @Override
28   void writeDataToTopic(Object data, DataWriter datawriter) {
29     LicensePlate d = (LicensePlate)data;
30     LicensePlateDataWriter w = (LicensePlateDataWriter) datawriter;
31
32     w.write(d, InstanceHandle_t.HANDLE_NIL);
33   }
34 }
```

Appendix E

Subscriber source code

Listing E.1: An abstract Java class for the subscribers

```
1  package icatcher.wrapper.subscriber;
2
3  import icatcher.profiles.Util;
4
5  import com.rti.dds.domain.DomainParticipant;
6  import com.rti.dds.domain.DomainParticipantFactory;
7  import com.rti.dds.infrastructure.StatusKind;
8  import com.rti.dds.subscription.DataReader;
9  import com.rti.dds.subscription.DataReaderListener;
10 import com.rti.dds.subscription.LivelinessChangedStatus;
11 import com.rti.dds.subscription.RequestedDeadlineMissedStatus;
12 import com.rti.dds.subscription.RequestedIncompatibleQosStatus;
13 import com.rti.dds.subscription.SampleLostStatus;
14 import com.rti.dds.subscription.SampleRejectedStatus;
15 import com.rti.dds.subscription.Subscriber;
16 import com.rti.dds.subscription.SubscriberListener;
17 import com.rti.dds.subscription.SubscriptionMatchedStatus;
18 import com.rti.dds.topic.InconsistentTopicStatus;
19 import com.rti.dds.topic.Topic;
20 import com.rti.dds.topic.TopicListener;
21
22 abstract class TopicSubscriberWrapper implements Runnable,
       DataReaderListener, SubscriberListener, TopicListener{
23   private DomainParticipant participant;
24   private DataReader datareader;
25   private int domainid;
26   private String topicname;
27   private String qosname;
28
29   abstract void registerType(DomainParticipant participant);
30   abstract String getTypeName();
31   abstract public void on_data_available(DataReader reader);
32
33   public TopicSubscriberWrapper(int domainid, String topicname, String
         qosname) {
34     this.domainid = domainid;
```

```
35      this.topicname = topicname;
36      this.qosname = qosname;
37    }
38
39    /**
40     * Run this thread and wait for data to arrive
41     */
42    @Override
43    public void run() {
44      // load QoS profiles
45      Util.setProfile();
46
47      // get the participant
48      participant = getParticipant();
49      if (participant == null) System.err.println("! Unable to create DDS
            domain participant");
50
51      registerType(participant);
52
53      Topic topic = getTopic();
54      if (topic == null) System.err.println("! Unable to create topic " +
            topicname);
55
56      Subscriber subscriber = getSubscriber();
57      if (subscriber == null) {
58        System.err.println("! Unable to create DDS Subscriber");
59        throw new RuntimeException("Subscriber creation failed");
60      }
61
62      System.out.println(this.getClass().getName()+": Creating the data
            reader...");
63      datareader = getDataReader(subscriber, topic);
64      if (datareader == null) System.err.println("! Unable to create DDS
            Data Reader");
65    }
66
67    private DomainParticipant getParticipant(){
68      if(participant==null)
69        participant = DomainParticipantFactory.get_instance().
              create_participant(domainid, DomainParticipantFactory.
              PARTICIPANT_QOS_DEFAULT, null, StatusKind.STATUS_MASK_NONE);
70      return participant;
71    }
72
73    private Topic getTopic(){
74      return participant.create_topic(topicname, getTypeName(),
            DomainParticipant.TOPIC_QOS_DEFAULT, this, StatusKind.
            STATUS_MASK_NONE);
75    }
76
77    private Subscriber getSubscriber() {
78      return participant.create_subscriber(DomainParticipant.
            SUBSCRIBER_QOS_DEFAULT, this, StatusKind.STATUS_MASK_NONE);
79    }
```

```
80
81    private DataReader getDataReader(Subscriber subscriber, Topic topic) {
82      return (DataReader) subscriber.create_datareader_with_profile(topic,
            "iCatcherLib", qosname, this, StatusKind.STATUS_MASK_ALL);
83    }
84
85    private void outputStatus(String status){
86      System.out.println(status);
87    }
88
89    public void on_sample_lost(DataReader reader, SampleLostStatus status)
          {
90      outputStatus("sample lost");
91    }
92    public void on_requested_deadline_missed(DataReader reader,
          RequestedDeadlineMissedStatus status) {
93      outputStatus("requested deadline missed");
94    }
95    public void on_requested_incompatible_qos(DataReader reader,
          RequestedIncompatibleQosStatus status) {
96      outputStatus("requested incompatible qos");
97    }
98    public void on_sample_rejected(DataReader reader, SampleRejectedStatus
          status) {
99      outputStatus("sample rejected");
100   }
101   public void on_liveliness_changed(DataReader reader,
          LivelinessChangedStatus status) {
102     outputStatus("liveliness changed");
103   }
104   public void on_subscription_matched(DataReader reader,
          SubscriptionMatchedStatus status) {
105     outputStatus("subscription matched");
106   }
107   public void on_data_on_readers(Subscriber arg0) {
108     outputStatus("data on readers");
109   }
110   public void on_inconsistent_topic(Topic arg0, InconsistentTopicStatus
          arg1) {
111     outputStatus("inconsistent topic");
112   }
113 }
```

Listing E.2: An implementation of the abstract subscriber Java class for the LicensePlate data type.

```
1  package icatcher.wrapper.subscriber;
2
3  import icatcher.idl.licenseplate.LicensePlate;
4  import icatcher.idl.licenseplate.LicensePlateDataReader;
5  import icatcher.idl.licenseplate.LicensePlateSeq;
6  import icatcher.idl.licenseplate.LicensePlateTypeSupport;
7
8  import com.rti.dds.domain.DomainParticipant;
```

```
9  import com.rti.dds.infrastructure.RETCODE_NO_DATA;
10 import com.rti.dds.infrastructure.ResourceLimitsQosPolicy;
11 import com.rti.dds.subscription.DataReader;
12 import com.rti.dds.subscription.InstanceStateKind;
13 import com.rti.dds.subscription.SampleInfo;
14 import com.rti.dds.subscription.SampleInfoSeq;
15 import com.rti.dds.subscription.SampleStateKind;
16 import com.rti.dds.subscription.ViewStateKind;
17
18 public class LicensePlateTopicSubscriber extends TopicSubscriberWrapper {
19
20   private LicensePlateSeq _dataSeq = new LicensePlateSeq();
21   private SampleInfoSeq _infoSeq = new SampleInfoSeq();
22   private SubscriberDataListener<LicensePlate> data;
23
24   public LicensePlateTopicSubscriber(int domainid, String topicname,
         String qosname, SubscriberDataListener<LicensePlate> data) {
25     super(domainid, topicname, qosname);
26     this.data = data;
27   }
28
29   @Override
30   String getTypeName() {
31     return LicensePlateTypeSupport.get_type_name();
32   }
33
34   @Override
35   public void on_data_available(DataReader reader) {
36     LicensePlateDataReader datareader = (LicensePlateDataReader) reader;
37     try {
38       datareader.read(_dataSeq, _infoSeq, ResourceLimitsQosPolicy.
             LENGTH_UNLIMITED,
39           SampleStateKind.NOT_READ_SAMPLE_STATE, ViewStateKind.
                 ANY_VIEW_STATE,
40           InstanceStateKind.ANY_INSTANCE_STATE);
41
42       for (int i = 0; i < _dataSeq.size(); ++i) {
43         SampleInfo info = (SampleInfo) _infoSeq.get(i);
44
45         if (info.valid_data) {
46           LicensePlate obj = ((LicensePlate) _dataSeq.get(i));
47           data.dataReceived(obj);
48         }
49       }
50     } catch (RETCODE_NO_DATA noData) {
51       // No data to process
52     } finally {
53       datareader.return_loan(_dataSeq, _infoSeq);
54     }
55   }
56
57   @Override
58   void registerType(DomainParticipant participant) {
```

```
59        LicensePlateTypeSupport.register_type(participant,
              LicensePlateTypeSupport.get_type_name());
60    }
61  }
```

Listing E.3: An interface for a listener to receive data.

```
1  package icatcher.wrapper.subscriber;
2
3  public interface SubscriberDataListener<V> {
4    void dataReceived(V data);
5  }
```