MASTER'S THESIS

NETWORK MANAGEMENT IN A DISTRIBUTED SYSTEM

Maurits David de Jong

HUMAN MEDIA INTERACTION
FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE

EXAMINATION COMMITTEE
dr. Job Zwiers
ir. Pierre G. Jansen
ing. Niels A.G.W. Kortstee
ing. Robert C.G. Poth

UNIVERSITY OF TWENTE.

29 AUGUST 2011

Network Management in a Distributed System

v1.3.9

Maurits D. de Jong

EEMCS, Human Media Interaction

University of Twente

A thesis submitted for the degree of

Master of Science Human Media Interaction

29 August 2011

We approve the master's thesis of Maurits D. de Jong.

Date of Signature

dr. Job Zwiers
Associate Professor of Human Media Interaction (HMI)
Thesis Supervisor
Chair of Committee

ir. Pierre G. Jansen *retired*
Associate Professor of Computer Architecture for Embedded Systems (CAES)
Thesis Co-Supervisor

ing. Niels A.G.W. Kortstee
Thesis Co-Supervisor

ing. Robert C.G. Poth
Thesis Co-Supervisor

Acknowledgements

During my work on this thesis I have experienced a lot of support from many directions, for which I'm very grateful. The coaching and critique I have received from my supervisors, both at the University of Twente and at PrismTech Netherlands, have helped me achieve more than I would have achieved without them. The team in general at PrismTech Netherlands was always helpful and also supported me with both technical and non-technical issues.

Besides the support from my professional environment, I would like to acknowledge two persons especially. First and most of all: my loving wife, *Eline*, who has supported me in every way possible. She has been an amazing help, listener and motivator. Her patience with me and the graduation process has been incredible. Secondly my father, *Marcel*, who has helped me forward with his brilliant technical insights; the resulting sparring we could do on technical subjects and ideas, motivational support and pulling me out of the mud when stuck in textual formulations, have greatly helped me with completing this thesis.

Abstract

When managing large scale *Data Distribution Service* (DDS) systems, the information requests of system-managing experts vary widely. A monitoring system is indispensable and must be either very extensive — which is unfeasible — or it must be simply extendible. The goal of such a system is to facilitate the experts' need for information by providing a straightforward and flexible solution, without introducing high hurdles to be cleared in order to use it. Basic monitoring and diagnostic features must thus be available, providing a friendly basis for evolution of the system and guaranteeing immediate usability of the solution.

A new concept is introduced, dubbed *Delegated Monitoring*, describing a framework that suits the DDS environment naturally. It solves many of the issues encountered in traditional network management solutions by using the facilities of DDS itself. The primarily in-band solution can even support (fully) autonomous monitoring and/or (fully) autonomous management solutions. The framework doesn't require an on-line central entity to define what is monitored and where it is monitored.

By distributing the management routines as well as the commands controlling them through DDS, the delegated monitoring concept provides a (fully) distributed cooperative framework that enables the required high flexibility.

The monitoring framework requires a strict definition of the domain semantics with regard to low level monitoring primitives, allowing for a uniform interpretation of these primitives. Understanding them well is particularly needed in diverse highly integrated DDS systems and helps to make such systems diagnosable.

Preface

As a newcomer in the world of DDS, I soon ran into the limitations of the currently available diagnostic tools. It appeared that the need for improvement was widely felt. Not surprisingly, time to research and develop a more useful tool or generic solution was really scarce. It was from the initial discussions on the topic that the subject for this thesis emerged, where the setting of a graduation project could provide for the time to properly research the current solutions and the possibility to come up with a new design or concept that would allow for an integral improvement of the diagnosability of the DDS system.

Working on it proved to be not only an insight increasing activity, but also a potentially essential contribution to the development of better diagnostic tools for DDS systems. In particular a fully distributed implementation of the DDS standard, such as OpenSplice DDS™, can benefit greatly from a monitoring and management approach that is also capable of being applied in a fully distributed manner.

I enjoyed exploring and working creatively in this highly specialised domain, where often seemingly incompatible requirements meet (and have to be met). Microseconds matter, yet still discussions take place at a level of abstraction that is not expected when worrying about microseconds. I have started to appreciate the underlying concepts and the proven applicability of the paradigm and therefore fully hope that my contribution may one day find it's way into the domain.

MAURITS DE JONG
Enschede, The Netherlands
August 2011

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements or standards defined entities from ordinary English. These conventions are not used in tables or section headings where no distinction is necessary.

Adobe Minion Pro — Standard body text.

Adobe Minion Pro Italic — Formulae and/or calculations.

`Thesis Mix Mono` — programming statements or standards defined entities.

Pluralisation Normal plural forms of entities (either programming or standards defined) are used for readability. The suffix will be formatted as standard body text. For example the plural of `Publisher` will become `Publisher`s, the possessive form `Publisher`'s and the plural possessive form `Publisher`s'. Just so, the plural of `Class` will be written as `Class`es. The name of an entity will however never be modified for proper pluralisation. In cases where that would otherwise be necessary, the suffix *-'s* will be used. The plural of `Policy` will thus be written as `Policy`'s instead of `Policies`.

Contents

List of Figures

List of Tables

Acronyms

MiB unit symbol for the *mebibyte* as defined by the *International Electrotechnical Commission* (IEC). The binary prefix *mebi* means 2^{20}, therefore 1 mebibyte is 1 048 576 byte. 52–54

RxO is the acronym for *requested–offered* — indicates a compatibility requirement for requested and offered *quality of service* (QOS) in the DDS specification. 47, B-1, *see* QOS & QoS-POLICY

API is the acronym for *application programming interface* — an interface implemented by a software program which enables it to interact with other software. 20, 55, 58, 71, A-1

AVM is the acronym for *application virtual machine* — an application inside a host *operating system* (OS) supporting a single process. 56, *see also* JVM

CIL is the acronym for *Common Intermediate Language* — the lowest-level human-readable programming language defined by the *Common Language Infrastructure* (CLI). 56, *see also* CLI

CLI is the acronym for *Common Language Infrastructure* — an open specification describing the executable code and runtime environment that form the core of the Microsoft .NET Framework and the free and open source implementations Mono and Portable.NET. xii, *see also* CIL

CMS is the acronym for *combat management system*. 33

CORBA is the acronym and short name for the *Object Management Group* (OMG's) *Common Object Request Broker Architecture* standard, which enables software components written in multiple computer languages and running on multiple computers to work together. xiii, A-1

CPU is the acronym for *central processing unit* — the portion of a computer system that carries out the instructions of a computer program. 25, 33, 35, 36, 39, 45, 48, 52–54, 58–60, 69

DBMS is the acronym for *database management system* — a set of programs that controls the creation, maintenance, and use of a database. 19, 22

DCPS is the acronym for *data centric publish subscribe* — a publish/subscribe paradigm that is defined around the requirement for availability of a specific piece of data, *i.e.*, the interface between publishers and subscribers is the data itself. It can also refer to the lower level interface specified in the DDS-specification. ix, xiii, 11–13, 15, 16, 20

DDS is the acronym for the OMG's *Data Distribution Service for Real-time Systems* standard, which defines a publish/subscribe middleware for distributed real-time systems, created by the OMG in response to the need to augment CORBA with a *data centric publish subscribe* (DCPS) specification. ii, iii, xi–xiii, 4–10, 12–14, 17–19, 21, 22, 25, 26, 28, 30–32, 34, 36, 37, 39, 40, 45, 48, 50, 51, 53–56, 61, 63–65, 67–72, 74, B-1

DLRL is the acronym for the *Data Local Reconstruction Layer* — an (optional) higher level interface defined in the DDS-specification for simple integration in the application layer. 13, 16, 19, 20

FDEB is the acronym for *force-directed edge bundles* — a self-organising edge-bundling method capable of bundling edges in any graph since it doesn't require hierarchical input (Holten & Van Wijk, 2009; Holten, 2009). x, 73, *see* HEB & GBEB

FSF is the acronym and short name of the *Free Software Foundation*. xiv

GBEB is the acronym for *geometry-based edge bundling* — an edge-bundling method that relies on the generation of a control mesh to guide the bundling (Cui *et al.*, 2008). 73, *see* HEB & FDEB

GDS is the acronym for *global data space* — the concept of a storage or memory that is accessible to multiple interested applications. 4, 22, 23

HEB is the acronym for a *hierarchical edge bundles* — an edge bundling technique developed by Holten (2006). 73, *see* FDEB & GBEB

IDL is the acronym for *Interface Description Language* — a specification language used to describe a software component's interface. 53, 55, A-3

IEC is the acronym and short name the *International Electrotechnical Commission* — a non-profit, non-governmental international standards organization that prepares and publishes international standards for all electrical, electronic and related technologies. xii

IETF is the acronym and short name for the *Internet Engineering Task Force* — a task force with the mission to make the internet work better by producing high quality, relevant technical documents that influence the way people design, use, and manage the internet. 30

IP is the acronym for the *Transmission Control Protocol* — a protocol providing the communication of data across packet-switched networks which is part of the internet layer of the Internet Protocol Suite. xv, *see also* TCP/IP

JIT is the acronym for *just-in-time compilation* — a method to improve the runtime performance of computer programs by compiling (parts of) a program during runtime. 57

JVM is the acronym for *Java™ Virtual Machine* — a set of computer software programs and data structures which use a virtual machine model for the execution of other computer programs and scripts. The model used by a JVM accepts a form of intermediate language commonly referred to as Java™ bytecode. xiv, xvii, *see* AVM

LGPL is the acronym and short name for the *GNU Lesser General Public License* — a free software license published by the *Free Software Foundation* (FSF). 19

MDE is the acronym for *model driven engineering* — a software development methodology which focuses on creating models, or abstractions, more close to some particular domain concepts rather than computing (or algorithmic) concepts. It is meant to increase productivity by maximising compatibility between systems, simplifying the process of design, and promoting communication between individuals and teams working on the system. 19

MIB is the acronym for *management information base* — a virtual database used for managing the entities in a communications network. 30

NMRG is the acronym and short name for the *Network Management Research Group* — a forum for researchers to explore new technologies for the management of the internet. 30

OMG is the acronym and name for the *Object Management Group* — a software consortium aimed at setting standards for distributed object-oriented systems, modelling (programs, systems and business processes) and model-based standards. xii, xiii, 4, 6, 12, 13, 18

OOP is the acronym for *object-oriented programming* — a programming paradigm using *objects* — data structures consisting of data fields and methods together with their interactions — to design applications and computer programs. 26, 27

OO is the acronym for *object-oriented*. 4, 20, 23, 27

OS is the acronym for *operating system* — software, consisting of programs and data, that runs on computers, manages computer hardware resources, and provides common services for execution of various application software. xii, 35, 54

P2P is the acronym for *peer-to-peer* — a computing or networking architecture that partitions tasks of workloads between equally privileged, equipotent participants. 2, 30, 36

QOS is the acronym for *quality of service* — a resource reservation control mechanism that provides the ability to provide different priority to different applications, users, or data flows, or to guarantee a certain level of performance to a data flow. xii, xviii, 4, 15, 24, 34, 36, *see also* QoS-POLICY

SNMP is the acronym for *Simple Network Management Protocol* — an internet-standard protocol for managing devices on *Internet Protocol* (IP) networks. 30, 34

SPLICE is the acronym for *Subscribe Paradigm for the Logical Interconnection of Concurrent Engines* — a system architecture developed with the goal to simplify the creation and maintainability of complex, distributed, real-time systems. 12, 17, 18

SPOF is the acronym for *single point of failure* — a part of a system that, if it fails, will stop the entire system from working. 1, 12, 23, 33, 35

TCP is the acronym for the *Transmission Control Protocol* — a protocol guaranteeing reliable, ordered delivery that is part of the transport layer of the Internet Protocol Suite. *see* TCP/IP

UDP is the acronym for the *User Datagram Protocol* — a protocol providing best-effort delivery which is part of the transport layer of the Internet Protocol Suite. *see* UDP/IP

Glossary

C is the name of the *C programming language* — an imperative (procedural) language and is one of the most widely used general-purpose computer programming languages. 20

C++ (pronounced: *see plus plus*) is the name of the *C++ programming language* — a multi-paradigm general-purpose computer programming language, which is among one of the most popular languages. 20, 27

C# (pronounced: *see sharp*) is the name of the *C# programming language* — a multi-paradigm programming language developed by Microsoft. 20

Ethernet is a family of frame-based computer networking technologies for local area networks, standardised as IEEE 802.3. 24

Java EE stands for *Java™ Platform, Enterprise Edition*, a standard platform for developing multitier enterprise applications with Java™. 1

Java™ is a programming language originally developed by Sun Microsystems[1]. Java applications are typically compiled to bytecode which can run on any *Java™ Virtual Machine* (JVM) regardless of computer architecture. xiv, xvii, 20, 33, 56, 57

Lingua Franca is a *bridge language* or *working language* — "a language systematically used to communicate between persons not sharing a mother tongue, in particular when it is a third language, distinct from both persons mother tongues" (Wikipedia, 2004a). 23

[1]Since January 27, 2010 fully acquired by Oracle Corporation

QoS-policy is a policy defining QOS — a general concept that is used to specify the behaviour of a service, often used for aspects like resource reservation control in communication systems. xi, 4, 5, 12–15, 21–23, 46, 47, 59–61, B-5, *see also* QOS

SOAP is a protocol specification for exchanging structured information using Extensible Markup Language (XML). Previously it was known as the acronym for *Simple Object Access Protocol*, but the acronym was dropped with version 1.2 of the specification. 19, 21

TCP/IP is a set of communication protocols. 24, *see* TCP & IP

UDP/IP is a set of communication protocols. 24, *see* UDP & IP

Chapter 1

Introduction

The domain of large-scale networked computer systems is largely dominated by client/server architectures like for example the World Wide Web or Java EE. Distributed parallel systems are becoming larger and more advanced and in many distributed application domains the publish/subscribe messaging paradigm has become popular. The paradigm — in contrast to traditional point-to-point models, *e.g.*, client/server — decouples end-points in time, synchronisation and space. This decoupling makes publish/subscribe systems potentially highly scalable (Eugster *et al.*, 2003, p. 117) and it also facilitates dynamic network topologies. Full advantage can be taken of the decoupling by implementing a fully distributed event dispatcher infrastructure, thereby eliminating a *single point of failure* (SPOF) which exists in centralised event dispatcher infrastructures. This can make a publish/subscribe middleware very robust. While many deployed publish/subscribe systems still use a centralised infrastructure, some commercial and academic efforts have led to highly-available publish/subscribe middlewares that use a truly decentralised architecture (Cugola *et al.*, 2003, p. 187).

The decentralised nature, increasing scale and overall complexity of such distributed systems makes analysis and comprehension of runtime behaviour increasingly complex. In many types of work people find working with distributed and/or parallel systems (*e.g.*, parallel debuggers or distributed algorithms) difficult (Kraemer & Stasko, 1993, p. 105). This is not lastly due to the amount of raw data that often needs to be analysed (Reed *et al.*, 1995). Specifically in a publish/subscribe infrastructure, which on a logical level is inherently connection-less, (system-) aspects like interactions (or lack thereof), dataflows and availability, performance and mappings of logical concepts onto physical entities are hard to analyse.

1.1 Network Management

Network management is concerned with system deployment, surveillance of important performance metrics and optimisation of system performance. It requires means to monitor important (performance) metrics and to control the behaviour of both the hardware and the software resources of the system. A network management framework should apart from that optimally support autonomous optimisation too.

The task of managing large scale systems is often concerned with vast amounts of data coming from highly complex infrastructure compositions. Since the supporting protocols, the infrastructure itself and sometimes even the topology are not completely understood, it is essential to support network management experts in gaining insight in the system.

Network management tasks that are well supported in traditional network systems are more difficult in more dynamic networks like in *peer-to-peer* (P2P) networks, ad-hoc networks or the logical network formed by the matched subscriptions of the publish/subscribe architecture, for example due to the fact that many tasks are very difficult to be performed by a centralised manager. In these dynamic systems the monitored network itself needs to implement part of the network management functionality in a distributed manner, enabling the network to perform distributed filtering and aggregation of monitoring data. This data can then be presented and visualised for human operators in a meaningful and accessible format. For example topological views allowing interactive exploration of non-volume related parameters like unusual traffic and patterns can be provided.

It is thus essential that a network management architecture for a publish/subscribe system supports these monitoring and management primitives. Such a solution can be built by making distributed entities cooperate at some level in order to achieve functionality outside entity boundaries. By utilising a cooperative architecture, the need for a centralised management entity is avoided.

1.2 Delegated Network Management

A new approach to enhance, or better enable, experts capability to monitor and diagnose publish/subscribe systems is proposed. Provisions to dynamically delegate monitoring tasks to all components of the system are designed. This distributed monitoring framework allows the management induced network-load to be effectively reduced while pertaining the good scalability characteristics of the underlying publish/subscribe system. Because the tasks are delegated, processing of monitoring data is done distributedly, also supporting the scalability of the network management framework.

Monitoring behaviour at distributed nodes or groups of nodes and even diagnostic tasks can be tuned to varying and/or evolving needs and desires of the network managing expert because of the dynamic delegation. This tunability is of vital importance as it provides a principal flexibility, that will almost certainly prove to be of essential value when expertise in this new way of network managing begins to grow. Although this new approach may prove to be a necessity without which further upscaling of highly reliable and dynamic network systems may be impossible — or at least extremely troublesome — the same approach may prove to be worthwhile also in systems of lesser extent. In fact, there is virtually no reason to live without the attractive characteristics of delegated monitoring even in the smallest of systems.

The proposed network management architecture implements a scalable, fully distributed and extensible monitoring solution, allowing for virtually unlimitedly complex distributed monitoring logic to be deployed, including distributed automatic optimisation algorithms.

1.3 Visualisation

When the monitoring and management primitives are combined with existing or newly developed network management and visualisation tools, a very powerful solution can be created. Analysis can be done on the conceptual publish/subscribe network, where for example the dataflows can be used to form an overlay network, even when the true underlying concept is fundamentally connection-less. Visualisation of monitoring data can be used to exploit the remarkable capabilities of the human visual system in identifying trends, patterns and peculiarities in datasets, allowing system exports to gain insight in the monitored system.

Allowing a user to interactively select matched subscriptions or lower level network channels will allow the user to gain specific insights regarding the mapping, connection attributes or connectivity in general.

1.4 Data Distribution Service

The *Object Management Group* (OMG) — a software consortium aimed at setting standards for distributed *object-oriented* (OO) systems, modelling (programs, systems and business processes) and model-based standards — has defined a publish/subscribe standard called *Data Distribution Service for Real-time Systems*[1], of which the first version[2] was published on June 1, 2003. It has been created in response to the need for a standardised data-centric programming model for distributed real-time systems. Many of the concepts of the DDS specification originated from a project aimed at simplifying the intricate task of designing large-scale systems with stringent requirements regarding robustness, availability, temporal behaviour, performance and maintainability, as described by Boasson (1993).

The DDS standard defines a data-centric publish/subscribe networking middleware that simplifies complex network programming. It defines the concept of a logical *global data space* (GDS) containing information expressed by data-units called `Topics` that are published /consumed by distributed `Publishers` /`Subscribers`. The DDS standard specifies a large range of *quality of service* (QOS) policies that drive the logical behaviour of data-distribution and availability.

The level of abstraction provided to applications using a DDS middleware, often makes it difficult to identify faults when they occur. The specific focus of the standard on performance, the typical stringent requirements on deployed systems and the complex interplay of the available QoS-policies defined in it, can make runtime analysis of such systems a daunting task. The rich feature set and global concept underlying the DDS specification enable and stimulate the efficient development of large scale distributed systems as described by Boasson (2002, chap. 5). However, the same extensive set

[1]*Data Distribution Service* (DDS)

[2]The current version of the specification is DDS v1.2 described in OMG document *formal/2007-01-01*, which can be obtained from `http://www.omg.org/spec/DDS/1.2/PDF/`.

of features and concept are on the other hand a complicating factor in understanding and addressing faulty runtime behaviour.

1.5 OpenSplice DDS™

An implementation of the DDS specification — implemented using a truly scalable and decentralised architecture — is OpenSplice DDS™. The core development of OpenSplice DDS™ is done at PrismTech Netherlands[1]. OpenSplice DDS™ is a very high-performance, real-time, data-centric publish/subscribe middleware platform for distributed mission-critical systems. It extends on the standard by allowing for a dynamic and programmatic mapping of logical DDS entities and their related QoS-policies (w.r.t. information urgency and importance, information reliability and persistence) onto the physical (run-time) infrastructure existing of interconnected (sub)systems, hierarchical networks, enterprise and embedded computing nodes, shared-memory within nodes, multi-threaded application processes and pluggable services. PrismTech has an open source edition of their implementation *OpenSplice DDS™ Community Edition*[2] available since April 2009.

Many large systems with extreme requirements regarding availability, robustness and performance utilise OpenSplice DDS™ because of its distinctive features with regard to data-centred control, deployment, services, modelling and other facilities it offers to the applications, developers and system designers.

1.6 Optimisation and Maintenance

The deployment of systems adumbrated above has in practice revealed that optimisation and curing and/or preventing system malfunction are daunting tasks that require system experts with overall knowledge of the system. The vast amount of — mostly textual and inherently sequential — data that can aid in resolving issues can be difficult to assimilate. The relation of the metrics to the issue at hand is often not well defined either, complicating the task even more. Typical issues that have to be dealt with involve inappropriate resource allocation, hardware failures and performance deficiencies.

[1]Hengelo (Ov.), The Netherlands

[2]OpenSplice DDS™ Community Edition can be downloaded from `http://opensplice.org`

Conceptually, tuning of the system can be seen as a multidimensional optimisation problem with complex optimisation criteria. Mankind searches for an optimal state in many areas — for as long as we exist. Optimisation science exists from ancient times and may well be one of the oldest sciences in the world (Neumaier, 2004, p. 3) and surely influences our daily lives. Many optimisation algorithms have been developed over the years (*e.g.*, Horst *et al.*, 2000; Weise, 2009). The web page created by Neumaier (1995) is also a good compilation of relevant matter on the subject.

Currently no automatic optimisation is done by OpenSplice DDS™, so it is interesting to determine to what extent optimisation algorithms (either on-line or off-line) can be applied to optimisation of deployed/running OpenSplice DDS™ systems. It is however anticipated that — even with the currently best possible automatic optimisation — human intervention will be necessary. Not least because large systems sometimes already require tuning in order to fulfil preconditions that must be satisfied in order to get OpenSplice DDS™ started. During this deployment/start-up phase, the system cannot perform on-line optimisations and configuration fixes and optimisations thus have to be carried out by system experts.

1.7 Organisation

The project was performed at PrismTech Netherlands and was supervised by *dr.* Job Zwiers (associate professor) and *ir.* Pierre G. Jansen (associate professor) on behalf of the University of Twente and *ing.* Robert C.G. Poth (chief architect) and *ing.* Niels A.G.W. Kortstee (technical-/team lead) on behalf of PrismTech Netherlands. Robert Poth and Niels Kortstee both have years of experience in developing mission-critical real-time systems at THALES Netherlands[1].

The goal of the graduation project is to analyse the issues encountered with management of OpenSplice DDS™ systems and design a solution that aids in- or resolves the encountered issues. Because the solution to this issue touches a lot of different domains of key research, an overview of the domains and their relevance will be given in the respective chapters in order to aid in answering the main questions of the thesis.

[1]Before *Holland Signaalapparaten B.V.* (or *Signaal* for short), one of the founding parties of the OMG-DDS specification

1.8 Research Questions

This section will introduce the main research questions that will be looked into in this thesis. In the following chapters the context of the questions will be highlighted and answers to the questions will be given. The main research question that is considered in this thesis is:

Research question I

How can human experts be optimally supported with their management tasks in DDS-based systems?

The main question to be answered within this thesis deals with the problem of how expert users of DDS middleware can be assisted in their system-management tasks by making behaviour of the system diagnosable. In this thesis the main focus will be on assisting the users in their tasks. It is however also possible that — at least a subset of — the problems encountered can be autonomously solved by the system by applying automatic optimisation algorithms. This also leads to the definition of research question II:

Research question II

Which network management and visualisation of management data is needed in DDS-based systems?

Visualisation is not a goal in itself, so the relevance of visualisation for network management related tasks will have to be determined. By finding the types of tasks and areas of network management where the users typically need to be assisted will allow the definition of suitable assisting methods. So probably, subsets of tasks will be best assisted by either (semi-)automatic optimisation, visualisation or even rendered obsolete by an autonomous capability of the system. In anticipation that there will be use-cases for which visualisation is useful despite other options like autonomous optimisation, the following questions are defined.

Research question III

What functionality is needed for management tasks in DDS systems?

Answering this question will give insight in the problems that need to be tackled in order to provide the right data in the right format to the users. Literature research on similar problems like network management in traditional network systems and data-visualisation will also provide valuable information on typical problems/solutions in this area. The insight gained by the answer on research question II will act as input to defining which functionality will have to be available in order to satisfy the visualisation and information needs for network management in DDS-based systems. This is expected to range from practical functionality like data gathering to abstract functionality like data presentation.

> **Sub research question III.a**
> *In what areas is the OpenSplice DDS™ system currently lacking?*

The current state of OpenSplice DDS™ with regard to the functionality as found in the answer to research question III will drive which parts of the implementation require further research. Shortcomings of OpenSplice DDS™ can be either in availability or design of specific functionality.

> **Sub research question III.b**
> *How can data be collected in a scalable manner, suiting the environments in which DDS is typically deployed?*

The typical nature of the systems in which DDS is deployed will render an even apparently trivial task like data-gathering very difficult. This issue will need to be solved in a way that suits the deployment environment, not yielding too many compromises on the network management capabilities.

> **Sub research question III.c**
> *Which data and/or metrics are needed for management of DDS systems?*

If it is possible to identify a type or set of data and metrics that will be needed for network management, defining them will help in finding existing applicable visualisations or creating new or adapted ones, tailored to the DDS network management needs.

> **Research question IV**
> *Is visualisation in a network management solution for DDS a luxury or a necessity?*

Given the availability of a considerable amount of network management related data, one may question whether it makes a difference to represent these data graphically. This question may be especially valid in the specific professional area of network managing experts. Some of these highly skilled persons have indeed developed a surprising and laudable ability in analysing raw or simply tabulated data. Nonetheless, if the raw data are properly transformed to graphics, the analysis may even be performed faster. Although thorough in-depth analysis of raw data related to a graphically localised issue may still be necessary, network management experts may still benefit greatly from graphical representations of the data.

Research question V

How can a network management implementation in OpenSplice DDS™ be designed?

A management solution for OpenSplice DDS™ will touch a lot of layers of both the product as well as abstractions of the product. In order to facilitate efficient management, a design will have to be created that satisfies all criteria found in this thesis.

1.9 Structure of this Thesis

In this chapter the context of the work performed for this thesis has been described. In chapter 2 (Data Distribution Service) the global concepts underlying the DDS specification and the specific implementation thereof in the form of OpenSplice DDS™ will be described. In chapter 3 (Network Management) the focus will be on traditional network management systems and how they relate to DDS. Furthermore the areas relevant for a network management solution in DDS will be discussed. In chapter 4 (Visualisation) the necessity of visualisation for network management and the general goal and concepts of visualisation are discussed. In chapter 5 (Design) a design will be proposed that suits the DDS environment and fulfils the requirements for a network management solution in DDS. Finally in chapter 6 (Conclusions) the findings to the research questions and recommendations for further research are given.

Chapter 2

Data Distribution Service

In chapter 1 (Introduction) a short introduction to publish/subscribe, DDS and OpenSplice DDS™ has been given. In this chapter, those topics will be discussed more elaborately and information will be provided about the publish/subscribe paradigm in section 2.1 below, the DDS standard in section 2.2 (OMG's DDS-Specification) and of OpenSplice DDS™ specifically in section 2.4 (OpenSplice DDS™). In section 2.3 (SPLICE) the heritage of both DDS and OpenSplice DDS™ will be introduced.

Goal of this chapter is to provide background information on the context in which the thesis is performed. This chapter will also answer research question II by explaining the difficulties encountered in deploying and managing systems based on the DDS specification.

2.1 Publish/Subscribe

Publish/subscribe[1] is a messaging paradigm related to the *message queue* paradigm. In the publish/subscribe concept, senders of messages (*publishers*) are not connected to the receivers (*subscribers*) of the messages. Messages are distinguished by classes to which they are published. There is no knowledge in the publisher as to what — if any — subscribers there are. Analogous, subscribers receive messages from classes they expressed interest in, without knowing what — if any — publishers there may be. The concept of decoupled senders and receivers is beneficial for the scalability and topology dynamics

[1]Also abbreviated *pub/sub* or PS

of systems implementing it.

Already in 1986 a patent was filed in The Netherlands for a *data centric publish subscribe* (DCPS)[1], shortly followed by an international application in 1987[2]. Also in 1987 at the *Symposium on Operating Systems Principles* conference a publish/subscribe subsystem of the *ISIS$_2$ Toolkit* called *News* (Birman & Joseph, 1987, p. 130) was published (Wikipedia, 2004b).

2.1.1 Filtering

Subscribers have interest in only a part of all published messages in typical systems and thus need means to filter the messages. Two forms for this *filtering* are often applied: *topic-based* and *content-based*. The first is characterised by the fact that the publisher is responsible for defining the classes and the classification of the messages, whereas in the latter case this is done by the subscriber.

Topic-based — Messages are published as *topics* to which subscribers can subscribe. All subscribers subscribed to a topic will receive the same messages. The publisher defines the classes/topics to which the messages belong.

Content-based — A subscription is taken on specific attributes or content of the messages, which will only be delivered if they match the constraints set by the subscriber. In this filtering approach, the responsibility for classifying messages lies with the subscriber.

These filtering approaches are not mutually exclusive and can thus be combined to form a *hybrid* system supporting both.

2.1.2 Advantages/disadvantages

As already noted earlier, the most notable advantages of the publish/subscribe paradigm are its *loosely-coupledness* and *scalability*. Because the publishers and subscribers are decoupled, they can be location agnostic and continue to operate if one of both ceases to exist. In a traditional client/server system, a client needs to know how to reach the

[1]Dec. 16, 1986, NL8603193

[2]Dec. 16, 1987, U.S. Pat. Appl. 133 679, continued in U.S. Pat. Appl. 600 275 on Oct. 17, 1990. Patent issued on Apr. 5, 1994, U.S. Pat. 5 301 339 (Boasson, 1994)

server *and* the server needs to be available at the time of the request of the client. This server is thus a SPOF in such a set-up. In a publish/subscribe system, the process implementing the functionality of the server (potentially consisting of both a publisher and a subscriber) can easily be relocated (for example to a more powerful machine) without affecting the whole system.

Because no end-to-end connections exist on the conceptual level, implementations can benefit by being able to scale easily. Especially when a lot of one-to-many relations exist. Although not all systems implementing the publish/subscribe paradigm actually do scale well, this is not an inherent limitation of the paradigm itself.

The most notable disadvantages of publish/subscribe systems often are attributed to one of the main advantages: its loosely-coupledness. The lack of end-to-end notion is considered to be a shortcoming by some, explained by the issue that arises when a publisher is interested in knowing whether anyone is actually doing something with the published information (or whether delivery succeeded)[1]. Such information is often not available to the publish/subscribe system.

Another case for which the lack of end-to-end notion is restrictive is sender/receiver verification or authorisation of content-access. While obviously highly implementation dependant, this traditionally requires the notion of the receiver on the sending side and vice versa.

2.2 OMG's DDS-Specification

As introduced in section 1.4 (Data Distribution Service), the OMG has defined the DDS standard which defines a DCPS architecture with a specific focus on performance. In data-centric communication, specific parameters can be specified for the data being distributed in the form of QoS-policies, allowing applications to be developed according to requirements regarding those policies. This leads to a system where the only interface between applications is the data itself together with the specified QoS-policies.

Being primarily based on two major proprietary DCPS implementations — *NDDS* from *Real-Time Innovations, Inc.* and *SPLICE* from *THALES* — the standard has been cre-

[1]One could argue that in a proper application of the publish/subscribe paradigm, the interest of the publisher should be expressed as a subscription to that information, which in turn should be published by the subscriber in cases where necessary

ated in a joint effort of these parties with the OMG. Apart from both mentioned vendors, *Objective Interface Systems, Inc.*, *The Mitre Corporation* and the *University of Toronto* also submitted and/or supported parts of the specification (OMG, 2007, pp. 3–4). Two levels of interfaces are defined in the specification: a lower DCPS level, which is targeted at efficiency and an (optional) higher *Data Local Reconstruction Layer* (DLRL) level, which is targeted at easier integration at the application level (OMG, 2007, p. 1).

The specification defines compliance profiles in which optional functionality is defined. For example the DLRL-layer is optional and is defined in a separate profile. The following profiles are distinguished (OMG, 2007, pp. 2–3):

Minimum profile — In this profile just the compulsory features of the DCPS-layer are contained.

Content-subscription profile — In this profile functionality is added that allows for content-based filtering. The default filtering mechanism in the minimum profile is topic-based.

Persistence profile — This profile adds a durability aspect to the middleware, which allows data to be stored on permanent or transient storage so that it can survive the life-cycle of a publisher/writer or even middleware outings.

Ownership profile — In this profile functionality to define an *owner* of data is added allowing the middleware to handle multiple publishers for the same data. This profile also adds the possibility to specify a history for instances.

Object model profile — This profile includes the DLRL-layer and adds some data-presentation functionality.

The DDS specification defines several QoS-policies by which application developers can influence how the service behaves, without the need to specify how this should be achieved. In the minimum profile, QoS-policies are available that affect predictability, overhead and resource utilisation. These policies follow directly from the main purpose for real-time applications of predictable data distribution with minimal overhead.

2.2.1 DCPS Model

This section is an excerpt of the basic concepts introduced in version 1.2 of the DDS specification, which should be consulted when a more detailed explanation is required.

2.2.1.1 Overview

In DDS information flows in the form of `Topics` from the sending side by means of a `Publisher` and `DataWriter` to a `Subscriber` and `DataReader` on the receiving side. All influenced by the QoS-policies of the respective entities. A `Topic` associates a name with a data-type and QoS-policy[1].

- The `Publisher` object is responsible for data distribution and may publish data of different types. Data for a specific type is written through a typed accessor of the `Publisher` called the `DataWriter`. When an application has used a `DataWriter` to communicate data-object values to the `Publisher`, it becomes the `Publishers` responsibility to perform the distribution. The association of a `Publisher` and a `DataWriter` defines a *publication*, which expresses the intent of an application to publish data described by the `DataWriter` in the context provided by the `Publisher`.
- The `Subscriber` object is responsible for receiving and providing data of different types to the receiving application. The received data can be accessed by the application by means of a `DataReader`, which defines a typed interface to a `Subscriber`. The association of a `Subscriber` and a `DataReader` defines a *subscription*, which expresses the intent of the application to subscribe to the data described by the `DataReader` in the context provided by the `Subscriber`.

Publications and subscriptions are matched based on `Topics`. Publications must be unambiguously identifiable for subscriptions to be able to refer to them, so the name of a `Topic` needs to be unique[2]. The QoS-policy related to the `Topic`, together with the QoS-policies of the `Publisher` and `DataWriter` control the behaviour on the publishing side, while the `Topic`-, `Subscriber`- and `DataReader`-QoS-policy control the behaviour on the subscribing side.

[1]The key of a `Topic` is also part of this association; keys are introduced in section 2.2.1.3 (Instances)
[2]Within a domain (see footnote 1 on page 16 for the definition of a domain)

2.2.1.2 Conceptual Model

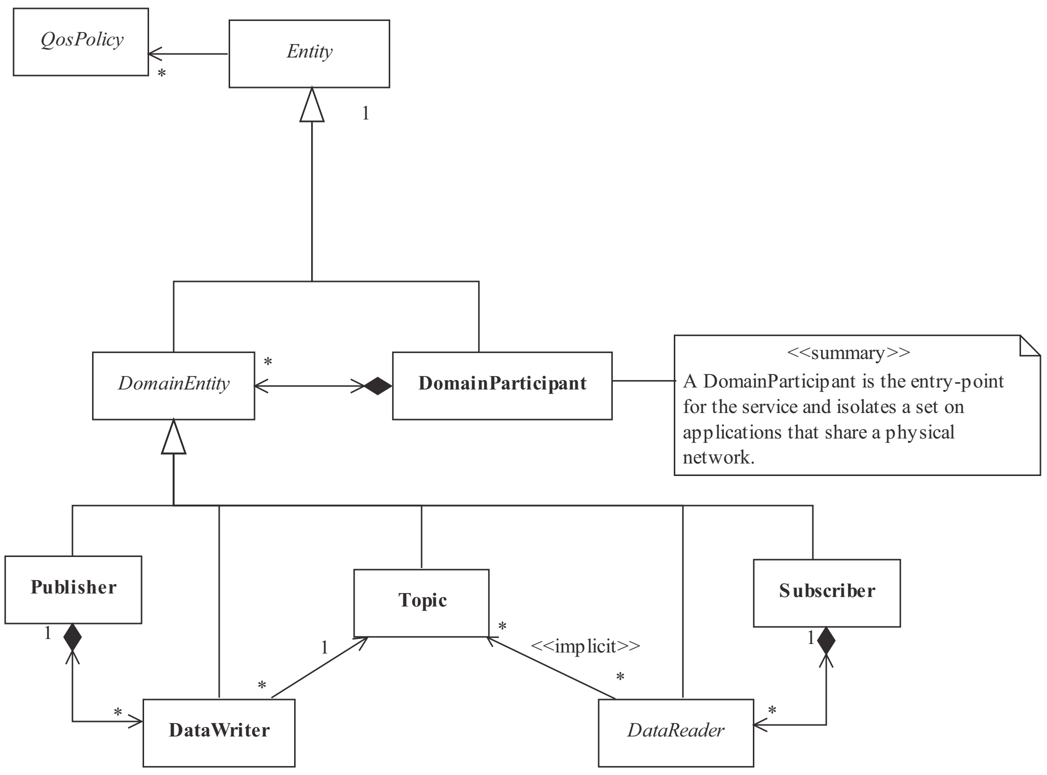


Figure 2.1: *DCPS conceptual model*
Source: OMG (2007, p. 10, simplified)

In Figure 2.1 above, the conceptual model of the DCPS communication entities is shown. Figure 2.2 shows that all specialisations of `Entity` follow the unified patterns of:

- Supporting QOS by means of `QosPolicy`'s — QoS-policies provide a generic mechanism for the application to control the behaviour of the middleware. Every `Entity` supports its own specialised kind of QoS-policy.
- Accepting a `Listener` — a `Listener` provides a generic mechanism for the middleware to asynchronously notify the application of events (notification-based communication).
- Accepting a `StatusCondition` — a `StatusCondition` (in conjunction with a

`WaitSet`) provides support for a wait-based communication style between the middleware and the application.

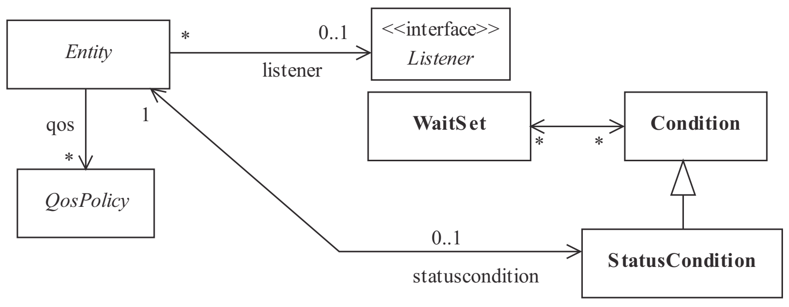


Figure 2.2: *Entity model*
Source: OMG (2007, p. 10, simplified)

The local membership of an application to a *domain*[1] is represented by a `DomainParticipant`, to which all DCPS entities are attached.

2.2.1.3 Instances

By definition, a `Topic` corresponds to a single data type, but several `Topics` may refer to the same data type. Information represented by data types are sent atomically[2]. These atomic data value sets are called *samples*.

A `Topic` identifies data of a single type, ranging from a single instance to a collection of instances. When a `Topic` identifies multiple instances, then the different instances must be distinguishable. This is achieved by means of specifying one or more of the data fields as *key*. The definition of the key, *i.e.*, the list of data fields whose values form the key, must be specified to the middleware. Different samples with the same key-value represent successive values for the same instance, samples with different key-values represent different instances. When a key definition is omitted, the data set associated with a `Topic` can only contain one instance.

[1]A *domain* is a distributed concept that links all applications that are able to communicate with each other

[2]On the DCPS level, the optional DLRL layer provides the means to break data-objects into separate elements, each sent atomically

2.3 Splice

One of the already existing implementations on which the DDS specification is based is SPLICE[1]. The development of this architecture already began in the 80s, leading to a patent application in 1986. It was driven by the difficulties encountered designing highly sophisticated control systems. The concept of *modularity* — where different pieces of functionality are separated into modules having some level of independence from other modules — was still lacking as a solution to this problem with regard to certain aspects. Particularly the non-functional requirements of such systems, like fault-tolerance, hybrid deployment environments with a multitude of different processing platforms and adaptability restrained the design freedom of the modular approach to the extent that it was hardly feasible to satisfy the requirements.

The SPLICE paradigm introduced an — also modular — approach that solved many of the issues inherent in the originally practised approach. In the SPLICE architecture, a system is (recursively, if wanted) composed of:

Applications — Independent, autonomous processes that are totally isolated from each other. They only interact with the rest of the system through *Agents*, with a well-defined interface.

Agents — Every application interacts with exactly one agent, which serves the application with storage and processing facilities needed for communication. All agents are identical and communicate through a message passing mechanism.

Network — The messages between agents are handled by the network that connects them.

All relevant data (outside of the application scope) is labelled in such a way that there is a one-to-one relation between a label and the interpretation of the data by an application that uses it. This label and the structure of the data being labelled are needed by application designers working on a particular subsystem. Communication decisions — like where to get specific data from or where to send it — are deferred until execution time. During execution, the agents determine, based on the labels of the data being produced or consumed, the communication needs. As a result, no application relies on the

[1]*Subscribe Paradigm for the Logical Interconnection of Concurrent Engines* (SPLICE)

presence or *knowledge* of another application. The only requirement is that the needed data is produced *somewhere* in the system.

2.3.1 Indices

A subscription to a particular label, a *datasort*, declares the interest in both the current as well as any future data-instance for that label. As a refinement to that principle, which allows for only one instance per label, the SPLICE paradigm allows specification of an index by means of key-fields. If an instance differs from already stored instances it will be stored separately, otherwise it will overwrite the previous value. This key-definition is a strictly local notion, and can thus differ between the producer and consumer, or multiple consumers of the same data.

Defining an index on the producing side may seem a bit strange at first, but it aids in flexibility in the order in which consuming applications become active. Typically, a consuming application will need all the currently valid instances for all its sorts. If the producer would not have an index defined, the consuming application would only get the (single) available data instance. To solve this, the producer may specify an index, so that a complete set of relevant instances is available instead. If an application subscribes to the sort, the producer's agent will first send out all stored instances and then send new instances as described earlier.

2.3.2 Refinements and Extensions

In order to tackle more specific issues typically encountered in control systems — the area in which SPLICE was developed — more extensions and refinements where made to the paradigm. For example support for redundancy and filtering. A more detailed explanation of the extensions and refinements is available by Boasson (1993, chap. III.B–C).

2.4 OpenSplice DDS™

The open source product of PrismTech's implementing the OMG-DDS specification is OpenSplice DDS™ and it has its roots in the SPLICE product developed at THALES according to the paradigm described in section 2.3 (Splice). The open source product is supplemented by commercial subscriptions, which extend the capabilities of the product even more.

Community Edition — The *OpenSplice DDS™ Community Edition* is available as open source and is therefore freely downloadable. This edition is licensed under the *GNU Lesser General Public License* (LGPL) and provides a great way of getting started with a full-fledged DDS implementation with support for all profiles except the object model profile.

Compact Edition — The *OpenSplice DDS™ Compact Edition* is available only by subscription. In addition to the features provided by the Community Edition, the Compact Edition includes *model driven engineering* (MDE) *PowerTools* to boost development productivity and the *OpenSplice DDS™ Tuner* tool to help inspect and optimise system performance.

Professional Edition — The *OpenSplice DDS™ Professional Edition* adds support for the optional object model profile, enhancing the Compact Edition with an Object Cache Object/Relational Mapping layer (*i.e.*, DLRL). It furthermore adds a *SOAP-Connector*, which allows for easy integration with SOAP-based environments.

Enterprise Edition — The *OpenSplice DDS™ Enterprise Edition* adds a connector to *database management system* (dbms's) and is the most complete and advanced edition available.

Apart from the basic descriptions above, the platform support generally gets better with more advanced editions.

2.4.1 Architecture

OpenSplice DDS™ utilises a shared-memory architecture where samples are physically present only once on any node. Internal administration still provides each `Subscriber` with its own private view on this data. This architecture enables a `Subscriber`'s data cache to be seen as an individual database, which can have content filters, queries, etc., by using the functionality provided in the DDS content-subscription profile. This shared-memory architecture results in low footprint, excellent scalability and optimal performance when compared to implementations where each `DataReader`/`DataWriter` is a communication end-point with its own storage (*i.e.*, historical data both at the `DataReader` and `DataWriter`) and where data itself has to be moved around in memory, even within one node. The same shared-memory is also used to interconnect all applications that reside within one computing node as well as for a configurable and extensible set of services, which provide pluggable functionality.

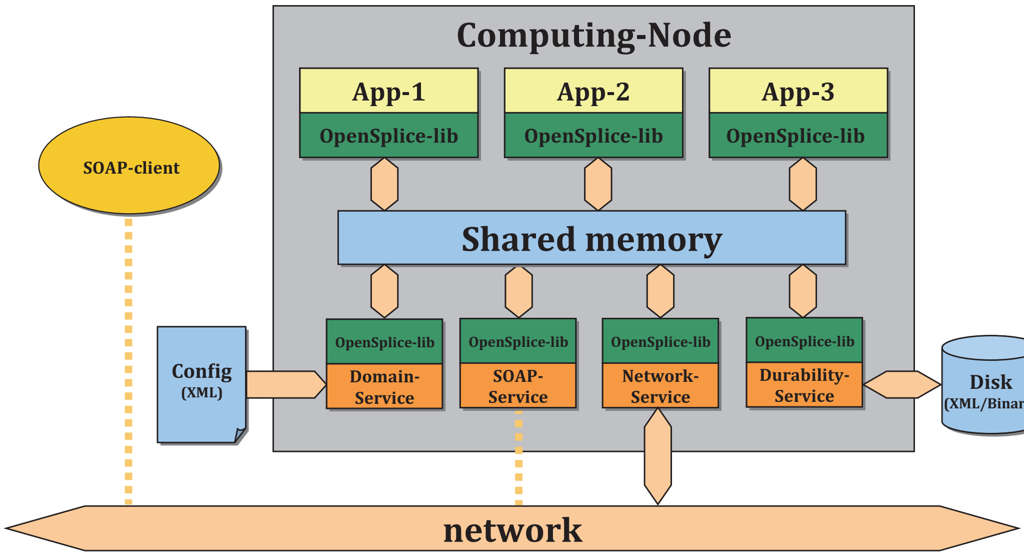


Figure 2.3: *OpenSplice DDS™ shared memory architecture*

In Figure 2.3 above a graphical representation of the shared memory architecture used in OpenSplice DDS™ is given. It shows four services — available in the OpenSplice DDS™ Professional Edition — and depicts the way services and applications interact in a standardised way with the shared memory. All applications and services access the shared-memory through a library which on the most basic level provides an OO database view on the shared-memory. There are several *application programming interface* (API) layers included in the library, each with a specific focus (*e.g.*, DCPS-API, DLRL-API and language-bindings for languages such as Java™, C, C++, C#, etc.).

2.4.2 Services

The OpenSplice DDS™ middleware comes with a set of pluggable services, which are highly configurable. The services can therefore be tailored to suit really specific and demanding use-cases. A description of the most common services is given below.

Domain-service — This service's responsibility lies with creating the shared memory and initialising the shared nodal administration for a specific DDS-domain on a node. It also configures the administration according to a configuration-file. The domain service also starts all pluggable services configured in the configuration-file, monitors the health and controls the life-cycle of the services. Furthermore it provides functionality to cleanly tear down a domain. Without this service, no application nor service can join a domain on that node.

SOAP-service — This service provides a remote interface to the monitor and control facilities of OpenSplice DDS™ by means of the SOAP-protocol. This enables for example the remote DDS-domain monitor and control functionality of the *OpenSplice DDS™ Tuner*[1].

Network-service — When communication endpoints are located on different nodes, the data produced locally must be communicated to the remote nodes joining the domain and vice versa. This service provides this bridge over a network interface. Multiple instances of this service can exist next to each other; each serving one or more physical network interfaces. The service is responsible for sending data to and receiving data from the network. It can be configured to distinguish multiple communication channels with different QoS-policies. These policies will then be used to schedule individual messages to specific channels.

Durability-service — OpenSplice DDS™ provides support for the optional persistence profile, which specifies the concept of so called durable data. Durable data produced by applications must stay available for late joining `DataReaders`. This means that `DataReaders` joining the system at a later stage will be able to receive (durable) data that has been produced before they joined. The durability of data can be either *transient* or *persistent* and is determined by the QoS-policy

[1]Which then acts as a SOAP-client as depicted in Figure 2.3 on the preceding page

of the `Topic`. If a specific `Topic` is marked to be transient, the corresponding data instances remain available in the system during the complete life-cycle of the system. If a specific `Topic` is marked to be persistent, the corresponding data instances even survive the shut-down of a system because they are written to a permanent storage (*e.g.*, a hard disk). The durability-service is responsible for the realisation of these durable properties of the data in the system.

DBMS-service [1] — This service provides a bridge between the real-time DDS and the enterprise DBMS domain. The service is capable of bridging data from DDS to any DBMS[2] and vice-versa. This provides many useful capabilities, such as for example the logging of DDS data in a DBMS, QoS-policy-enabled replication of a DBMS — even between different DBMS-implementations — and easy migration from existing DBMS-based applications to DDS-based applications.

2.4.3 Design

There are some fundamental principles underlying the architecture of OpenSplice DDS™, supporting its capabilities with regard to availability, real-time behaviour and scalability. In this section, some of those principles will be explained.

2.4.3.1 Shared Memory

OpenSplice DDS™ uses shared memory to implement the node-local concept of a GDS. This implementation suits the DDS concept very well, since it is the direct equivalent of the GDS-concept. The contents of the memory need only be available on a node once, no matter how many `Subscribers` there are for a specific `Topic`. This is one of the enablers for the very good scaling capabilities of OpenSplice DDS™, where use-cases are known that have more than 150 applications, 2 000 `DataWriters` and 6 000 `DataReaders` on one node. If all this applications and `DataReaders` would have their own copy of the data, these numbers would soon cause both resource and performance issues.

The shared memory is accessed through the OpenSplice DDS™-lib (as seen in Figure 2.3 on page 20). This standard way of interacting with the GDS is an implementation

[1]Not depicted in Figure 2.3 on page 20
[2]SQL'99 and ODBC-enabled

of a *Lingua Franca*; the communication language is defined, which ensures that the data *is* the interface. Although functional and OO decompositions have proven to be powerful methodologies, they often lead to tightly coupled systems, which complicates designing and maintaining large-scale interoperable distributed systems and systems of systems. Probably the greatest challenge is induced by the inherent fragility of the interfaces, which tend to change often. The GDS together with the Lingua Franca are tied on the information model, which is much more stable and extensible than the functional interfaces in an evolving system.

2.4.3.2 Hot-Swap Support

High availability distributed systems cannot afford to fail in delivering their services in case of a hardware or software failure. Failures therefore have to be properly masked and gracefully tolerated by the system. Performance implications induced by providing high availability should be minimised. OpenSplice DDS™ supports service replication by means of ownership QoS-policies, in which ownership of data and strengths of `DataWriters` can be specified. This enables a system-wide coordination of replicated writers without greatly impacting performance or increasing complexity at the application-level.

2.4.3.3 Time Decoupling

Loosely coupled distributed systems have to deal with *late joiners* and make sure that data is available to them without affecting performance. Time decoupling can be provided in different degrees, ranging from the availability of the 'latest' data, up to the availability of data after a full system restart. Due to high availability requirements, this cannot be implemented by a centralised server, which would introduce a SPOF. The `DURABILITY` QoS-policy controls the availability of data for late joiners and thus prescribes how published data needs to be maintained by the middleware. It defines four variants:

Volatile — Data does not need to be maintained for late-joining `DataReaders` (default behaviour).

Transient Local — Data availability for late joiners is tied do `DataWriter` availability and the middleware thus needs maintain the data for as long as the `DataWriter` is active.

Transient — Data outlives the `DataWriter`. The data needs to be maintained for as long as the middleware is active on at least one of the nodes.

Persistent — Data outlives complete system restarts. This implicates that the data must be stored on permanent storage in order to be able to make it available again after the middleware is restarted.

In OpenSplice DDS™, the durability provisions are implemented in a fully distributed manner, which facilitates the high availability of the system, even in case of partial failures.

2.4.3.4 Singleton Communication

In timing- and mission-critical systems resources need to be kept under strict control. This is true for all resources that are critical to the communication in a system. In distributed applications, the connecting network is a critical resource too. In OpenSplice DDS™, the network resources are therefore governed in order to ensure predictable, scalable and dependable behaviour. In situations where temporary overload conditions can occur, distributed applications often start to misbehave, causing the system to provide degraded QOS or even miss specific QOS agreements. These overload conditions can be controlled at an application level, but at the expense of dramatically increased application complexity. Therefore OpenSplice DDS™ relies on a single service per node for dealing with the network traffic management. This solution ensures that global properties can be enforced and the network resource is properly protected. This also enables pre-emptive scheduling on the network-level ensuring extremely low latencies for high priority data. OpenSplice DDS™ supports association of a priority with data and uses it as an expression of importance. This can then be used by the networking-service for pre-emptive network scheduling.

2.4.3.5 Information Priority

Most distributed real-time systems are deployed on non-real-time networks, such as TCP/IP or UDP/IP and Ethernet. Real-time systems however require priority to be ensured and enforced end-to-end, to avoid unbounded priority inversion. In OpenSplice DDS™ this is solved by providing the ability to configure *priority lanes* throughout the system that ensure that high-priority data can pre-empt lower priority data. Priority lanes also bring coping with priority inversion under complete control of the engineer deploying the system.

2.4.3.6 Information Urgency

Distributed systems often have to deal with the trade-off between *latency* and *throughput*. OpenSplice DDS™ provides a mechanism to control this trade-off while preserving time- and priority properties associated with the data. A time-budget can be associated with data, that expresses the amount of latency that is acceptable for the distribution, allowing the middleware to optimise network utilisation and reduce load on the *central processing unit* (CPU).

2.4.3.7 Network Partitioning

Large scale distributed systems often have independent information flows with vast data volumes. OpenSplice DDS™ provides — apart from the logical partitioning capabilities provided by the DDS specification — means to partition information distributed on the network level. This way hardware filtering supplied by modern routers/switches or other network hardware can be utilised to achieve even higher total system throughput. The logical partitioning can thus be mapped onto the physical partitions.

2.4.4 OpenSplice DDS™ Management

The DDS paradigm has in practice shown to be difficult to grasp for a lot of users. That can probably partly be attributed to the fact that the paradigm differs quite a bit from the better known ones. As already pointed out in section 1.4, the level of abstraction of the paradigm and the decoupling in time, synchronisation and space can make comprehension of system behaviour difficult in non-trivial deployments. In typical multi-node systems with stringent performance requirements, the network will have to be managed and monitored to ensure proper behaviour when deployed. Insight in things like the communication patterns that are occurring, distribution of processing and publishers/

subscribers and oscillation of resource usage across system components can help the end users to optimise the system.

In case of a system malfunction, the source of the fault will need to be identified. This is nothing different from traditional networked systems and there is an area of research dedicated to finding solutions for network management challenges. In the field of *Network Management* the focus is however primarily on physical network and protocol management. Management of OpenSplice DDS™ will, apart from the traditional physical network management, also have to deal with the logical network and related issues.

The huge amount of management data that can be generated in a runtime system and the lack of automated mechanisms for anomaly detection or optimisation support the requirement for human system administrators. Visualisation of the management data can be applied in order to make use of the advanced capabilities of the human visual system.

In answer to research question II we can conclude that there is a clear need for visualisation and network management within the context of OpenSplice DDS™ systems. There are specific analogies with traditional network management on the physical level that may translate to the logical level too. Finding ways to manage the logical network will be a challenge because of the differences with traditional systems.

2.5 Splice as a System Design Approach

Very often DDS is put forward when there are stringent requirements on various (non-functional) properties of a system that has to be developed or deployed, for example due to the mission-critical nature of it. There are however a lot more advantages to the design paradigm underlying DDS. This has already been explained a bit in the heritage of OpenSplice DDS™ in section 2.3 (Splice).

A well known and very popular programming paradigm is *object-oriented programming* (OOP), which focusses on the abstraction from data types. The reasoning behind this is that this allows for much easier replacement of certain functionality if needed — because of the abstraction of implementation details — aiding in modular designs. The concept of objects also has an intuitive analogy with real-world objects, which has led to the paradigm being promoted as a means to reduce the difficulty of designing software.

This would then reduce the effort needed for creating- and increase the quality of the design.

The power of the analogy of object abstractions and the real-world entities for analysis and simulation was first explored in the simulation tool Simula, which later turned into a full programming language. Simula has had a big influence on the OOP languages we have today. Bjarne Stroustrup[1] has acknowledged that Simula 67 was the greatest influence on him to develop C++.

Because all data access in the OOP paradigm is necessarily achieved through procedural interfaces, the client-server paradigm was naturally chosen as the interaction pattern for the OO approach.

For simulation and analysis the object abstraction is often very useful. When building a simulation, the target is to create an artefact that shows behaviour analogous to a real-world object; the applicability of the paradigm is obvious. Object abstraction has also proven to be very beneficial for analysis situations, where the hierarchies introduced by object abstractions can greatly ease comprehension of the system. The simulation and analysis tasks however greatly differ from the design task. Analysis focusses on understanding functional components of a system and their interactions; simulation on mimicking real-world object behaviour. Designing however is all about implementing desired functionality conforming to specific non-functional properties (*e.g.*, scalability, fault-tolerance, footprint, performance, maintainability, etc.). Implementation is non-trivial due to quality goals that are formulated and the major design problem is to structure the design to a workable compromise with an acceptable quality level. This issue is not tackled by the object abstraction paradigm, although it is often proposed as an all-round solution for programming problems.

For the design of components of a system, a suiting paradigm should always be chosen to match the components needs. This obviously includes the application of the OOP paradigm where appropriate. So it is important to distinguish the scale at which a specific paradigm is applied. OpenSplice DDS™ provides support for several implementation paradigms, including OOP.

[1]The creator of C++

2.6 Conclusion

This chapter introduced the concepts on which OpenSplice DDS™ is based; the publish/subscribe paradigm, data-centricity, the DDS-specification and the heritage of OpenSplice DDS™. Some architectural advantages of the implementation and some OpenSplice DDS™ specific extension on the specification have been explained also.

Management and Deployment Difficulties The complexity and nature of OpenSplice DDS™ have in practice revealed to be complicating the management of (to be) deployed systems. A growing number of highly specialised tools are created for OpenSplice DDS™ that aid in displaying specific system metrics or behaviour. The fact that many of these specialised tools are created for scenarios where insight in the system is lacking, stipulate the demand for improved ways of monitoring and managing OpenSplice DDS™ based systems.

Chapter 3

Network Management

In this chapter an introduction will be given on existing network management techniques and tools and how they are applied.

3.1 Introduction

The network management discipline generally deals with vast amounts of data, due to the composition of the infrastructure of heterogeneous devices in a complex, large-scale topology. Often, the infrastructure itself and the working of the supporting protocols are not completely understood. Pras *et al.* (2007, pp. 106–107) highlighted in the *IEEE Communications Magazine* that data analysis and visualisation is a key research challenge within the network management area. Visualisation exploits unique capabilities of the human visual system to efficiently detect patterns and anomalies (van Wijk, 2005, p. 78) and therefore naturally suits complex analysis tasks.

Barbosa & Granville (2010) sum up various efforts in the computer networks area that investigate the employment of information visualisation techniques. Mansmann & Vinnik (2006) proposed a mapping method utilising a treemap-like visualisation in order to gain deeper insight in network flow behaviour; Keim *et al.* (2006) developed a toolkit that anticipates potential anomalies by visualising typical network communication activities. Dobrev *et al.* (2009) used visualisation of the node interaction dynamics to find patterns in the polling cycle of stations and topology changes using. They used existing tools in their effort. Common in all efforts previously summed up, is that they used static visualisations. Barbosa & Granville (2010) present and evaluate an interactive visualisation technique for the study of the *Simple Network Management Protocol* (SNMP). They extended on the existing efforts by introducing interactivity in the visualisations of the traffic measurements, showing how interactive visualisations can improve understanding of the protocol. They visualised data resulting from the methodology proposed by the *Network Management Research Group* (NMRG) in RFC5345 (Schönwälder, 2008), which presents a systematic methodology for measurements and statistics generation of SNMP traces in order to identify usage patterns of the protocol (Barbosa & Granville, 2010, p. 74).

3.2 Challenges

In network research, network management is still identified as one of the key challenges (Al-Shaer *et al.*, 2009; Pras *et al.*, 2007; Schönwälder *et al.*, 2009). Pras *et al.* (2007) identify seven important challenges within the area of network management: *network management architectures*, *distributed monitoring*, *data analysis and visualisation*, *ontologies*, *economic aspects*, *uncertainty and probabilistic approaches* and *behaviour of managed systems*. The three areas that are of most interest within the context of network management for OpenSplice DDS™ are highlighted here.

Architectures — A lot of research has been done with regard to network management architectures. For example the *Internet Engineering Task Force* (IETF) has worked on specifying three different architectures for network management: a *management information base* (MIB) based approach, a script based approach, and a remote operations based approach. Also in the telecommunications area several recommendations have been defined by the telecommunication standardisation sector (ITU-T) that specify functional, physical, information, and logical layered architectures. However, all these architectures rely on a client/server model, making them not very well suited for managing dynamic networks like the logical DDS network, P2P networks or ad-hoc networks. Traditional management tasks can become quite difficult for a centralised manager to perform and these systems therefore require distributed and cooperative management capabilities, which is a field of research that still requires a lot of attention.

Distributed Monitoring — For monitoring to be effective, the state of the system needs to be available at the required accuracy, at the right place and against minimal cost. A lot of monitoring tasks even require the state to be available in real-time. In order to implement this in a scalable fashion, the monitored network itself needs to implement this functionality in a distributed way and provide monitoring primitives to management or monitoring applications in end systems. There are a lot of challenges in realising such a distributed monitoring layer, ranging from defining a set of protocols in order to let the layer perform monitoring to efficient state aggregation under constraints. Both Pras *et al.* (2007, pp. 105–106) and Al-Shaer *et al.* (2009, p. 38) describe challenges in this area quite extensively.

Data Analysis and Visualisation — The vast amount of data in monitoring and measurement data sets needs to be filtered, aggregated and visualised in order to make the data available to human operators in a meaningful and accessible format. Traditional systems often incorporate time-series displays and topology overviews. Many improvements to the traditional visualisations are needed in order to satisfy new visualisation needs like improved topological views that scale well with the growing number of network elements, interactive visualisation exploration, visualisation of non-volume related parameters like unusual traffic and unusual traffic pattern visualisation and (near) real-time visualisations of high throughput networks.

3.2.1 Relevance

Visualisation and system monitoring in OpenSplice DDS™ has a lot of analogies with traditional network monitoring and the challenges described above are therefore analogously applicable. Some aspects of the visualisation are even more complicated because of the differences between the logical network formed by the abstract *connections* between DDS entities and the physical network on which the logical network is deployed. Furthermore, because of the lack of actual connections on the logical level, an extra challenge of defining a useful, intuitive mapping and its semantics exists.

3.3 Network Management in DDS

The current relevant challenges in the network management area as listed in section 3.2 reveal many issues that are also relevant for a network management solution for DDS systems. In this section the need for network management in DDS systems will be discussed in order to answer research question II. In order to be able to answer research question III some possible solutions will be delved into based on a study of relevant publications. Effective management extends beyond tuning of a system and requires monitoring, interpreting and controlling the behaviour of both the hardware and the software resources of the distributed system.

3.3.1 Tasks

Components of distributed systems of any kind are connected with each other by means of a communication link, forming a network of the hardware and software resources that make up the entire distributed system. Management of these networks involves tasks like deployment, maintenance, monitoring and tuning. Deployment is concerned with the initial set-up of the system. Network monitoring in general is concerned with the surveillance of important performance metrics of networks to supervise network functionality, to detect and prevent potential problems, and to develop effective countermeasures for networking anomalies and sabotage as they occur.

The task of optimising system performance is very common across different domains. However, the optimisation goals, *i.e.*, the dimension(s) over which the optimisation is performed — and thereby the ways of optimisation that are applied — vary greatly in different domains. For example in the domain of defence systems the availability, predictability and fault-tolerance of the system prevail over other performance aspects. The systems are designed with stringent requirements for the worst-case scenarios. In a battle setting, the number of parallel tasks dramatically increases, but it would be unacceptable if this would cause for example the jitter on the fire-control to increase, since that would imperil the ship and its crew.

In the financial market however, performance in the sense of number of transactions performed per second is of the utmost importance. Optimisation is therefore carried out for best-case performance with the requirement of graceful degradation in case of a less than best-case scenario.

TACTICOS CMS In order to illustrate the complexity of systems in which network management capabilities are needed, the TACTICOS *combat management system* (CMS) developed by THALES will be described. TACTICOS is a highly advanced, mission-critical CMS, comprising command and control, command support and fire-control facilities for anti-air, surface, anti-submarine and electronic warfare as well as naval gunfire support, allowing the command team to assess and monitor the tactical situation, to plan and coordinate naval operations and to control the sensor and weapon systems. TACTICOS can be used in a variety of warships, ranging from destroyers and frigates down to fast attack craft size.

TACTICOS is based on a distributed computer architecture, applying a multi-node, multi-processor concept in a battle-damage resistant configuration. The system is in use with more than 20 navies in over 136 ship systems, executing 2 200 executables in a fully distributed manner on heterogeneous platforms consisting of over 150 CPUS (embedded as well as workstation class), implementing the data-centric approach described in section 1.4 (Data Distribution Service). TACTICOS has no SPOF and is self-forming and self-healing. All sensors and actuators are networked devices; many of which are safety critical and have hard-real-time requirements, providing updates at 4 000 Hz. The TACTICOS CMS is capable of handling over 1 500 tracks while delivering automatic multi-sensor data fusion, threat evaluation, weapon and sensor advise and scheduling.

The data-centric approach and component-based development of TACTICOS enables the co-existence of legacy software with components written in programming languages that didn't even exist when the first components were developed, *e.g.*, Java™. All these autonomous components — ranging from small software-modules, up to complete 'wrapped legacy-clusters' — communicate with the *OpenSplice DDS™ information-backbone*. Autonomy is an essential requirement in dynamic/spontaneous systems, which is reflected in the fact that components are entirely decoupled, *i.e.*, are not aware of each other. The only interactions take place with the information backbone that provides access to a shared *information-model*. The information-model represents a stable basis for the components to work upon.

Availability of information at any place and at any time allows applications to join the system at any time. Failing applications can be restarted, effectively re-joining the running system. The decoupling of applications from each other and the application's focus on simply processing information also supports fault-tolerance by allowing redundant and replicated components without increasing their complexity. Obviously, this complexity doesn't magically vanish; it has gone into the information-backbone. This way it has only been implemented once and without complicating the applications connected to the information-backbone. In TACTICOS this is utilised to create a huge system without unnecessarily complicating the components with functionality related to the distribution of data. The data distribution service must adhere to the defined QOS like reliability, persistency and latency of the information.

In order to monitor and manage systems like this, the architecture must scale well and be able to deal with the heterogeneity in processing power.

3.3.2 Network Management Architectures

The architecture that underlies the network monitoring tools used for network management tasks in a distributed system defines the limits of the management solution. The suitability of a network management architecture for DDS-based systems should therefore be carefully considered in order to ensure that all envisioned network management tasks can be supported by it. This includes functionality surpassing the basic visualisation related network management capabilities, like autonomous optimisation. This section will discuss two fundamentally different architectures and their applicability in the DDS-domain in order to answer research question III and research question V partially.

3.3.2.1 Platform-Centred Architectures

Traditional network management architectures follow a platform-centred paradigm. The SNMP is probably the most prominent example. System data is monitored and collected by agents which are accessible via management protocols. These architectures are however inherently limited with regard to their scalability and — in heterogeneous distributed systems like many DDS-based systems — the semantic heterogeneity of management data imposes a further limitation for a platform-centred management architecture. Developing management applications that are able to handle the diverse semantics of operational behaviour of hardware and software resources is getting so complex and expensive, that it is becoming impractical to implement.

Platform-centred architectures also have limited application in high-speed distributed systems, where the architecture induces very high data collection rates and introduces high bandwidth-needs to get the management data of the system at a single workstation for processing and analysing. Especially since centralised management will tend to increase data access rates when the system is least likely to be capable to handle them. This is another symptom of the bad scalability of centralised paradigms, which imposes a limit on the number of network entities that can be monitored at a given time; the scaling limit is dictated by the available bandwidth and processing power of the central monitoring unit.

In general, a central monitoring entity is also a SPOF. When the centralised component fails, control is lost and the network will be without surveillance. This means that the availability of the entire network management architecture depends solely on the functionality of a single component.

A central entity is often also very limited in its view on the network, because it has hardly any means to monitor interactions between devices that do not include the central entity.

3.3.2.2 Decentralised Architectures

A decentralised approach to the network management architecture is better suited to current high-performance distributed systems in many aspects. While most functional requirements can be fulfilled with a centralised as well as a decentralised architecture, satisfying the non-functional requirements is more challenging when the management architecture doesn't match the system on which it is deployed. For management tools (just as many other applications) there is obviously no such thing as the perfect single paradigm architecture. Both architectures should probably be augmented to attain an optimally flexible solution, benefiting from the centralised decision making of a platform-centred approach and the efficient data-gathering of a decentralised approach.

The issue of heterogeneous management data can be overcome in a decentralised system by applying the semantics translation at the source, ensuring system wide generality of the data. An example of a basic system metric that has varying semantics is CPU-load. Whether 85% CPU-load is problematic depends — obviously primarily on the application requirements, but also — on the *operating system* (OS), scheduling-class, processor-type, etc. For example on a CPU with six cores, a CPU-load of 85% can still mean that there is one core fully available, allowing a program to be executed in a deterministic way, while on a single-core machine the same situation can be a trigger for caution.

3.3.2.3 Cooperative Architectures

A special case of a decentralised architecture is the cooperative architecture. It extends on the decentralised architecture by making the distributed entities cooperate at some level to achieve functionality crossing entity boundaries.

In typical DDS systems, the connection-less architecture, dynamics and scale make it quite difficult for centralised managers to perform traditional management tasks, such as end-to-end QOS-monitoring and fault handling. The dataflows in a DDS system can conceptually be seen as a form of overlay network — albeit with the reservation that the true underlying concept is fundamentally connection-less — analogue to the way overlay networks are used in P2P networks. Many of the network management difficulties encountered in such networks are analogous to DDS systems too. Such networks are considered to require cooperative management capabilities, for example, to collect statistics, repair faults or pass meta-information about the network.

Highly segmented or ad-hoc networks may not be manageable from a centralised system, which implies that cooperative management often will be performed somewhat autonomously. A difficulty is however introduced by the fact that in automated management mechanisms multiple control loops may be created, which work well in isolation but may endanger the system stability because of interference with each other. Controlling and managing highly dynamic environments, such as P2P and ad-hoc networks, is still considered to be far from trivial by Pras *et al.* (2007, p. 105).

One advantage that can be leveraged when using an overlay over the connection-less paradigm underlying the DDS specification, is that for network management and monitoring any level of zoom can be selectively applied to the conceptual overlay network. This allows the complexity of the management and decision tasks to be greatly reduced, because selective zoom can be applied only to the components that need optimisation or that display unexpected behaviour. This is possible because of the lack of real end-to-end 'connections'. In the publish/subscribe paradigm every entity has an accompanying set of publications and/or subscriptions. Now consider two connected DDS subsystems A and B. When network management tasks need to be performed in subsystem A, the complete topology of subsystem B can be aggregated into one conceptual unit with the combined publications and subscriptions of all the entities in subsystem B, thereby greatly reducing the number of managed entities. The same principle can equally be applied within subsystem A. Entities can be aggregated to combined entities per node, subnet, etc.

3.3.3 Distributed Monitoring

This section discusses challenges and solutions for distributed monitoring and data-gathering, answering sub-question III.b and sub-question III.c.

A popular and often easy to implement approach to retrieving management data in distributed systems is for a query unit to ask for the data from the remote nodes of interest. The requested data is then sent from the source nodes to the data sink, effectively implementing a *pull-based* collection strategy. A pull-based approach at query-time has some severe limitations for larger or highly heterogeneous distributed systems. Primarily, there is the potential for very large latencies in getting the desired data out of a multitude of source nodes randomly distributed across the physical network. In a centralised approach like this there will always be a trade-off between the amount of work performed at the time the data is generated and the work performed at query-time. Query-time processing generally introduces latencies and indeterminism that is not acceptable for monitoring applications. Secondly, real-time monitoring using a pull-based approach can result in excessive polling rates, leading to errors due to the perturbations introduced by the polling itself.

Pulling the needed data by querying on the other hand also provides an advantage of being able to reduce the amount of transmitted data at the source, *i.e.*, before distribution, by letting the source perform selection based on the query. This is an effective way of reducing the strain on the network while distributing the computing load. For a distributed monitoring application in OpenSplice DDS™, both the processing on and the distribution of the management data needs to be done in a scalable way.

3.3.3.1 Compression

In order to support real-time monitoring, the influence of the monitoring on the system dynamics has to be kept as low as possible. The management information thus needs to compressed at the source. This compression can range from basic aggregates like average, standard deviation and minimum/maximum values, to more complex aggregates. In general compression can be seen as applying a *health-function* that reduces management information to a smaller meaningful representation that suits the analysis currently performed. If meaningful, the compression can go as far as aggregating all locally available management information to a single *health*-index. In general, any dimensionality reduction technique can provide a potentially useful compression. The most obvious dimensionality reduction techniques in this context are compression based on information theory and projection (*e.g.*, principal component analysis).

In contrast to compression and projection, there are also dimensionality reduction techniques that do not alter the original representation of the variables. Feature selection or filtering techniques do not alter the original representation of the variables, but merely select a subset of them, preserving the original semantics of the variables. This has as advantage that — since the original semantics are preserved — the interpretability by domain experts is preserved too.

Aggregations or complexity-reductions are often applied in fields where the amount of information is simply too large for regular inspection or analysis, or for example to increase efficiency of machine learning algorithms. The dimensionality of the monitoring variables will probably not ever come close to the amount of dimensions in for example the area of bioinformatics, but the techniques that are used in such fields for complexity reduction can still be used beneficially in the simpler case.

3.3.3.2 Delegation

In order to achieve a distributed and scalable monitoring solution, the monitoring algorithms need to be distributed, otherwise sharing load across the monitored network cannot be achieved. In order to be able to provide a flexible solution with regard to distributed monitoring in OpenSplice DDS™, the monitoring capabilities must not be rigid, *i.e.*, consist of a predefined set of monitoring primitives. When the monitoring algorithms are not statically defined, network monitoring entities still have the freedom to apply traffic reduction or other means to reduce the strain on the system or keep up with evolving systems.

For example in growing or large scale DDS systems, components of the system may be extended by external entities. The heterogeneous networks that arise cannot be forced to comply with a rigid set of monitoring features and can not easily provide the type of access needed for arbitrary monitoring needs of a central entity. If network entities are able to be dynamically programmed to perform specific tasks, the monitoring computations and logic can be dispatched to the network entities.

The idea of dispatching dynamic monitoring routines is crucial for enabling distributed monitoring in distributed systems in a scalable way and is coined *monitoring by delegation*. Every node or monitored entity capable of executing distributed monitoring logic can also provide transformations from local to common representations for low-level metrics, implemented by the subsystem creator. This solves the issue of heterogeneity of management data by defining a common understanding and letting the implementer of the subsystem define the transformations to the common representation. The amount of complexity of the distributed logic is virtually unlimited, allowing for example distributed automatic optimisation algorithms to be deployed.

3.3.3.3 Metrics

When all metrics and statistics available in the middleware can be locally accessed, then with the solution explained above, defining a set of management data beyond the low-level system primitives (*e.g.*, CPU-load, memory-usage, etc.) is not needed. The management delegates can aggregate over low-level system metrics, middleware internal metrics and higher-level intra-delegate communications, and provide that information to whatever management station interested. The set of metrics should thus not be limited to the low-level metrics. The sole reason for the need to define a basic set of low-level metrics is because they may need transformation to a global representation. Besides the basic set, generally as many as possible other metrics of potential interest should be made available. Due to the flexibility of the solution, the total set of available metrics can be augmented whenever needed.

3.4 Conclusion

In this chapter the field of network management has been introduced, shedding light on typical problems encountered in network management of large scale systems, complex topologies and highly heterogeneous compositions of the systems. Processing network traces and similar high-volume network statistics data in order to identify usage patterns and visualise protocol usage in order to gain understanding in how the protocols are used is one of the primary goals of network monitoring.

In order to support such processing and visualisations, the network needs to support the analysis and monitoring functionality required. Therefore the architecture, monitoring approach and data analysis need to be suitable for the scale of the networks which need to be monitored.

In section 3.3.3.2 (Delegation) the concept of dynamic delegation is introduced that allows for a monitoring implementation that fully suits the DDS environment. It is scalable, fully distributed and extensible. It furthermore does not require a central entity and supports autonomous state aggregation under constraints, tackling a lot of issues with network management systems. The proposed solution fully leverages the functionality provided by the DDS environment in order to overcome some of the limitations of similar systems for traditional networks. In chapter 5 (Design) a possible design for such a framework is described.

Chapter 4

Visualisation

This chapter will introduce the general concept and goal of visualisation. The goal of this chapter is to identify the role of visualisation in network management and to identify possible issues that will need to be resolved or focussed on when creating visualisations.

4.1 Introduction to Visualisation

There are several fields of research with regard to *visualisation* and different kinds of visualisation are distinguished. The terms *data visualisation*, *scientific visualisation* and *information visualisation* are frequently seen. In this section an introduction to the field of visualisation will be given, including an overview of topics considered key research and challenges within the field.

The Holy Grail of visualisation, regardless of the specific area in which it is applied, is for users to gain *insights*[1].

4.1.1 Data Visualisation

The domain of *data visualisation* focusses on the visual representation of *data*, defined as "information which has been abstracted in some schematic form, including attributes or variables for the units of information" (Friendly & Denis, 2001, p. 2). In the field of

[1]Generally the definition of *insight* as it appears to be used in publications is quite broad, *e.g.*, a deepened understanding, an intellectual breakthrough, unexpected discoveries, seeing intuitively, etc. Merriam-Webster's definition can be found on page 43

data visualisation, statistical graphics and cartography, graphical depictions of quantitative information are often seen. Both statistical graphics and thematic cartography are concerned with the visual representation of quantitative and categorical data, albeit with different representational goals. Cartographic visualisation is typically constrained to representations in the spatial domain; statistical graphics applies more broadly to any domain in which graphical methods are used to aid in statistical analysis. Both areas however share the common goal of visual representation for exploration and discovery, based on quantitative data.

Many metrics and statistics available in OpenSplice DDS™ could already be visualised with one of the many available graphic methods of visualizing patterns, trend and indications from the domain of data visualisation.

4.1.2 Scientific Visualisation

The field of *scientific visualisation* focusses primarily on the visualisation of three-dimensional phenomena like architecture, meteorology, medical- and biological structures, etc. Emphasis is often on realistic representations of illumination sources, surfaces, volumes, eventually visualised with a dynamic (time) component (Friendly & Denis, 2001). The field of scientific visualisation has little relevance to visualisation of management data in networked systems, although some visualisation techniques can probably be reused.

4.1.3 Information Visualisation

The term *information visualisation* can refer to computer generated interactive graphical representations of information as well as the process of producing these information visualisation representation, in which it is concerned with the design, development and application of computer generated interactive graphical representations of information (Chen, 2010, p. 387). Information visualisation often deals with abstract, non-spatial data and how to transform such data to meaningful and intuitive graphical representations. Creating these transformation is a creative process much like arts, aiming to communicate complex ideas or data to its audience.

4.2 Insight

As introduced in section 4.1 (Introduction to Visualisation), the main purpose of visualisation is to provide insights for users, or as Card *et al.* (1999, p. 6) state; "The purpose of visualisation is insight, not pictures." While this seems quite obvious and probably matches most peoples gut feeling, it is really hard to reason about *how* exactly insight is gained by people. Information visualisation often involves complex data transformations and sophisticated representations. However, the ultimate goal of visualisation is to exploit the remarkable capabilities of the human visual system in identifying trends, patterns and peculiarities in datasets.

According to Yi *et al.* (2008, p. 1) the concept of insight is — at least within the domain of information visualisation — not yet well defined or understood. A few definitions exist, but there is not a commonly accepted one in the field of work. The Merriam-Webster dictionary defines the following:

> *Insight*: The capacity to discern the true nature of a situation; The act or outcome of grasping the inward or hidden nature of things or of perceiving in an intuitive manner. — Merriam-Webster

North (2006) stipulates the fact that defining insight is challenging and has proceeded by defining five essential characteristics of insight. These characteristics can be used to evaluate measurement methods for insight. Understanding the meaning of insight is at least necessary in order to evaluate visualisation techniques, but having better understanding about how insight is acquired can also facilitate the creation of visualisations in a more insight-oriented way.

4.2.1 Insight Characteristics

The characterisation of insight by North (2006, p. 6) provides a good starting point for defining and validating insight-based evaluation techniques. It is however good to stay aware of the potential pitfall of starting to use the characterisation as being a definition, as for example done by Plaisant *et al.* (2008, p. 120) who even go as far as stating that "Insight can simply be defined as a nontrivial discovery about the data or as a complex, deep, qualitative, unexpected, and relevant assertion", which I personally expect to be cutting corners too much.

North (2006, p. 6) recognises the following five important characteristics of insight:

Complex — Insight is complex, involving all or large amounts of the given data in a synergistic way, not simply individual data values.

Deep — Insight builds up over time, accumulating and building on itself to create depth. Insight often generates further questions and, hence, further insight.

Qualitative — Insight is not exact, can be uncertain and subjective, and can have multiple levels of resolution.

Unexpected — Insight is often unpredictable, serendipitous, and creative.

Relevant — Insight is deeply embedded in the data domain, connecting the data to existing domain knowledge and giving it relevant meaning. It goes beyond dry data analysis, to relevant domain impact.

According to North, insights that rank highly in each of the above characteristics are typically the most interesting.

4.2.2 Gaining Insight

Very often, insight is not a well-defined end result of an insight gaining process, but a by-product of some kind of (random) exploration. It is furthermore difficult to define a *unit of insight* that can be used in insight-based evaluation studies. The fact that insight frequently acts as a stimulus for new insights is also a clear hint that insight is not an end-product pur sang. Understanding the procedural aspect of insight, *i.e.*, how people gain insight, is therefore considered to be of great value for enhancing the way new visualisations are created.

4.3 Visualising Dataflow

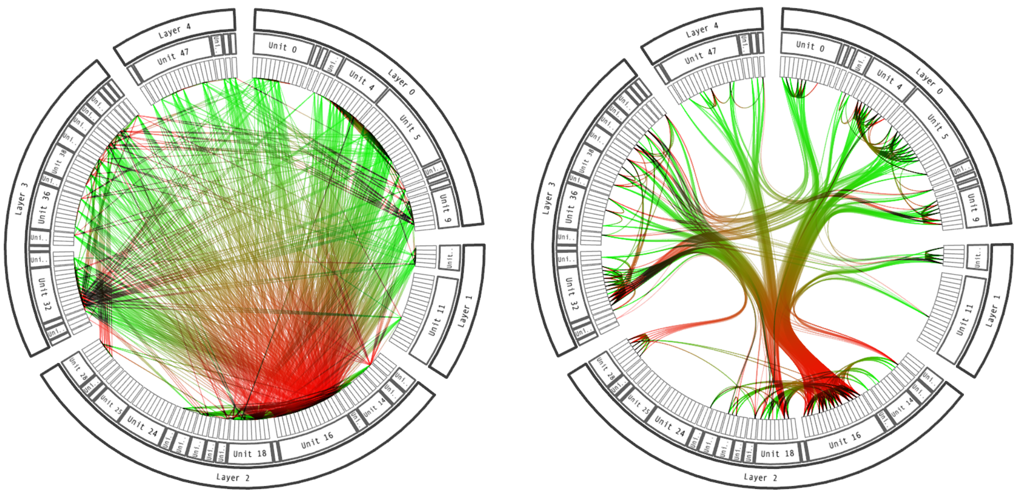
Many metrics and aggregates of metrics that can be retrieved from an OpenSplice DDS™ system can be visualised with techniques from within the domain of Data Visualisation. While there possibly are specially tailored data-visualisations that can be thought of, generally common visualisation techniques and tools can be applied to OpenSplice

DDS™ management data in a generic manner. As explained in section 4.1 (Introduction to Visualisation), there are several areas of visualisation. Particularly the field of Information Visualisation is concerned with visualisation of data that is often non-spatial and needs to be transformed to a meaningful representation.

4.3.1 Topology

The physical nodes in a network can be laid out in a graph-like visualisation according to detection of existence, revealing the network connectivity of the OpenSplice DDS™ nodes by arcs between two nodes. A lot of topology or connectivity/link visualisations are described (Chen, 2010; Dobrev *et al.*, 2009; Salvador & Granville, 2008). A very powerful technique for visualising hierarchical relations is presented by Holten (2006). It is very important that the topology and dataflows are visually presented in a scalable way. In Figure 4.1(a) an example of a possible dataflow visualisation with direct (straight) edges is given. It is immediately clear that this doesn't scale well. In Figure 4.1(b) the same data is visualised, however with edge-bundling applied. Both visualisations in Figure 4.1 use a colour-gradient to indicate direction. Holten (2009, chap. 6) has also refined the technique for relations lacking a hierarchical component. These visualisation can also be applied at the DDS level, where connectivity information can be visualised by edges representing matched publications and subscriptions. These edges represent potential flow of data per type. Attributes of the matched subscriptions, like reliability properties, urgency, etc., can be made distinguishable by means of different edges-types.

Superimposition of the physical network connectivity and the matched subscriptions will provide insight in the load or availability requirements for the physical connections. Allowing a user to selectively display matched subscriptions — using for example the bundle-based interaction pattern described by Holten (2006, sec. 4) — or network channels will allow the user to gain specific insights regarding the mapping in order to optimise for reduction of traffic over the physical network, reducing network load and latency, increasing throughput versus increased parallelism and reduced CPU-load. The cost-model depends on the actual network and deployed applications.



(a) *Dataflow without edge-bundling* **(b)** *Dataflow with edge-bundling*

Figure 4.1: *Example dataflow visualisations*
Source: (Holten, 2006, p. 754)

4.3.2 Dataflow Analysis

In a system composed of a large number of components, many times even specialists don't have a complete overview of all interactions. It can therefore be really hard to find sources of trouble or causes for things like mediocre performance, high jitter or unwanted oscillation of resource usage on system components. For example the oscillating behaviour of components can be a cause of too little throughput being achieved. On the other hand, knowing which resource usage oscillates on components can also provide clues for improving parallelism or better distributing load across the system. It would therefore be valuable to be able to visualise complete dataflows through the system and be able to compare the mapping of the flow on the physical network in a way analogue to the mapping described in section 4.3.1 (Topology), allowing potential paths through the system to be identified.

The dataflow analysis can be based on matched subscriptions and the knowledge about which `Publishers` and `Subscribers` belong to a `DomainParticipant`. Combining matched subscriptions with the grouping per `DomainParticipant` will allow for the potential flow of data in the system to be constructed. The QoS-policies of the matched subscriptions describe some properties of the dataflow, which can include for example aspects like minimal update frequency (`DEADLINE` QoS-policy), transport priority (`TRANSPORT_PRIORITY` QoS-policy), etc. It is however not possible to determine actual volume of the data without measurements on the running system, because the QoS-policies only specify extremes.

When subscriptions are not matched to publications due to *requested–offered* (RxO) mismatches of QoS-policies, this may influence dataflow. Any analysis of topology and dataflow should explicitly support highlighting of unmatched subscriptions due to RxO QoS-policy mismatches, because those are very often not by design and are hard to identify.

4.3.2.1 Basic Dataflow Example

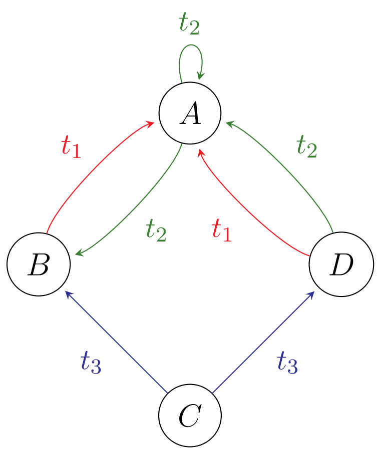
Suppose we have a system composed of four `DomainParticipants`: A, B, C and D which publish and/or subscribe to `Topics` t_1, t_2 and t_3. The publications and subscriptions of the nodes are listed in Table 4.1 below.

Table 4.1: *Basic dataflow example*

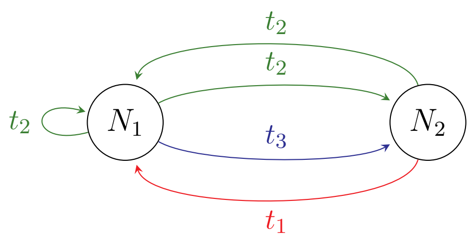
Participant	Node	Publication(s)	Subscription(s)
A	N_1	t_2	t_1, t_2
B	N_2	t_1	t_2, t_3
C	N_1	t_3	
D	N_2	t_1, t_2	t_3

When this data is placed into a directed graph — as done in a fundamentally basic way in Figure 4.2(a) on the next page — the dataflow across the `DomainParticipants` can be easily distinguished. Equally in Figure 4.2(b) the dataflows are placed in a directed graph grouped by nodes. The potential flow of data between nodes can easily be seen in this representation too. The graphs are as basic as possible for the sake of the example; eye-candy is less relevant here than the contained information.

While the graphs in Figure 4.2 provide a better overview of the dataflow than Table 4.1, it has still very little added value for optimisation tasks, since it doesn't provide all the information needed. Ideally the information in both graphs should be contained in one graph, revealing the mapping of the logical connections onto the physical connec-



(a) *With four* `DomainParticipants`



(b) *With two nodes*

Figure 4.2: *Dataflow graphs*

tions. Figure 4.2(a) displays logical connections on the DDS level, whereas Figure 4.2(b) shows the logical connections across the physical connection(s) between nodes.

4.3.3 Dataflow Highlighting and Selection

In a visualisation of dataflow in a system, a specific flow of data can be selected[1], revealing connected elements in the graph. For example the subgraph composed of all reachable graph-nodes in Figure 4.2(a) from `DomainParticipant` B is different than when the subgraph is composed starting in `DomainParticipant` C. This can effectively be used to reduce the amount of visible nodes. Further filtering can be applied on any parameter, allowing the end-user to reduce the complexity of the system

Nodes and arcs can also be decorated with applicable metrics. For example in a node-view of the system, the CPU-load can be added as a decorative aspect of the nodes mapped on a suitable visualisation dimension like colour or size, a bar-graph, etc. This allows end-users to gain insight regarding the mapping of the logical dataflows on the physical components of the system.

[1]An example selection method is depicted in (Holten, 2006, fig. 16, p. 747) using an edge-bundled visualisation like in Figure 4.1

4.4 Conclusion

There are obvious advantages for highly task-specific visualisations, especially in time-critical analysis or prediction activities (Treinish, 1998, p. 407). While highly generic systems can often be employed in such a way that they provide similar functionality, the lack of focus on a specific task in the interface can increase the learning time beyond acceptability. Generic visualisation systems on the other hand provide very useful functionality when prototyping new applications, analysing new issues or generally gaining insight in data.

The need for task-specific or highly specialised visualisations thus evolves out of identified specific tasks. The use of such visualisations will with a high likelihood be identified, so any visualisation solution that is created optimally needs to support both types of visualisation, promoting high-level reuse of underlying tools and design elements.

The evolutionary process of visualisations within the scope of the network management solution proposed in this thesis will certainly include episodes in which improvement of visualisation methods and techniques will be the central issue. It is therefore strongly suggested that this field be not neglected but gets its fully deserved attention.

Chapter 5

Design

In chapter 3 (Network Management) an introduction and analysis of network-management tasks is given in the context of OpenSplice DDS™. The goal of this chapter is to answer research question V by describing a design that fits within the boundaries defined by the answers given in the previous chapters, supporting any potential visualisation approach like described in chapter 4 (Visualisation).

5.1 Introduction

The design of a network management solution in OpenSplice DDS™ will have to encompass the findings of the previous chapters. The design may contain OpenSplice DDS™–specific aspects and terminology. The concepts underlying this solution can be more generally applied to DDS, but some parts of the solution may specifically leverage the capabilities and scalability of the OpenSplice DDS™ middleware. Furthermore is it only in DDS systems that actually see deployment in large-scale systems that good scalability is a requirement for the applied network management solution.

This chapter will provide more specific details and examples for the dynamic delegated monitoring concept introduced in section 3.3.3 (Distributed Monitoring) as a solution for distributed monitoring in a scalable and flexible way.

5.2 Global Design

In the following sections specific components and design choices will be discussed for a network management solution for DDS systems.

5.2.1 Architecture

The network management architecture that is the most natural fit for OpenSplice DDS™ is a distributed architecture. In section 3.3.2 (Network Management Architectures) platform-centred and decentralised architectures are compared. A decentralised architecture also allows for deployment of cooperative components and is the logical choice for an architecture for network monitoring in OpenSplice DDS™.

In a cooperative architecture, every node participating in the DDS-domain has essentially an active role in the network management and monitoring. Coordination can take place at practically all levels (*e.g.*, node, clusters of nodes or system-wide). By applying an architecture based on cooperation between components, the need for a centralised management entity is avoided.

5.2.2 Platform Abstraction

In order to overcome the possible limitations implied by the heterogeneity of the distributed system components, proper abstraction from this diversity has to be performed. In any approach to achieving a proper abstraction, it is essential that the component specific semantics can be translated to a shared semantic model. The most obvious approach is to perform the translation at the source, allowing for simple interpretation of metrics. The abstractions provided in this section are meant to illustrate the kinds of abstractions that are needed and the somewhat theoretical definitions strive to underline the importance of a definition that is conceptual by nature and not implementation or platform driven.

As also stated in section 3.3.2.2 (Decentralised Architectures), this approach ensures a consistent, system-wide interpretability of the metrics without complicating the network management tools. It is important to keep the set of metrics needing abstraction as small as functionally useful, to allow for easy adoption of emerging platforms. The semantics of the metrics need to be of value within the network management scope, *e.g.*, a percentage or amount of free disk space may not have the expected meaning within the context of the system; the amount of storage available for persistent data more closely resembles the sought for metric. The difference is small and the value may on many systems not even differ, but key is that the semantics of the metric are defined in such a way that the information needed within the network management scope can be easily deducted.

5.2.2.1 Abstraction Set

In Table 5.1 below a minimal set of metrics is listed that may be hardware specific and hence potentially need abstraction. For all metrics a format is chosen that is deemed useful. The scope of the abstractions is a single system, so any limitations imposed by implementation choices are also within the same system/nodal-scope.

Table 5.1: *Example set of metrics to be abstracted*

Metric	Unit
`available_cpus`	*n/a*
`cpu_usage`	*n/a*
`memory_stats`	MiB
`persistent_storage_stats`	MiB
`running_processes`	*n/a*
`host_name`	*n/a*

Definition (`available_cpus`)
Let `available_cpus` be the set $\{c_0, \ldots, c_{n-1} \mid n > 0\}$, where c_i is the identifier of processing unit i of the system. For this section, let $A = available\_cpus$.

The processing units available on a system and their arrangement varies widely. There are single-CPU, multi-CPU, multi-core or even multi-CPU–multi-core machines and some of them furthermore support logical processing units, like for example Intel®'s Hyper-Threading Technology. The chosen abstraction for this metric is based on the assumption that the hardware layout is not relevant within the domain of network management in OpenSplice DDS™, treating all available processing units equally.

The identifier for a processing unit needs to be unique, allowing other measurements for that processing unit to be related to the specific identifier. A possible *Interface Description Language* (IDL) definition for the metric is given in Listing A.1 in Appendix A.

In order to introduce hierarchy in the processing units, one could deviate from the most basic (consecutive) assignment of identifiers for CPUs in a system. If there are restrictions on inspecting the load on all processing units or if detail to the level of individual CPUs is not needed, the abstraction-layer can transparently define a surjective function $f : A \twoheadrightarrow B$. Optimally f is also left-total, allowing all elements in A to be accounted for in B according to f. In a system where only the total CPU-usage is available, f can for example be defined as $f(x) = 0$, which is both surjective and left-total giving the impression of only one CPU ($|B| = 1$) with identifier $c_0 = 0$.

Another useful example can be given in the context of a system equipped with Intel® Hyper-Threading Technology for all cores. Suppose that the actual CPU-usage is considered to be obfuscated by the usage measured on the logical Hyper-Threading CPUs and therefore the measurements of actual and logical cores need to be combined. In this case a function f needs to be defined, such that $|A| = 2\,|A'|$, mapping the virtual cores on the actual cores on which they are executed. Supposing that in A an actual core has an even identifier c_i and its accompanying logical core c_{i+1} has identifier $c_i + 1$, then f can be defined as $f\,(x) = \frac{x-(x \bmod 2)}{2}$.

Definition (`cpu_usage`)
Let `cpu_usage` be $\{(c_0, u_0), \ldots, (c_{n-1}, u_{n-1}) \mid \forall c_i, u_i : c_i \in A, u_i \in \mathbb{R}, n = |A|\}$, *and* u_i *is the usage of the processing unit* c_i.

Because the `cpu_usage`-metric is logically defined as the ratio $\frac{used}{available}$, there are no obvious reasons to assume that this metric actually needs platform abstraction. Due to its dependency on the `available_cpus`-metric — which *does* need abstraction — this metric is nonetheless defined in the minimal abstraction-set.

An IDL specification for the tuple describing the elements of the `cpu_usage`-metric and the set of tuples can be found in Listing A.2 in Appendix A.

Definition (`memory_stats`)
Let `memory_stats` be defined as the tuple (t, a, c), *where* t *is the total amount of memory,* a *is the amount of available memory and* c *the amount of cached memory in the system, all in* mebibyte *(MiB) and viewed from the DDS application scope.*

This metric represents statistics of the system memory as seen from the DDS-application scope, intended as a general measure for system health. Due to restrictions on certain platforms on the amount of memory that can be used by a process, the amount of memory available to a process can deviate from the low-level OS-metric, for which this abstraction is defined.

The tuple describing the statistics can be defined as in Listing A.3 in Appendix A.

Definition (`persistent_storage_stats`)
Let `persistent_storage_stats` be defined as `memory_stats`, except that t, a and c reflect the amount of persistent storage memory in the system, all in MiB and viewed from the DDS application scope.

This metric represents statistics of the persistent storage memory as seen from the DDS-application scope, intended as a measure for persistent-storage health. Due to restrictions on certain platforms on the amount of persistent storage that can be used by a process or user, the amount of persistent-storage memory available to a process can deviate from the low-level OS-metric for storage, *e.g.*, disk-space. Furthermore, not all platforms may have traditional disks for persistent storage, so this value needs to map to the statistics of the actual available persistent-storage space.

The tuple describing the persistent statistics can be defined like in Listing A.4 in Appendix A.

Definition (`running_processes`)
Let the set `running_processes` be $\{(p_0, q_0, r_0, s_0), \ldots, (p_{n-1}, q_{n-1}, r_{n-1}, s_{n-1})\}$, and p_i is the identifier of process i of the system, q_i is the name of the executable, r_i is the amount of CPU used by the process and s_i is the amount of memory used by the process.

Due to the potentially dynamic nature of the number of running processes (or the equivalent on platforms like VxWorks), this metric is not split like for example `available_cpus` and `cpu_usage`. In practice, network management may require more detailed statistics about a process than defined above. The simple definition can easily be extended when it is found to be incomplete. The metric can be defined like done in Listing A.5 in Appendix A.

5.2.3 Delegation

The monitoring by delegation concept is very powerful in enabling monitoring in a DDS system in a way that can be kept scalable, distributed and flexible. The underlying design allowing distributed monitoring — and potentially control — is described in this section.

The crux of monitoring by delegation is that the network monitoring routines can be dynamically modified and executed in a distributed fashion. This is achieved by letting all nodes in the system that need to participate in the network-management tasks run a *Delegation Agent* (`DelegationAgent`) instance. A `DelegationAgent` is an agent that is able to execute the dynamically distributed monitoring routines. The `DelegationAgent` is basically a `DomainParticipant` that is capable of starting and stopping a *Delegation Runtime* (`DelegationRuntime`). A `DelegationAgent` is controlled by control `Topics` and reports its status through status `Topics`. A `DelegationRuntime` is an executable that adheres to an interface defined between `DelegationAgents` and `DelegationRuntimes` and that performs specific monitoring functionality. In its simplest form, a `DelegationRuntime` is also a `DomainParticipant` and publishes its monitoring metrics by means of `Topics`. A `DelegationRuntime` can also use the platform abstraction API defined in section 5.2.2 (Platform Abstraction) to monitor low-level system metrics.

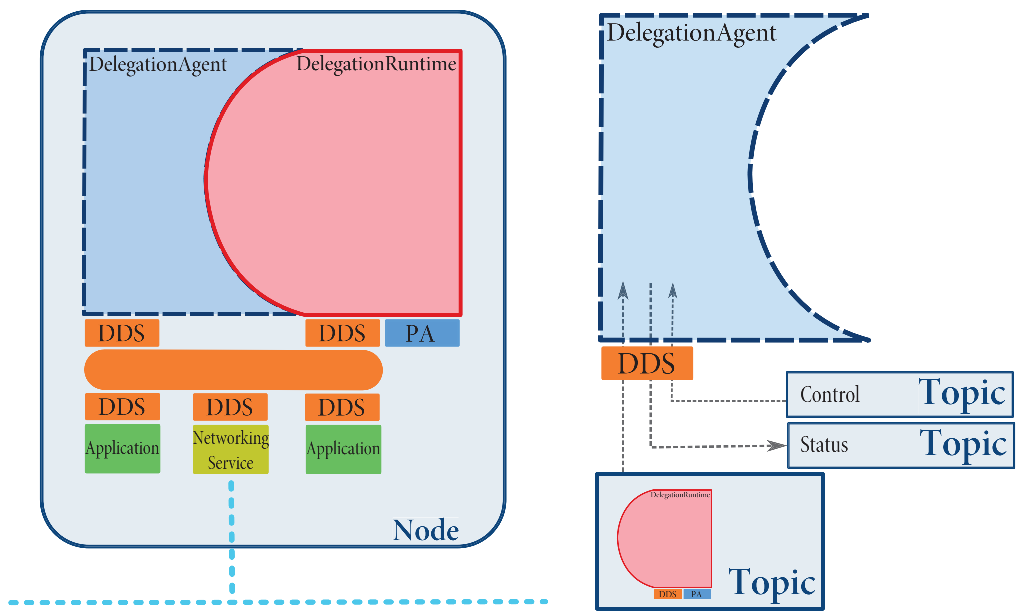
In Figure 5.1 on the next page, an overview of the above described components is given. It shows the `DelegationRuntime` using the platform abstraction layer by means of the PA-block in Figure 5.1(a), illustrates the fact that all component use DDS for communication and illustrates the concept of control and status `Topics`[1] being published and/or subscribed to through DDS in Figure 5.1(b).

An example definition in IDL of the interfaces between the `DelegationAgent` and `DelegationRuntime` is listed in Listing A.6 (Pseudo-IDL specification of delegation-module) in Appendix A.

5.2.3.1 Delegation Agent

A `DelegationAgent` is capable of dynamically controlling — *e.g.*, `start` and `stop` — `DelegationRuntimes`. By taking a subscription on a control `Topic`, the `Delegation-`

[1]Result `Topic` of `DelegationRuntime` not displayed, but concept is the same



(a) *Overview* **(b)** *DelegationAgent detail*

Figure 5.1: *Dynamic monitoring by delegation*

Agent can be controlled. DelegationRuntimes are distributed by means of a Topic. A DelegationAgent publishes its status — and that of the loaded DelegationRuntimes — through publishing a status Topic.

The DelegationAgent is the enabler for the dynamic monitoring capabilities; it acts as the runtime system for dynamic executables that form the monitoring and management solution. Conceptually it can be regarded as a controller for an *application virtual machine* (AVM) taking its commands from DDS by means of control Topics.

5.2.3.2 Delegation Runtime

A DelegationRuntime is a program library that can be executed by a DelegationAgent. It is distributed as a Topic containing an executable and parameters necessary for execution. While in a homogeneous system the format of the executable can theoretically be any kind of native format, in heterogeneous systems it is more useful to distribute the executable in a format that can be executed on all platforms. For example a bytecode format like Java™ bytecode or Microsoft's *Common Intermediate Language*

(CIL).

The advantages of using bytecode for `DelegationRuntimes` are the same as for regular applications; a few advantages of which are portability, compactness and efficiency. By using an intermediate language, only $n+m$ translators instead of $n \cdot m$ translators are needed to implement n languages on m platforms. When choosing a widely available intermediate language, this implies that the application logic only needs to be developed and distributed once (under the assumption that the translators for all m platforms are already available). The size of a program compiled to intermediate code is often much smaller than the size of the original source, which is important in the context of dynamically downloaded/distributed code. Thanks to the later translation to a specific native platform, the execution platform can make optimal use of the knowledge of the actual machine on which the program is executed, or even adapt to the dynamic behaviour of the program, *e.g.*, *just-in-time compilation* (JIT).

Except for a desire to keep the distribution of `DelegationRuntimes` possibly simple, there is no technical need to limit the format in which the `DelegationRuntimes` are distributed. As long as the `Topic` contains the information needed to start the `DelegationRuntime`, it can be any format. If not all platforms support the format, then it will become more complex, since in that case multiple versions of the same `DelegationRuntime` need to be distributed. The feasibility and need for supporting multiple executable formats would need to be studied when implementing. Whether Java™ bytecode is a proper format for the `DelegationRuntime` executable will have to be determined. It can at least be run on practically all platforms and there are many compilers available that compile to that executable format.

5.2.4 Distributed Monitoring: Two Examples

Many of the aspects covered in section 3.3 (Network Management in DDS) illustrate the need for distributed monitoring algorithms. Obviously not all monitoring routines need to be fully distributed and the functionality provided by the delegation framework described in the previous sections provides enough degrees of freedom to tailor the monitoring algorithms to a specific problem; including those that can more simply be solved in a platform-centred way. In this section an example algorithm will be described which highlights some of the aspects of the proposed system.

First a very basic algorithm is presented that simply gathers metrics from the platform abstraction API, aggregates the data and publishes the aggregated data at a fixed rate. Thereafter a more interesting example is discussed where the first example is extended with a feedback-loop in the monitoring algorithm. This illustrates a cascading monitoring solution that only publishes high-bandwidth data when the situation demands so. It consists of two `DelegationRuntime`'s; one that monitors the load on the CPUs, another one that publishes the list of running processes. The goal of the distributed monitoring example is to provide system-wide metrics regarding CPU-load and node-specific process-lists when the CPU-load has exceeded a threshold a specified amount of times.

These examples demonstrate many of the aspects discussed in chapter 3 in their simplest form; exemplary for the concepts, covering aggregation, compression, platform-abstraction and a fully distributed algorithm with a feedback-loop.

In this solution there is no central entity that controls the availability of the data. All monitoring takes place in a distributed manner and even decisions about the availability of specific monitoring data are autonomously made. Through (logical) partitioning and carefully defined mappings on physical partitions, the scope and distribution of the data can be controlled. It goes beyond the scope of a simple example to delve into the specifics of partitioning in this case.

5.2.4.1 Cpu-load `DelegationRuntime`

The most basic `DelegationRuntime` in this example is responsible for measuring the CPU-loads on a node and compressing those metrics. It uses the functionality provided by the platform-abstraction layer to retrieve the load on all processing entities of the system and the name of the host. The compounded relative CPU-load $\left(\frac{used}{available}\right)$, *i.e.*, the load expressed in an integer number representing the load as a percentage of the maximum CPU-load, is published with a frequency of 1 Hz.

This `CPULoadBasic` `DelegationRuntime` publishes the measurements in the `Topic` `CPULoad`, which is defined as in Listing 5.1.

Listing 5.1: *Definition of* ***`CPULoad`*** `Topic`

```
module DDSMonitoring{
  typedef string hostName;

  struct CPULoad{
    hostName host;
    long load;
  };
  pragma keylist CPULoad host
};
```

The topic is published with the default QoS-policy[1] except that the `RELIABILITY` QoS-policy is set to `BEST_EFFORT` and the `DESTINATION_ORDER` QoS-policy to `BY_-SOURCE_TIMESTAMP`, allowing subscribers of the data to retain proper order of the data. Since the data is published frequently, it doesn't need to be delivered reliably nor does it need to be persisted. The key on the data is defined on the name of the host, allowing for identification of streams of measurements from one host. The source–time stamp that is added by the middleware because of the `DESTINATION_ORDER` QoS-policy allows for proper ordering of the sampled data at the subscribers.

The pseudocode for the algorithm of the `CPULoadBasic` `DelegationRuntime` is shown in Program 5.1. It show usage of the abstraction-layer in lines 2, 3, 5 and 7. In lines 8 to 12 compounding of multi-CPU-load to a single relative load is performed.

A very efficient way to reduce the frequency of network management data updates on the network, would be to only publish the load when a (large) delta occurred. This would however change the QoS-policy needed and thereby complicate this example unnecessarily.

5.2.4.2 ProcessList `DelegationRuntime`

To illustrate the possibility of using a feedback loop in monitoring, the `ProcessList` `DelegationRuntime` is explained here, which task consists of publishing the process-list of all currently running processes and their statistics, allowing for remote analysis of which applications consume the available processing power. Because always publishing the process-list would create quite a large volume of data, the `DelegationRuntime` is only started or will only publish data in an assumed fault-condition.

[1]See Appendix B (DDS Specification) for more detailed information on possible QoS-policies and the defaults

Program 5.1: `CPULoadBasic` *pseudocode*

```
procedure CPULoadBasic($f$,$p$)
    $host \leftarrow$ getHostName()                ▷ Get hostname from abstraction layer
    $A \leftarrow$ getAvailableCPUs()          ▷ Get available CPUs from abstraction layer
    $n \leftarrow |A|$
    $s \leftarrow$ getCurrentTime()
    repeat
        $U \leftarrow$ getCPUUsage()              ▷ Get CPU-usage from abstraction layer
        $l \leftarrow 0$
        for all $u \in U$ do                 ▷ Compress multi-CPU-load to a single load
            $l \leftarrow l + \left(\frac{u \times 100}{n}\right)$
        end for
        $l \leftarrow round(l)$
        $m \leftarrow (host, l)$
        Publish($m$,$p$)                   ▷ Publish measurement $m$ in partition $p$
        $s \leftarrow s + \frac{1}{f}$                ▷ Determine time to wake for publication at $f$ Hz
        sleepUntil($s$)
    until stopped by hosting DelegationAgent
end procedure
```

The `ProcessList` `DelegationRuntime` publishes the measurements in the `ProcessList` `Topic`, which is defined in Listing 5.2 below[1].

Listing 5.2: *Definition of* `ProcessList` `Topic`

```
module DDSMonitoring{
  struct ProcessList{
    hostName host;
    running_processes processes;
  };
  #pragma keylist ProcessList host
};
```

The `Topic` is published with the same QoS-policy as the `CPULoad` `Topic`, except that the `DURABILITY` QoS-policy is set to `TRANSIENT`, allowing a late joining monitoring application to retrieve the listings.

[1]See Listing A.5 in Appendix A for the type definition of `running_processes`

Intra-`DelegationRuntime` An intra-`DelegationRuntime` control loop could be introduced if the monitoring system — that first consisted solely of the `CPULoadBasic` `DelegationRuntime` — would be extended with the `ProcessList` `DelegationRuntime`. In that case, the `ProcessList` `DelegationRuntime` takes a subscription on the `Topic` published by the `CPULoadBasic` `DelegationRuntime` with the default `DataReader` QoS-policy[1], except with the `HISTORY-depth` set to 10. This way, the last ten samples will be available for analysis by the `DelegationRuntime`. If the average of these values gets higher than a specified threshold, the `ProcessList` `DelegationRuntime` can start publishing the process-list periodically, until the load drops below the threshold again. The *hot-standby* capability of OpenSplice DDS™ (suspend/resume) could also be used here, even allowing measurements from before reaching the threshold to be published in the system.

Inter-`DelegationRuntime` A control loop that controls across a `DelegationRuntime` boundary can in this simple example be created by extending the `CPULoadBasic` `DelegationRuntime` with control over the `ProcessList` `DelegationRuntime` by letting it publish control-`Topic`s. The pseudocode for this solution is listed in Listings 5.2 and 5.3 on the next page.

5.3 Conclusion

With the proposed architecture, distributed monitoring approach and the application of the publish/subscribe paradigm and DDS provided QoS-policies, it is possible to design a network management solution that is highly flexible, extendible and scalable. This creates a foundation for network managing experts to extend, allowing specialised network management tools, autonomous network management components and visualisations to be created, aiding in gaining insight about specific deployments and typical usage patterns of DDS systems.

[1]See Appendix B (DDS Specification)

Program 5.2: `CPULoadExtended` *pseudocode*

```
procedure CPULoadExtended($t$,$h$,$f$,$p$)
    $host \leftarrow$ getHostName()
    $A \leftarrow$ getAvailableCPUs()
    $n \leftarrow |A|$
    $s \leftarrow$ getCurrentTime()
    $i \leftarrow 0, avgLoad \leftarrow 0, L \leftarrow 0$
    repeat
        $U \leftarrow$ getCPUUsage()
        $avgLoad \leftarrow avgLoad - \left(\frac{L_i}{h}\right)$
        $L_i \leftarrow round\left(average\left(U\right) \times 100\right)$
        $avgLoad \leftarrow avgLoad + \left(\frac{L_i}{h}\right)$
        if $avgLoad \geq t$ then                              ▷ Threshold $t$ exceeded
            Publish($ProcessList$, $start$, $host$)  ▷ Start ProcessList on $host$
        else
            Publish($ProcessList$, $stop$, $host$)   ▷ Stop ProcessList on $host$
        end if
        $m \leftarrow (host, L_i)$
        Publish($m$,$p$)
        $i \leftarrow (i + 1)$ mod $h$
        $s \leftarrow s + \frac{1}{f}$
        sleepUntil($s$)
    until stopped by hosting DelegationAgent
end procedure
```

Program 5.3: `ProcessList` *pseudocode*

```
procedure ProcessList($f$,$p$)
    $host \leftarrow$ getHostName()
    repeat
        $P \leftarrow$ getRunningProcesses()  ▷ Get process-list from abstraction layer
        $m \leftarrow (host, P)$
        Publish($m$,$p$)
        $s \leftarrow s + \frac{1}{f}$
        sleepUntil($s$)                  ▷ Publish measurement $m$ in partition $p$
    until stopped by hosting DelegationAgent
end procedure
```

Chapter 6

Conclusions

In section 1.8 (Research Questions) at the beginning of this thesis, five research questions were set out to be answered. In this chapter the findings to the research questions will be discussed. Furthermore new interesting questions that arose while answering the research questions will be listed. Some of those have led to recommendations for further research on related subjects.

6.1 Findings to Research Questions

The findings to research question I can be best discussed after discussion of the other research questions. Research question I is discussed in section 6.1.5.

6.1.1 Findings to Research Question II

Research question II — *Which network management and visualisation of management data is needed in DDS-based systems?*

Large DDS-based systems encounter problems during start-up as well as during every-day's operation. Problems may stem from a relatively wide variety of sources. These sources can be roughly grouped into:

- physical limitations and/or imperfections
- logical errors, configuration issues, overload and the like

- node performance limitations, *e.g.*, limited functioning of hardware components

Although caused by a wide variety of particular sources or combinations of sources, the induced symptom is generally of the singular type: a drop in overall system performance. It is therefore almost impossible to diagnose the system on the basis of the symptom. Currently available diagnostic tools for traditional network environments are based on physical connection oriented protocols. They are not suitable to relate a failure or drop in functionality of the "connectionless" DDS-system to a location in the connection oriented hardware.

A DDS-system in its operational state is — as is typical for middleware — a black box delivering data at the right time at the right spot. Fascinating and attractive as that may be, the dark side pops up when a problem occurs. The only and most sensible thing that can be said of the black box in the latest mentioned state is that it does not function. It is therefore a must that diagnostic tools and "view glasses" be developed in order that light can be shed inside the black box. The many relations between the DDS-based system and the environment in which it is deployed must be made viewable.

Having discussed this topic within and outside the DDS-world, it appeared that many people have difficulty in getting a clear picture of the situation. I therefore tried to find some parallels in the known world to help understand the situation. The first one is the traffic parallel: Assume smoothly running traffic on the Dutch vehicle roads to start with. Then a symptom is observed where vehicles stop to arrive at Schiphol, Den Bosch, Gouda and other main cities. Just for arguments sake we list a number of possible causes:

- Accident on Oudenrijn junction — easy to find
- Beautiful weather and holiday in Germany — will not be found when inspecting the road status
- Double load of freight lorries on the A12 due to low water level in the main rivers — will not be found

It will be clear that a diagnostic tools dedicated to the hardware connection (road cameras, induction loops) are insufficient to properly diagnose the problem. What is needed is a kind of compound eye, that surveys node behaviour as well as connection behaviour.

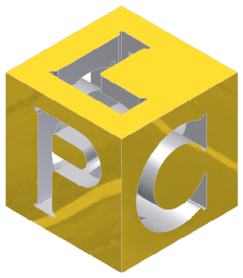
Sub-Pictures Nature has given us a good example of how to run a compound eye, looking into all directions. Several systems are applied by nature where signals of more than one source are compounded to sub-pictures. The use of compounding data, *i.e.*, the use of sub-pictures, greatly reduces the load on insect brains and nerves. The sub-pictures can be combined to get the complete overview picture. A similar approach is proposed to monitor the large multitude of nodes, processes, connections etc., that interfere with the operation of a DDS system.

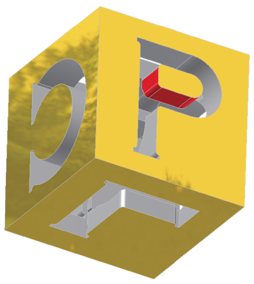
Perforated Cube Another parallel that may shed some light on the really complex nature of a DDS system is that of the multiple perforated cube. Assume the black box of a DDS system to be a golden cube. Seen from one side, you can see through the system, as it is *Logical* — and clear, Figure 6.1(a). Looking to another side, things appear to be as clear as possible, as everything boils down to *Processes* — or at least may seem to do so, Figure 6.1(b). Looking at yet another side may give the impression that you can see through the system as long as you concentrate on *Connections*, seen in Figure 6.1(a).

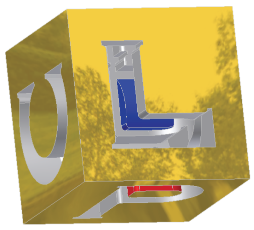
(a) *Logical face* **(b)** *Process face* **(c)** *Connections face*

Figure 6.1: *Three faces of a perforated cube*

A more realistic picture not only shows that all three dimensions are equally true, but also that they interact, showing large grey areas inside the still golden cube (Figure 6.2(a)). It will be a real challenge to get a view on the Red spot (Figure 6.2(b)), which may be literally floating. Or a Blue spot (Figure 6.2(c)) both floating and deeply hidden in the grey inside of the golden black box. It would be more realistic to represent a large DDS system by a multitude of more or less similar three-dimensional cubes, like in Figure 6.2(d).


(a) *Three faces combined*


(b) *Red issue*


(c) *Blue issue*

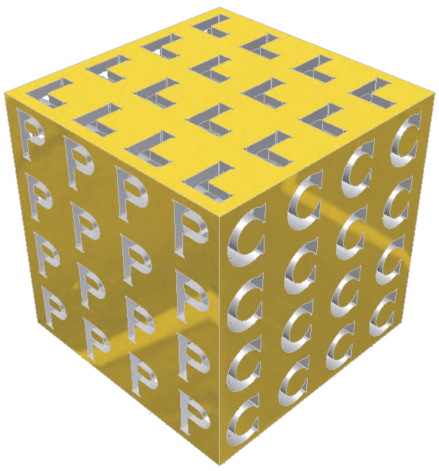

(d) *Multiple similar cubes forming a complex whole*

Figure 6.2: *Multiple faces of 3D cube*

To honour the greater complexity — the greater number of degrees of freedom or dimensions — another illustrative view is represented by a multi-faceted body in Figure 6.3. The dazzling number of colourful light scatterings and facets is maybe also a useful representation paying tribute to the fluidity of the light (information) flowing through it and the many facets or connections to both the inside and outside world, while still being brilliant in nature and function.

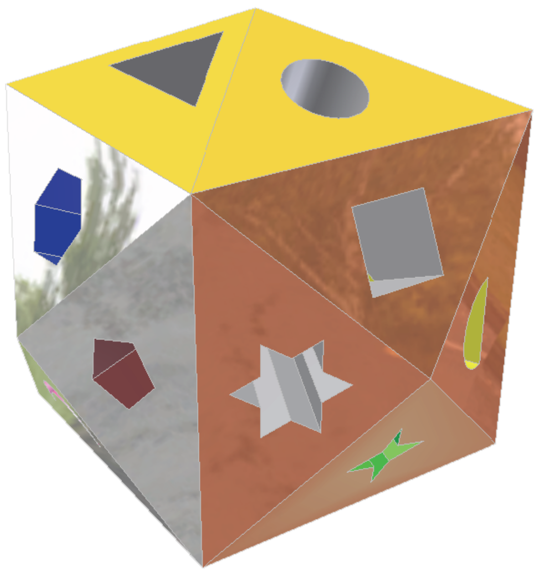


Figure 6.3: *Multi-faceted body*

6.1.2 Findings to Research question III

Research question III — *What functionality is needed for management tasks in* DDS *systems?*

The requirements on the main functionalities needed for network monitoring in DDS systems are twofold.

Scalable First of all the system needs to be *scalable*. The scalability requirement is almost synonymous with its direct consequence, namely the requirement that the monitoring be performed in a distributed manner.

Flexible Secondly, the system needs to be *flexible*. The flexibility requirement means that the monitoring system must have the possibility to select data to be monitored, the selection being dependable on momentary monitoring interests. It is for example essential that the data flow is not contaminated with a consistent flow of repeated, constant value OK-messages, emerging from a multitude of properly functioning nodes.

In the following sections some more detailed questions are answered with regard to the needed functionality of the system.

6.1.2.1 Findings to Sub-Research Question III.a

Sub-research question III.a — *In what areas is the OpenSplice DDS™ system currently lacking?*

The current system lacks the distribution of the data gathered through locally performed monitoring. Monitored data — locally acquired at a certain node — are currently kept at that node, thus disabling the possibility to acquire an overall view of the situation. There is no systematic way to locate a (potentially) troublesome node or cluster and no way to control the amount of monitoring performed at a remote system.

The current system also lacks the required flexibility. There is no possibility to request selected monitoring data from the distributed nodes nor control over what is monitored.

6.1.2.2 Findings to Sub-Research Question III.b

Sub-research question III.b — *How can data be collected in a scalable manner, suiting the environments in which DDS is typically deployed?*

It is essential for large systems that no bottle-necks be created by the monitoring itself. Filtering and reduction of monitoring data should therefore be performed to the largest possible — or at least optimum — extent at the source node. Data aggregation and compression should be done at the source and/or at distributed clusters, in order to satisfy the requirement.

The newly introduced *dynamic delegated monitoring* concept allows for extreme flexibility and extendibility within the scope of DDS. The framework naturally suits the DDS environment and solves many of the issues encountered in traditional network management solutions, by using the facilities available in DDS itself. The primarily in-band solution supports the potential deployment of (fully) autonomous monitoring and/or management routines.

The proposed solution doesn't require an on-line central entity to define what is monitored and where it is monitored. Monitoring priority or focus can simply be escalated based on pre-defined logic or built-in trigger/control loops. The implementation of such a (layered) control loop can thus be used to intensify or otherwise alter the local monitoring activities, including the publication properties, *e.g.*, where the data is publised and when/if the data is published, of the (preferably compressed) monitoring data. This approach prevents late signalisation of important monitoring data, while keeping monitoring data transfer load as low as reasonably possible. It furthermore allows for monitoring patterns not generally available, *e.g.*, monitoring of interactions in which the node at which the monitoring occurs does not participate. This functionaliy is supported by the fully distributed data gathering that is supported by the proposed solution.

6.1.2.3 Findings to Sub-Research Question III.c

Sub-research question III.c — *Which data and/or metrics are needed for management of DDS systems?*

It is possible to define a minimal data set that should always be available. This minimal data set includes number of CPUS, (compounded) CPU load, available storage capacity and similar basic data. The semantics of this set needs to be defined within the scope of DDS, allowing general interpretability of the data, *i.e.*, abstracted from the underlying operating system or hardware. However, system experts managing large scale DDS systems have widely varying data interests. It must therefore be facilitated that the monitoring system manager can collect a wide range of specific monitoring data on request.

6.1.3 Findings to Research Question IV

Research question IV — *Is visualisation in a network management solution for DDS a luxury or a necessity?*

Given the ability of the human visual system to quickly analyse graphical information and the inability to quickly process textual data leads inevitably to the need to present relevant information in a graphic form. Data visualisation can relatively straightforward be applied and any tooling of framework supporting or easing the creation of data visualisations may be of worth as-is already.

As discussed in section 4.3 (Visualising Dataflow), where the power of even a very basic application of information visualisation is illustrated, visualisations not only are a different representation of numbers. Visualisations can (implicitly) add domain-knowledge or decorate basic data, allowing information to be communicated that was not explicitly available in the raw data or aggregations thereof. In a stage where OpenSplice DDS™ is still found to be evolving continually into new markets, new deployment kinds and new deployment scales, having access to visualisations providing this information is crucial in gaining understanding and insights about the system. Eventually the gained insights may render the need for visualisations redundant. For example, when meaningful dimensionless quantities or similar descriptors are found — like for example the Reynolds number, Rockwell scale or Strain — that can describe or predict system behaviour, the same information of a complex visualisations may be captured in such a quantity. Until then, all methods at hand — of which information visualisation is expected to be an important one — that can increase the amount of insight should be carefully looked into in order to achieve the full potential of both the deployed systems as well as the involved network management related tasks.

6.1.4 Findings to Research Question V

Research question V — *How can a network management implementation in OpenSplice DDS™ be designed?*

The design of a network management system for OpenSplice DDS™ must meet the functional requirements set out in the answers to the previously discussed research questions.

The facilities that a DDS system offers make it possible to design an *in-band* solution that utilises the DDS system for the communication of management data. Although there may be exceptional cases where *out-of-band* communication can be preferable, it is most attractive to use the already available in-house facilities of the DDS system to be monitored. This is especially true when the network management system requires only marginal system bandwidth, as is the case with the proposed system where management data are compounded, aggregated and compressed, before being released into the DDS system traffic and are distributedly gathered.

The solution proposed and discussed in chapter 5 (Design) — built on the dynamic "delegated monitoring"-concept introduced in section 3.3.3 (Distributed Monitoring) — is very flexible and supports scalable, fully decentral, dynamically extendible and eventually autonomous network monitoring and management. The proposed solution essentially comprises:

- Platform-Abstraction definition and API
- `DelegationAgent` `Topics` and semantics definition
- `DelegationRuntime` `Topics` and semantics definition
- Delegation API

The functional composition of the above results in the desired flexibility. The network management system can be extended while in operation, due to its data-centric, connectionless design; `DelegationRuntimes` themselves are also distributed by means of `Topics`.

As `DelegationRuntime` can also be supplied with parameters, their behaviour can be easily modified while using only minimal communication overhead.

6.1.5 Findings to Research Question I

Finally, the first and main research question can be answered.

Research question I — *How can human experts be optimally supported with their management tasks in DDS-based systems?*

In order to satisfy the information requests of system managing experts, a monitoring system must be incredibly extensive, or simply extendible. As not all future information request types and search patterns can be predicted, it is sensible to facilitate the experts with a provision that is intrinsically so flexible that it does not pose a limit by itself. It must at the same time be prevented that the provision is flexibility pur sang, with no predefined basic tools and contents, as that might well act as a threshold, that user must "overcome" before entering its use. That means that a basic set of functionalities, covering the most common diagnostic information, must be available from the beginning.

The system can then — as it is being used by the experts — further evolve on the basis of gathered experience and newly emerging expert demands.

It would be best to enable start-up of the management system at any convenient time — in other words — without switching the system into for example a diagnose modus. The typical DDS systems are so huge, that otherwise it would be (almost) impossible to establish the cause of a problem when it presents. Certain local monitoring tasks could run in standby modus and be triggered by predefined events. Experts could also initiate tests on a live system.

To date, expert developed diagnostic tools have mainly focused on node-local analysis, because system-wide analyses would be too complex to set up. By creating essentially better monitoring facilities, that principally reach beyond the boundaries of individual nodes, the proposed network management solution or framework can serve as an enabler for system-wide expert analyses.

6.2 Recommendations for Further Research

Throughout the thesis several topics were identified and discussed that would benefit greatly from or require further research. In this section a few topics will be highlighted.

6.2.1 Logical Network Visualisation

One of the aspects that makes management of DDS systems difficult is that fact that both the physical network and the logical network need to monitored, but their relation is not clearly defined. The logical network is formed by matched publications and subscriptions, as explained in section 4.3 (Visualising Dataflow). Possible similarities of the logical network to physical networks or other logical overlay networks may allow reuse of tools or visualisations that exist for those areas within the management of DDS systems. A lot can at least be learnt from the difficulties encountered while creating those tools. It may furthermore be possible to transfer those tools to allow management of the logical network.

For example the visualisation techniques described by Holten (2009) could prove to be of great value when visualising large-scale topologies. The dataflow graphs that can be constructed define dependencies between `Topics`, which could be displayed by using *hierarchical edge bundles* (HEB) or similar edge-clutter reduction technique. Ellis & Dix (2007) provide a general taxonomy of techniques for reducing visual clutter. HEB provide means to display node–link diagrams containing a large number of nodes and edges, without the visual clutter. Even more interesting would be to research whether techniques like *force-directed edge bundles* (FDEB) — a self-organising edge-bundling technique that doesn't require hierarchy in its input — or *geometry-based edge bundling* (GBEB) — an edge-bundling technique that depends on a control mesh to guide the bundling — can be applied in order to visualise the relation between the physical and the overlay network. The physical network topology could provide for the positioning of the nodes or as input for the control mesh. In Figure 6.4 an example application of FDEB is shown applied on a migration graph.

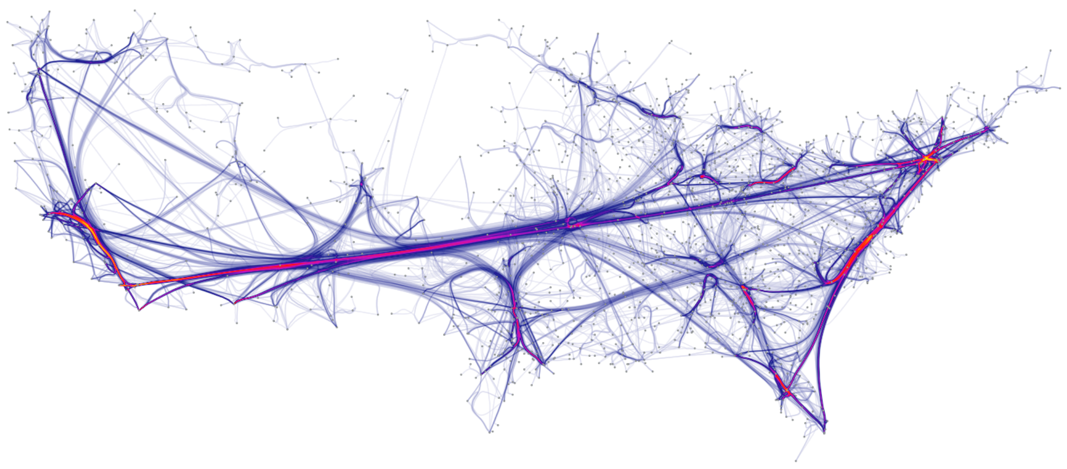


Figure 6.4: *Force-directed edge bundles example showing a U.S. migration graph consisting of 1 715 nodes and 9 780 edges*
Source: *Holten & Van Wijk (2009, p. 988, Figure 9b)*

6.2.2 Visualisation Evaluation

When the visualisation capabilities of a network management solution for DDS evolve, the need for verification or comparison of effectiveness will emerge. Newly created visualisations will have to be verified against the goal of the visualisations, *i.e.*, *insight*. Verification of effectiveness is far from trivial, and a literature study should be performed to see whether applicable verification techniques exist.

6.2.3 Security

The distribution of `DelegationRuntimes` allows for extremely flexible network management solutions to be deployed. This however also brings some security implications. The `DelegationAgents` are capable of running all executable code that is distributed, so some way of sender-verification will have to be applied in order to prevent a misbehaving application to distribute dangerous executables. OpenSplice DDS™ provides quite an extensive set of security features at different levels and the applicability of that functionality would need to be studied in the context of the delegated monitoring framework.

It may also be possible to attain enough security by choosing the appropriate executable format in which the `DelegationRuntimes` are distributed. For example signed executable packages may already provide enough safety, so in that case support for executable signing may become a requirement for the chosen executable format.

References

Al-Shaer, E., Greenberg, A., Kalmanek, C., Maltz, D.A., Ng, T.S.E. & Xie, G.G. (2009). New frontiers in internet network management. *SIGCOMM Comput. Commun. Rev.*, **39**, 37–39. doi:10.1145/1629607.1629615. 30, 31

Barbosa, P. & Granville, L. (2010). Interactive SNMP traffic analysis through information visualization. In *Network Operations and Management Symposium*, 73–79, IEEE. doi:10.1109/NOMS.2010.5488438. 29, 30

Birman, K. & Joseph, T. (1987). Exploiting virtual synchrony in distributed systems. *SIGOPS Operating Systems Review*, **21**, 123–138. doi:10.1145/37499.37515. 11

Boasson, M. (1993). Control systems software. In *Fifth Euromicro Workshop on Real-Time Systems*, 156–161, IEEE Computer Society Press, Ouiu, Finland. doi:10.1109/9.231463. 4, 18

Boasson, M. (1994). System for dynamic communication among subsystems implementinig data requests by data type and locating data by data type matching, U.S. Pat. 5 301 339, apr 5, 1994. 11

Boasson, M. (2002). Embedded systems unsuitable for object orientation. In J. Blieberger & A. Strohmeier, eds., *Reliable Software Technologies — Ada-Europe 2002*, vol. 2361 of *Lecture Notes in Computer Science*, 53–80, Springer Verlag Berlin / Heidelberg. doi:10.1007/3-540-48046-3_1. 4

Card, S., Mackinlay, J. & Shneiderman, B. (1999). *Readings in information visualization: using vision to think*. Morgan Kaufmann. 43

Chen, C. (2010). Information visualization. *Wiley Interdisciplinary Reviews: Computational Statistics*, **2**, 387–403. doi:10.1002/wics.89. 42, 45

Cugola, G., Picco, G. & Murphy, A. (2003). Towards dynamic reconfiguration of distributed publish-subscribe middleware. In A. Coen-Porisini & A. van der Hoek, eds., *Software Engineering and Middleware*, vol. 2596 of *Lecture Notes in Computer Science*, 187–202, Springer Verlag Berlin / Heidelberg. doi:10.1.1.13.911. 1

Cui, W., Zhou, H., Qu, H., Wong, P.C. & Li, X. (2008). Geometry-based edge clustering for graph visualization. *Visualization and Computer Graphics, IEEE Transactions on*, **14**, 1277–1284. doi:10.1109/TVCG.2008.135. xiii

Dobrev, P., Stancu-Mara, S. & Schönwälder., J. (2009). Visualization of node interaction dynamics in network traces. In R. Sadre & A. Pras, eds., *Scalability of Networks and Services*, vol. 5637 of *Lecture Notes in Computer Science*, 147–160, Springer Verlag Berlin / Heidelberg. doi:10.1007/978-3-642-02627-0_12. 29, 45

Ellis, G. & Dix, A. (2007). A taxonomy of clutter reduction for information visualisation. *Visualization and Computer Graphics, IEEE Transactions on*, **13**, 1216–1223. doi:10.1109/TVCG.2007.70535. 73

Eugster, P.T., Felber, P.A., Guerraoui, R. & Kermarrec, A.M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, **35**, 114–131. doi:10.1145/857076.857078. 1

Friendly, M. & Denis, D.J. (2001). Milestones in the history of thematic cartography, statistical graphics, and data visualization [online, last checked Dec. 9, 2010]. 41, 42

Holten, D. & Van Wijk, J. (2009). Force-directed edge bundling for graph visualization. In *Computer Graphics Forum*, vol. 28, 983–990, Wiley Online Library. doi:10.1111/j.1467-8659.2009.01450.x. xiii, 73

Holten, D.H. (2006). Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. In *Proceedings of Vis/InfoVis 2006*, vol. 12, 741–748, IEEE Computer Society. doi:10.1109/TVCG.2006.147. xiv, 45, 46, 48

Holten, D.H. (2009). *Visualization of graphs and trees for software analysis*. Ph.D. thesis, Eindhoven University of Technology. xiii, 45, 73

Horst, R., Pardalos, P.M. & Thoai, N.V. (2000). *Introduction to Global Optimization*. Nonconvex Optimization and Its Applications, Kluwer Academic Publishers, 2nd edn. 6

Keim, D., Mansmann, F., Schneidewind, J. & Schreck, T. (2006). Monitoring network traffic with radial traffic analyzer. *Symposium On Visual Analytics Science And Technology*, **0**, 123–128. doi:10.1109/VAST.2006.261438. 29

Kraemer, E.T. & Stasko, J.T. (1993). The visualization of parallel systems: An overview. *Journal of Parallel and Distributed Computing*, **18**, 105–117. doi:10.1006/jpdc.1993.1050. 1

Mansmann, F. & Vinnik, S. (2006). Interactive exploration of data traffic with hierarchical network maps. *IEEE Transactions on Visualization and Computer Graphics*, **12**, 1440–1449. doi:10.1109/TVCG.2006.98. 29

Neumaier, A. (1995). Introduction to global optimization [online, last checked Jul. 30, 2011]. 6

Neumaier, A. (2004). Global optimization and constraint satisfaction. *Acta Numerica*, **13**, 271–369. doi:10.1.1.3.944. 6

North, C. (2006). Toward measuring visualization insight. *IEEE Computer Graphics and Applications*, **26**, 6–9. doi:10.1109/MCG.2006.70. 43, 44

OMG (2007). Data Distribution Service for Real-time Systems, rev. 1.2. Doc. nr. *formal/2007-01-01*, Object Management Group (OMG), Data Distribution Services PSIG (DDSIG). 13, 14, 15, 16, B-1

Plaisant, C., Fekete, J.D. & Grinstein, G. (2008). Promoting insight-based evaluation of visualizations: From contest to benchmark repository. *Visualization and Computer Graphics, IEEE Transactions on*, **14**, 120 –134. doi:10.1109/TVCG.2007.70412. 43

Pras, A., Schönwälder, J., Burgess, M., Festor, O., Pérez, G.M., Stadler, R. & Stiller, B. (2007). Key research challenges in network management. *Communications Magazine, IEEE*, **45**, 104–110. doi:10.1109/MCOM.2007.4342832. 29, 30, 31, 36

Reed, D., Shields, K., Scullin, W., Tavera, L. & Elford, C. (1995). Virtual reality and parallel systems performance analysis. *Computer*, 28, 57–67. doi:10.1109/2.471180. 1

Salvador, E. & Granville, L. (2008). Using visualization techniques for SNMP traffic analyses. In *Computers and Communications*, 806–811, IEEE Computer Society. doi:10.1109/ISCC.2008.4625672. 45

Schönwälder, J. (2008). Simple Network Management Protocol (SNMP) traffic measurements and trace exchange formats. Informational, IRTF/NMRG [last checked Nov. 26, 2010]. 30

Schönwälder, J., Fouquet, M., Rodosek, G.D. & Hochstatter, I.C. (2009). Future internet = content + services + management. *Comm. Mag.*, 47, 27–33. doi:10.1109/MCOM.2009.5183469. 30

Treinish, L. (1998). Task-specific visualization design: a case study in operational weather forecasting. In *Visualization '98. Proceedings*, 405–409. doi:10.1109/VISUAL.1998.745330. 49

van Wijk, J. (2005). The value of visualization. In C. Silva, E. Groeller & H. Rushmeier, eds., *IEEE Visualization*, 79–86. doi:10.1109/VISUAL.2005.1532781. 29

Weise, T. (2009). *Global Optimization Algorithms — Theory and Application*. Self-Published, 2nd edn. [last checked Jul. 30, 2011]. 6

Wikipedia (2004a). Wikipedia: Lingua franca [online, last checked Sep. 7, 2010]. xvii

Wikipedia (2004b). Wikipedia: Publish/subscribe [online, last checked Aug. 17, 2010]. 11

Yi, J.S., Kang, Y.A., Stasko, J.T. & Jacko, J.A. (2008). Understanding and characterizing insights: how do people gain insights using information visualization? In *Proceedings of the 2008 conference on BEyond time and errors: novel evaLuation methods for Information Visualization*, BELIV '08, 4:1–4:6, ACM, New York, NY, USA. doi:10.1145/1377966.1377971. 43

Appendix A

IDL Specifications

A.1 Platform Abstraction

In this appendix some possible definitions for the data returned by the platform abstraction API are given. The implementations details are given in *Common Object Request Broker Architecture* (CORBA) IDL.

Listing A.1: *Type definition for* ***available_cpus*** *metric*

```
typedef unsigned short identifier;
typedef sequence<identifier> available_cpus;
```

Listing A.2: *Type definition for* ***cpu_usage*** *metric*

```
typedef unsigned short identifier;
typedef float usage;

struct cpuUsage{
  identifier id;
  usage usage;
};

typedef sequence<cpuUsage> cpu_usage;
```

Listing A.3: *Type definition for* `memory_stats` *metric*

```
typedef unsigned long memory;

struct memory_stats {
  memory total;
  memory used;
  memory cached;
};
```

Listing A.4: *Type definition for* `persistent_storage_stats` *metric*

```
typedef memory_stats persistent_storage_stats;
```

Obviously, if practice reveals that `persistent_storage_stats` requires a different definition than `memory_stats`[1], the above definition will not be viable and it will need to be appropriately redefined.

Listing A.5: *Type definition for* `running_processes`

```
typedef unsigned short identifier;
typedef float usage;
typedef unsigned long memory;

struct process_stats {
  identifier id;
  string name;
  usage usage;
  memory memUsage;
};

typedef sequence<process_stats> running_processes;
```

[1]Defined in Listing A.3 on this page

A.2 Delegation Module Pseudo-IDL

The code listing below is an example specification for the interfaces of the delegation module in IDL and is illustrative[1].

Listing A.6: *Pseudo-IDL specification of delegation-module*

```
module Delegation{
  typedef time_t time; // Timestamp storage
  interface DelegationRuntime;
  typedef string DelegationRuntimeName;
  enum DelegationRuntimeState {
    LOADING,    // DR is loading into the DA
    LOADED,     // DR is loaded into the DA and can be started
    STARTING,   // DR is starting
    RUNNING,    // DR is started and running
    IDLE,       // DR is started, but is idling
    PAUSED,     // DR is started, but is paused by the DA
    STOPPING    // DR is stopping
  };

  struct DelegationRuntimeStatus {
    DelegationRuntimeName name;
    DelegationRuntimeState state;
    time timestamp;
  };

  typedef
    sequence<DelegationRuntimeStatus> DelegationRuntimeStatusSeq;
  typedef
    sequence<Param> ParamSeq; // Sequence of parameters

  // Creation of a Delegationentities is not defined, but could
  // follow pattern analogue to the creation of a
  // DomainParticipant in DDS.
```

[1]Do not expect to be able to compile the IDL

```
  interface DelegationAgent{
    // Loads the specified DelegationRuntime and binds it to the
    // specified name.
    retCode_t
    delegationRuntimeLoad(
      in DelegationRuntimeName drName,
      in DelegationRuntime dr);

    // Starts the DelegationRuntime identified by the specified
    // name passing the specified parametes.
    retCode_t
    delegationRuntimeStart(
      in DelegationRuntimeName drName,
      in ParamSeq params);

    // Pauses the DelegationRuntime identified by the specified
    // name. For example hotstandby functionality can be tied
    // to the PAUSED state.
    retCode_t
    delegationRuntimePause(
      in DelegationRuntimeName drName);

    // Resumes the DelegationRuntime identified by the specified
    // name from the PAUSED state.
    retCode_t
    delegationRuntimeResume(
      in DelegationRuntimeName drName);

    // Stops the DelegationRuntime identified by the specified
    // name.
    retCode_t
    delegationRuntimeStop(
      in DelegationRuntimeName drName);

    // Unloads the DelegationRuntime identified by the specified
```

```
      // name.
      retCode_t
      delegationRuntimeUnload(
        in DelegationRuntimeName drName);

      // Retrieves the current status of the DelegationRuntime
      // identified by the specified name.
      DelegationRuntimeStatus
      delegationRuntimeGetStatus(
        in DelegationRuntimeName drName);

      // Enumerates the status of all loaded DelegationRuntimes.
      DelegationRuntimeStatusSeq
      delegationRuntimeGetStatusAll();
    };

    interface DelegationRuntime{
      // Starts the DelegationRuntime with the specified parameters
      retCode_t start(in ParamSeq params);

      // Pauses the DelegationRuntime
      retCode_t pause();

      // Resumes the DelegationRuntime
      retCode_t resume();

      // Stops the DelegationRuntime
      retCode_t stop();

      // Returns the status of the DelegationRuntime
      DelegationRuntimeStatus getStatus();
    };
  };
```

Appendix B

DDS Specification

B.1 QoS-policies

Table B.1 lists the default QoS-policies for DDS entities, as listed in version 1.2 of the DDS specification, section 7.1.3. For a more in-depth explanation of the QoS-policies and the defaults, additional information regarding RxO, changeability and semantics of the policies, please consult the specification by the OMG (2007).

Table B.1: *Default QoS-policies for DDS-entities*

QoS-policy	Meaning	Concerns
`USER_DATA`	User data, distributed by means of built-in topics. The default value is an empty (zero-sized) sequence.	`Domain, Participant, DataReader, DataWriter`
`TOPIC_DATA`	User data, distributed by means of built-in topics. The default value is an empty (zero-sized) sequence.	`Topic`
`GROUP_DATA`	User data, distributed by means of built-in topics. The default value is an empty (zero-sized) sequence.	`Publisher, Subscriber`

Continued on next page

Table B.1: *(continued from previous page)*

QoS-policy	Meaning	Concerns
DURABILITY	Expresses whether data should be maintained for late-joiners. Possible values are VOLATILE[1], TRANSIENT_LOCAL, TRANSIENT and PERSISTENT.	Topic, DataReader, DataWriter
DURABILITY_-SERVICE	Specifies the configuration of the durability service, which implements the TRANSIENT and PERSISTENT DURABILITY kinds. It specifies resource-limits, history and a cleanup-delay.	Topic, DataWriter
PRESENTATION	Specifies how samples of an instance are presented to the application, and allows for specification of the largest scope over which an application is able to preserve order and coherency of changes.	Publisher, Subscriber
PRESENTATION	Specifies how samples of an instance are presented to the application, and allows for specification of the largest scope (INSTANCE, TOPIC, GROUP)[2] over which an application is able to preserve order and coherency of changes.	Publisher, Subscriber
DEADLINE	The period[3] to which a DataWriter commits to update the value of each instance and at which a DataReader expects a new sample.	Topic, DataReader, DataWriter

Continued on next page

[1]This is the default
[2]The default is INSTANCE
[3]The default is INFINITE

Table B.1: *(continued from previous page)*

QoS-policy	Meaning	Concerns
LATENCY_-BUDGET	The maximum delay[1] from the write-time until the notification of the receiving application that the data is delivered.	Topic, DataReader, DataWriter
OWNERSHIP	Specifies whether it is allowed for multiple DataWriters to write the same instance, and if so, how these modifications should be arbitrated. Two kinds of ownership are possible: SHARED and EXCLUSIVE[2].	Topic, DataReader, DataWriter
OWNERSHIP_-STRENGTH	Explicitly specifies the value used to arbitrate the strength if the OWNERSHIP QoS-policy is of kind EXCLUSIVE.	DataWriter
LIVELINESS	Determines the mechanism (manual or automatic) and parameters used by the application to determine or assert whether an Entity is alive. The *liveliness* of an Entity is used to maintain instance ownership in combination with the setting of the OWNERSHIP QoS-policy. The application can also be informed via a listener when an Entity is no longer alive.	Topic, DataReader, DataWriter
TIME_-BASED_-FILTER	Filter allowing a DataReader to express interest in a (potentially) subset of the data by expressing a minimum separation between two changes.	DataReader
PARTITION	Set of strings that introduces a logical partition among the Topics visible by the Publisher and Subscriber.	Publisher, Subscriber

Continued on next page

[1]The default is 0; as a hint for the middleware to minimise the delay
[2]The default is SHARED

Table B.1: *(continued from previous page)*

QoS-policy	Meaning	Concerns
`RELIABILITY`	Indicates the level of reliability offered/ requested by the middleware and a maximum time an operation may block trying to guarantee the reliability. The kinds `RELIABLE`[1] and `BEST_EFFORT`[2] are defined.	`Topic`, `DataReader`, `DataWriter`
`TRANSPORT_PRIORITY`	Acts as a hint to the underlying infrastructure on how to prioritize the data.	`Topic`, `DataWriter`
`LIFESPAN`	Specifies the validity[3] of the data written by the `DataWriter`.	`Topic`, `DataWriter`
`DESTINATION_ORDER`	Controls the criterion[4] used to determine the logical order among changes made by a `Publisher`, allowing for `BY_RECEPTION_TIMESTAMP` and `BY_SOURCE_TIMESTAMP`.	`Topic`, `DataReader`, `DataWriter`
`HISTORY`	Specifies the amount of history the middleware must retain for an instance in case an instance changes before the last change was successfully communicated. Possible options are `KEEP_ALL` and `KEEP_LAST`[5] with a depth parameter controlling the amount of *last* samples to attain.	`Topic`, `DataReader`, `DataWriter`

Continued on next page

[1]This is the default for `DataWriters`
[2]This is the default for `DataReaders`
[3]The default is `INFINITE`
[4]The default is `BY_RECEPTION_TIMESTAMP`
[5]The default is `KEEP_LAST` with a depth of 1

Table B.1: *(continued from previous page)*

QoS-policy	Meaning	Concerns
RESOURCE_LIMITS	Specifies the resources that the middleware may consume in order to satisfy the requested QoS-policy, by specifying the upper limits[1] max_samples, max_instances and max_samples_per_instance.	Topic, DataReader, DataWriter
ENTITY_FACTORY	Specifies whether an Entity acting as a factory for other Entitys creates new Entitys in an *enabled* state.	DomainParticipantFactory, DomainParticipant, Publisher, Subscriber
WRITER_DATA_LIFECYCLE	Specifies how a DataWriter automatically manages the lifecycle of instances.	DataWriter
READER_DATA_LIFECYCLE	Specifies how a DataReader automatically manages the lifecycle of instances by means of two durations.	DataReader

[1]By default, all limits are UNLIMITED