Design and Run-Time Quality of Service Management Techniques for Publish/Subscribe Distributed Real-time and Embedded Systems

Joseph William Hoffert II
jhoffert@dre.vanderbilt.edu
Electrical Engineering and Computer Science Dept.
Vanderbilt University, Nashville, TN, 37235

January 22, 2010
Contents

1 Introduction 1
   1.1 Emerging Technologies and Trends .......................... 1
   1.2 Challenges for QoS-enabled Pub/Sub Systems ............... 1
   1.3 Enhancing Developer Productivity and Run-time Support for QoS-enabled Publish-Subscribe Middleware ........ 3
   1.4 Proposal Organization ..................................... 4

2 Design Time Management of QoS Configuration Complexity 5
   2.1 Motivating Example: NASA’s Magnetospheric Multiscale Mission .......... 5
   2.2 Related Research ........................................... 7
   2.3 Unresolved Challenges ..................................... 8
   2.4 Solution Approach: Distributed QoS Modeling Language (DQML) 9
      2.4.1 Context for DQML .................................... 10
      2.4.2 Structure and Functionality of DQML ................ 17
      2.4.3 DQML Productivity Analysis for the MMS Case Study ........ 22
   2.5 Lessons Learned ........................................... 26

   3.1 Motivating Example: Search and Rescue (SAR) Operations for Disaster Recovery .................. 28
   3.2 Related Research ........................................... 31
   3.3 Unresolved Challenges ..................................... 32
   3.4 Solution Approach: FLEXible Middleware and Transports ....... 33
      3.4.1 The Structure and Functionality of FLEXMAT and ReLate2 33
      3.4.2 Experimental Setup, Results, and Analysis .............. 38
   3.5 Lessons Learned ........................................... 47

4 Autonomic Adaptation of QoS-enabled Pub/Sub Middleware 50
   4.1 Motivating Example: Ambient Assisted Living in Smart Cities .......... 50
   4.2 Related Research ........................................... 53
   4.3 Unresolved Challenges ..................................... 55
   4.4 Proposed Solution Approach: ADAptive Middleware and Network Transports ........... 56
# List of Figures

2.1 Example MMS Mission Scenario with QoS Requirements  
2.2 Incompatible MMS Ground Station and Spacecraft Deadline QoS  
2.3 Inconsistent QoS Policies for an MMS Ground Station  
2.4 Deadline QoS Policy Relationships (UML notation)  
2.5 Deadline QoS Policy Compatibility Constraints  
2.6 DDS Entities Supported in DQML  
2.7 Example of DQML QoS Policy Variability Management  
2.8 Example of DQML QoS Policy Compatibility Constraint Checking  
2.9 Example of DQML QoS Policy Consistency Constraint Checking  
2.10 QoS Policy Configuration File for Figure 2.7  
2.11 Metrics for Manual Configuration vs. DQML’s Interpreter  
3.1 Search and Rescue Motivating Example  
3.2 Uses of Infrared Scans during Disaster Recovery  
3.3 Uses of Video Stream during Disaster Recovery  
3.4 OpenDDS and its Transport Protocol Framework  
3.5 FEC Reliable Multicast Protocol - Sender-based  
3.6 FEC Reliable Multicast Protocol - Receiver-based (LEC)  
3.7 MPEG Frame Dependencies  
3.8 Emulab: 3 readers, 0% loss, 25Hz  
3.9 Emulab: 3 readers, 0% loss, 50Hz  
3.10 Emulab: 3 readers, 1% loss, 25Hz, no RMcast  
3.11 Emulab: 3 readers, 1% loss, 25Hz  
3.12 Emulab: 3 readers, 1% loss, 25Hz  
3.13 Emulab: 3 readers, 1% loss, 25Hz  
3.14 Emulab: 3 readers, 3% loss, 25Hz  
3.15 Emulab: 3 readers, 3% loss, 50Hz  
3.16 Baseline Emulab Experiment: Updates Received (3 readers, 3% loss, 50 Hz update rate)  
3.17 ReLate Metrics for Emulab Experiment: 3 readers, 3% loss, 50 Hz update rate  
3.18 ReLate2 Metrics for Emulab Experiment: 3 readers, 3% loss, 50 Hz update rate
3.19 ISISlab: 3 readers, 1% loss ......................... 45
3.20 ISISlab: 3 readers, 1% loss ......................... 45
3.21 ISISlab: 3 readers, 5% loss ......................... 45
3.22 ISISlab: 3 readers, 5% loss ......................... 45
3.23 ISISlab: 20 readers, 1% loss ....................... 46
3.24 ISISlab: 20 readers, 1% loss ....................... 46
3.25 ISISlab: 20 readers, 5% loss ....................... 46
3.26 ISISlab: 20 readers, 5% loss ....................... 46

4.1 Motivating Example for Smart City Ambient Assisted Living  51
4.2 Architecture for Proposed ADAMANT Solution Approach  57

5.1 Doctorial Research and Dissertation Timeline ................ 61
## List of Tables

2.1 MMS pub-sub QoS Policy Requirements .......................... 6  
2.2 DDS QoS Policies .................................................. 11  
2.3 Potential Incompatible DDS QoS Policies ...................... 15  
2.4 DDS QoS Consistency Constraints ............................... 16  
2.5 DDS QoS Policies for data writers .............................. 23  
2.6 DDS QoS Policies for data readers .............................. 24  

3.1 Transport Protocols Evaluated ................................. 36  
3.2 Emulab Variables ................................................... 39  
3.3 ISISlab Variables ................................................... 39  

5.1 Summary of Research Contributions ............................ 61
Abstract

The proliferation of publish/subscribe (pub/sub) middleware for distributed, real-time, and embedded (DRE) systems has increased dramatically in recent years due to the separation of concerns afforded by decoupling senders and receivers. Pub/sub middleware platforms often provide many configurable policies that affect end-to-end quality of service (QoS). These QoS policies are relevant for areas of concern such as fault tolerance, timeliness requirements, data reliability, and security.

Although the flexibility and functionality of pub/sub middleware platforms has increased, configuring their QoS policies in semantically compatible ways has become more complex. Developing a QoS configuration can be complicated not only by the number and type of parameters for a single policy, the number of policies available, and the policy interactions but also by the accidental complexity inherent in accurately transforming a QoS configuration design into appropriate implementation artifacts. Moreover, this same complexity is present whether an initial QoS configuration is being developed or an existing QoS configuration is being updated or modified.

Even when a system’s QoS configuration is correctly designed and transformed into implementation artifacts, implementation mechanisms for achieving specified QoS in one particular operating environment might not achieve specified QoS for a different operating environment. Moreover, implementation mechanisms to support diverse QoS requirements can adversely affect each other, e.g., mechanisms for reliability and mechanisms for low latency, which can increase the complexity of achieving specified QoS for a given operating environment. Furthermore, perturbations in the environment during run-time, e.g., increase in network latency or packet loss, can cause specified QoS not to be met. The initial environment in which the system was deployed and for which the specified QoS was maintained can change causing discontinuity in QoS.

This thesis resolves the challenges of developing QoS configurations and supporting run-time QoS in the following ways. First, it presents the Distributed QoS Modeling Language (DQML)—a domain specific modeling language for pub/sub DRE systems—that automates development, analysis, and synthesis of semantically compatible QoS policy configurations for pub/sub middleware. DQML eases the development of new QoS configurations as well as modifications to existing QoS configurations. Second, it presents FLEXible Middleware And Transports (FLEXMAT), which integrates and enhances standards-based pub/sub middleware with a network transport framework, along with composite QoS metrics to support quantitative performance analysis in determining the most appropriate QoS mechanisms for a given environment. FLEXMAT provides developers with an evaluation framework to determine appropriate middleware mechanisms in light of specified QoS. FLEXMAT also provides flexibility in the creation, selection, and use of transport protocols to support specified QoS.
Finally, this thesis proposes ADAPTive Middleware And Network Transports (ADAMANT) to enable autonomic adaptation of middleware mechanisms to maintain specified QoS in dynamic environments. ADAMANT proposes to use machine learning techniques trained on quantitative performance analysis to determine appropriate settings so that QoS is supported as changes in the environment occur. The hypothesis is that ADAMANT will be able to consistently improve QoS provided for a given QoS configuration over manual methods for dynamic environments. Furthermore, ADAMANT will be able to provide the timely adaption crucial for DRE systems.
Chapter 1

Introduction

1.1 Emerging Technologies and Trends

With increasing advantages of cost, performance, and scale over single computers, the proliferation of distributed systems in general and distributed event-based systems in particular have increased dramatically in recent years [1]. In contrast to distributed object computing middleware (such as CORBA and Java RMI)—where clients invoke point-to-point methods on distributed objects—publish/subscribe (pub/sub) middleware platforms distribute data from suppliers to (potentially multiple) consumers. Examples of standardized pub/sub middleware include the Java Message Service (JMS) [2], Web Services Notification [3], CORBA Event Service [4], and OMG Data Distribution Service (DDS) [5]. These event-based services allow the propagation of data throughout a system using an anonymous pub/sub model that decouples event suppliers from event consumers.

To support the requirements of a broad spectrum of application domains, pub/sub middleware for event-based distributed systems typically provides many policies that affect end-to-end system quality of service (QoS) properties. Examples of these policies include persistence, i.e., determining how much data to save for current subscribers; durability, i.e., determining whether to save data for late joining subscribers; and grouped data transfer, i.e., determining if a group of data needs to be transmitted and received as a unit. Moreover, some pub/sub middleware platforms provide a rich set of QoS policies with very fine-grained control.

1.2 Challenges for QoS-enabled Pub/Sub Systems

While tunable policies provide fine-grained control of system QoS, several challenges emerge when developing QoS-enabled pub/sub systems. Configuring and managing QoS has become more complex as QoS support for pub/sub middleware platforms has increased. This increase in complexity manifests itself in
the following ways:

1. **Development of Valid QoS Configurations.** Developing a QoS configuration can be complicated by the following challenges: (1) the number and type of parameters for a single policy, (2) the number of policies available and the legal set of policy interactions, and (3) the accidental complexity inherent in accurately transforming a QoS configuration design into an implementation artifact. Moreover, this same complexity is present whether an initial QoS configuration is being developed or an existing QoS configuration is being updated or modified.

   Each QoS policy supported by a pub/sub middleware platform may have multiple attributes associated with it, such as the data topic of interest, data filter criteria, and the maximum number of data messages to store when transmitting data. Moreover, each attribute can be assigned one of a range of values, such as the legal set of topics, a range of integers for the maximum number of data messages stored for transmission, or the set of criteria used for filtering. Not all combinations of QoS attribute values are legitimate for a single QoS policy nor are all combinations of QoS policies semantically compatible, i.e., produce configurations that elicit the desired flow of data.

   Traditionally, validating a QoS configuration has been done at run-time which impacts development by (1) lengthening the development process since validation can only occur when a system is implemented and running and (2) creating a disconnect between when a QoS configuration problem is found at run-time and when it’s resolved at design-time. The lengthening of the QoS configuration development time and the loss of context for finding and fixing QoS configuration problems decreases developer productivity and increases accidental complexity.

   Furthermore, once a QoS configuration is validated it must be faithfully transformed and incorporated into an implementation. The research challenges addressed in this thesis thus focus on (1) choosing the right set of values for QoS policies, (2) ensuring that these QoS policies are configured together in a semantically compatible way, i.e., that they do not conflict with or contradict each other, and (3) accurately mapping the QoS configuration from the design space to the implementation space.

2. **Evaluation of Run-time QoS Mechanisms for Static Environments.** Mechanisms used by the middleware to ensure certain QoS properties for a given environment configuration may not be applicable for a different environment configuration. For example, a simple unicast protocol (such as UDP) may provide adequate QoS regarding latency when a publisher sends to a small number of subscribers. UDP could incur too much latency, however, when used for a large number of subscribers due to publishers sending UDP messages to each individual subscriber.

   Challenges also arise when enforcing multiple QoS policies that interact with each other. For example, a system might specify low latency QoS and
reliability QoS, which can affect latency due to data loss discovery and recovery. Certain transport protocols (again such as UDP) provide low overhead but no end-to-end reliability. Other protocols (such as TCP) provide reliability, but incur unbounded latencies due to acknowledgment-based retransmissions. Still other protocols balance reliability and low latency, but provide benefit over other protocols only for specific environment configurations. Determining which particular transport protocol as well as the appropriate protocols parameters can be a complex decision.

3. **Run-time Management of QoS for Dynamic Environments.** Even when appropriate QoS mechanisms are determined for a particular environment, perturbations in the environment during run-time, e.g., increase in network latency or packet loss, can cause specified QoS not to be met. The environment in which the system was initially deployed and for which the specified QoS was initially maintained can change causing discontinuity in QoS. Moreover, human intervention is often not responsive enough to meet system timeliness requirements when the environment changes and QoS mechanisms need to be updated.

1.3 **Enhancing Developer Productivity and Run-time Support for QoS-enabled Publish-Subscribe Middleware**

To address the challenges identified in Section 1.2, this dissertation proposes to enhance the productivity, flexibility, and adaptability of QoS-enabled pub/sub middleware. The novel contributions of this dissertation focus on the following three synergistic areas: (1) a model-driven technique for developing semantically compatible QoS configurations and applicable implementation artifacts; (2) a technique for evaluating QoS mechanisms in specific operating environments and providing guidance for run-time QoS management of pub/sub systems in static environments; and (3) a technique to support autonomic adaptation of QoS-enabled pub/sub middleware in dynamic environments.

We briefly summarize the three separate but synergistic contributions proposed by this dissertation as follows:

1. **QoS configuration modeling language for pub/sub DRE systems** which includes capabilities for (1) modeling desired entities and associated QoS policies for a pub/sub DRE system, (2) checking the semantic compatibility of the modeled QoS policies, and (3) and automatically generating implementation artifacts for a configuration model. Section 2.4 describes the QoS modeling process in detail.

2. **Evaluation of QoS mechanisms for pub/sub DRE systems** which includes (1) a technique that integrates and enhances QoS-enabled pub/sub middleware with a flexible network transport protocol framework, (2)
composite QoS metrics to evaluate multiple QoS concerns, and (3) guidance gleaned from performance analysis of the enhanced middleware in different environment configurations. Section 3.4 describes the enhanced middleware, composite metrics, and quantitative analysis in detail.

3. **Autonomic adaptation of pub/sub middleware mechanisms for managing QoS** which includes the integration and enhancement of (1) QoS-enabled pub/sub middleware, (2) a flexible network transport protocol framework, (3) a monitoring subsystem to determine run-time environment configuration information, (4) machine learning to determine an optimal transport protocol and protocol settings for a given environment configuration, and (5) a controller subsystem to autonomically adapt the middleware to the optimal transport protocol. Section 4.4 describes the autonomically adaptive middleware in detail.

### 1.4 Proposal Organization

The remainder of this proposal is organized as follows: each chapter describes a single focus area, the related research, the unresolved challenges, our research approach to solve these challenges, and evaluation criteria for this aspect of the research. Chapter 2 discusses QoS configuration development for pub/sub DRE systems. Chapter 3 discusses a flexible QoS-enabled pub/sub middleware evaluation framework in varied environment configurations. Chapter 4 discusses autonomic adaptation of pub/sub middleware to maintain QoS in dynamic environments. Finally, chapter 5 provides a summary of the research contributions, publications, and a timeline for the thesis completion.
Chapter 2

Design Time Management of QoS Configuration Complexity

Chapter 1 showed an overview of the need for (1) shortening the QoS configuration development cycle, (2) maintaining the context between when a QoS configuration design problem is found and when it is resolved, and (3) faithfully transforming QoS configuration from design to implementation. This chapter presents more in-depth information of our design-time QoS configuration management approach by (1) detailing a motivating example, (2) outlining existing research in the field of QoS configuration management, (3) enumerating unresolved challenges with current research, and (4) resolving the challenges via a solution approach. This chapter also presents an empirical evaluation of the solution approach to generate implementation artifacts for a representative pub/sub DRE system.

2.1 Motivating Example: NASA's Magnetospheric Multiscale Mission

We chose NASA's Magnetospheric Multiscale (MMS) Mission [6] as a case study to showcase the complexities of configuring QoS policies in pub/sub middleware. MMS comprises five co-orbiting and coordinated satellites instrumented identically to study various aspects of the earth's magnetosphere, e.g., turbulence in key boundary regions, magnetic reconnection, and charged particle acceleration. The satellites can be (re)positioned into different temporal/spatial relationships, e.g., to construct a three dimensional view of the field, current, and plasma structures.

An example MMS spacecraft deployment is shown in Figure 2.1.
Figure 2.1: Example MMS Mission Scenario with QoS Requirements

This deployment includes a non-MMS satellite that communicates with the MMS satellites, as well as a ground station that communicates with the satellites during a high-capacity orbit window. The figure also shows the flow of data between systems involved in the deployment, along with the QoS requirements applicable to the MMS mission.

To transport telemetry data, the MMS satellites are equipped with both downlink and uplink capability. To enable precise coordination for particular types of telemetry and positioning data each satellite gathers, stores, and transmits information regarding neighboring spacecraft. Instrumentation on each satellite is expected to generate \( \sim 250 \) megabytes of data per day. To enable the satellites to wait for high-rate transmission windows and thereby minimize ground station cost, each satellite also stores up to 2 weeks worth (i.e., 3.5 GB) of data. To meet these data requirements, the pub/sub middleware used for MMS needs to support the QoS policies summarized in Table 2.1.

<table>
<thead>
<tr>
<th>MMS Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redundancy</td>
<td>data redundancy (store data on another satellite)</td>
</tr>
<tr>
<td>Durability</td>
<td>making data available at a later time for analysis</td>
</tr>
<tr>
<td>Presentation</td>
<td>maintain message ordering and granularity</td>
</tr>
<tr>
<td>Transport priority</td>
<td>prioritizing data transmissions</td>
</tr>
<tr>
<td>Time-based filtering</td>
<td>flow control to handle slow consumers</td>
</tr>
<tr>
<td>Deadline</td>
<td>deadlines on receipt of data</td>
</tr>
<tr>
<td>Reliability</td>
<td>no loss of critical data</td>
</tr>
<tr>
<td>Resource limits</td>
<td>effective provisioning of resources</td>
</tr>
<tr>
<td>Liveliness</td>
<td>assurances of properties when spacecraft is unavailable</td>
</tr>
</tbody>
</table>

Table 2.1: MMS pub-sub QoS Policy Requirements

A challenge for MMS developers is to determine how the interaction of the
QoS policies listed in Table 2.1 impacts the deployed system. Key issues to address involve detecting conflicting QoS settings and ensuring proper behavior of the system in light of such conflicts. Not all combinations of QoS policies and parameter values are semantically compatible, i.e., only a subset actually make sense and provide the needed capabilities. Ideally, incompatibilities in QoS policy configurations should be detected before the MMS system runs so modifications will be less costly and easier to implement, validate, and optimize.

2.2 Related Research

QoS management of DRE systems enables developers to create QoS configurations and incorporate them into implementations. Existing techniques that enable developers to manage QoS configurations can be classified as follows:

**DSMLs for configuring QoS.** There are currently several DSMLs developed to model QoS requirements for distributed real-time embedded (DRE) systems. The *Distributed QoS Modeling Environment* (DQME) [7] is a modeling tool that targets essential elements of dynamic QoS adaptation. DQME is a hybrid of domain-specific modeling and run-time QoS adaption methods, with emphasis on adapting QoS to changing conditions with limited system resources. DQME focuses on QoS solution exploration of a running system by providing run-time QoS adaption strategies as modeling elements to be incorporated into an existing DSML.

The *Options Configuration Modeling Language* (OCML) [8] is a DSML that aids developers in setting compatible component configuration options for the system being developed as opposed to supporting QoS policy configuration for data-centric middleware that can be applicable across various endpoints such as processes, objects, or components. OCML is a modeling language intended to be domain-independent that captures complex DRE middleware and application configuration information along with QoS requirements. It currently supports configuration management only for distributed object computing (DOC) architectures rather than data-centric pub/sub architectures such as DDS. This difference is important because the endpoints receiving data in a system utilizing DDS do not specify details of the type and implementation characteristics of the endpoints. For instance, these endpoints could be processes, objects, or components.

The MicroQoSCORBA [9] middleware for embedded systems includes a GUI-based tool that helps guide the developer with configuration options and provides semantic compatibility for resource constrained environments. More specifically, for each of the various QoS policies allowed, i.e., fault tolerance, security, and timeliness, MicroQoSCORBA supports multiple implementations that enforce any single QoS policy. These implementations are needed to offer different tradeoffs between QoS and resource consumption which is often crucial for embedded systems. Additionally, an implementation for enforcing one QoS policy may not be compatible with an implementation for supporting a different QoS policy due to resource constraints for the particular hardware platform.
The configuration tool guides the developer through reconciling these incompatibilities to ensure the desired balance between QoS and resource consumption. While the configuration tool helps to address the QoS needs of resource-constrained environments, it is targeted to distributed object computing middleware rather than the more generalized pub/sub middleware.

Prism Technologies (www.prismtechnologies.com) has developed a DSML for creating QoS configurations. The DSML checks for validity of the configuration and lets the user know if there are problems. Moreover, the DSML supports generation of implementation artifacts and integration with the OpenSplice DDS implementation. However, PrismTech’s DSML supports only the OpenSplice DDS implementation.

**Runtime monitoring.** Real-time Innovations Inc. (www.rti.com) and Prism Technologies (www.prismtechnologies.com) have developed DDS products along with MDE tools that monitor and analyze the flow of data within a system using DDS. These tools help verify that a system is functioning as designed for a particular QoS policy configuration and for a particular point of execution. However, discovering configuration problems at run-time is very late in the development cycle when problems are more expensive and obtrusive to fix. Moreover, these tools are designed only for the vendor-specific DDS implementation.

**Content-based pub/sub development.** Tools such as Siena [10] and the Publish/subscribe Applied to Distributed REsource Scheduling (PADRES) [11] system provide support for flexible and efficient content-based subscription. PADRES is used for composite event detection and in this vein includes support for expressing time along with bindings for variables, coordination patterns, and composite subscriptions. Siena provides scalability to large content-based networks while minimizing missed deliveries and unnecessary traffic. However, PADRES and Siena do not support correct QoS policy configurations at design-time but rather focus on managing dynamic content-based subscriptions during run-time.

**QoS Profiles.** The Unified Modeling Language (UML) [12, 13] provides a profile for modeling QoS properties and mechanisms [14]. The profile specifies a notation for various QoS categories within UML such as throughput, latency, security, and scalability. The profile does not, however, provide explicit support for all the QoS policies in DDS which is the pub/sub standard providing the richest QoS support. However, extensions to the profile can be made to support arbitrary QoS policies. The profile also does not provide automated enforcement of semantic compatibility between QoS properties at design-time.

### 2.3 Unresolved Challenges

Existing approaches for managing QoS configuration complexity focus on various individual pieces of the problems. For example, some approaches focus only on a particular implementation. Other approaches focus only on components or objects which are subsets of the more generalized pub/sub paradigm. Still other approaches do not focus on QoS aspects and managing the richness of
QoS-enabled pub/sub middleware for DRE systems.

The following challenges represent a gap in the current research regarding design-time validation of QoS configurations:

1. Traditionally, a QoS configuration is developed at design-time and validated at run-time. This development approach creates a lengthy iteration time for QoS configuration design and validation. Moreover, it creates a disconnect between when a QoS configuration problem is discovered (i.e., at run-time) and when it is resolved (i.e., at design-time). This disconnect creates a loss of context which exacerbates the problem of developing valid QoS configurations.

2. The implementation artifacts of a QoS configuration are traditionally coupled and interwined with the business logic artifacts. For example, code for addressing QoS configuration concerns is located in the same source code files as the business logic. This coupling of QoS code and business logic code increases the accidental complexity of developing a valid system.

3. Once a QoS configuration design is validated manual creation of implementation artifacts increases the accidental complexity of a valid implementation.

Our solution approach leverages model-driven engineering (MDE) techniques coupled with a domain-specific modeling languages (DSML) to (1) shorten the iteration cycle for developing a QoS configuration, (2) validate a QoS configuration at design-time so that the context of when a QoS configuration problem is discovered is the same context of when the problem is resolved, and (3) automatically generate implementation artifacts from a valid QoS configuration design thus greatly reducing the accidental complexity of using a QoS configuration once it has been designed.

2.4 Solution Approach: Distributed QoS Modeling Language (DQML)

This section describes the design and implementation of the Distributed QoS Modeling Language, which is a DSML for QoS configurations. The key design goals of DQML are (1) providing design-time validation of QoS configurations and (2) automatically transforming the design to implementation artifacts.

This section also explores the challenges of generating QoS policy configurations for pub/sub middleware and presents DSML-based solutions. We analyze these challenges in the context of a prototype MMS mission (see Section 2.1) implemented using the OMG Data Distribution Service (DDS) [5] pub/sub middleware (see Section 2.4.1). We selected DDS as our middleware platform due to its powerful and flexible standard API and its extensive support for QoS policies all of which are relevant to the MMS mission case study.
2.4.1 Context for DQML

DQML initially utilizes the DDS as a QoS-enabled pub/sub middleware platform. We therefore present a brief overview of DDS. While DDS’s rich set of QoS policies makes it a particularly relevant platform, the DQML’s analysis and approach are also applicable to other pub/sub middleware and application domains. Additionally this section evaluates different solution approaches including the DSML approach which DQML embodies.

Overview of the OMG Data Distribution Service (DDS)

The OMG DDS specification defines a standard architecture for exchanging data in pub/sub systems. DDS provides a global data store in which publishers and subscribers write and read data, respectively. DDS provides flexibility and modular structure by decoupling: (1) location, via anonymous publish/subscribe, (2) redundancy, by allowing any numbers of readers and writers, (3) time, by providing asynchronous, time-independent data distribution, and (4) platform, by supporting a platform-independent model that can be mapped to different platform-specific models, such as C++ running on VxWorks or Java running on Real-time Linux.

The DDS architecture consists of two layers. The Data-Centric Publish Subscribe (DCPS) layer provides efficient, scalable, predictable, and resource-aware data distribution. The Data Local Reconstruction Layer (DLRL) provides an object-oriented facade atop the DCPS so that applications can access object fields rather than data and defines navigable associations between objects. This paper focuses on DCPS since it is better specified and supported than the DLRL.

The DCPS entities in DDS include topics, which describe the type of data to be written or read, data readers, which subscribe to the values or instances of particular topics, and data writers, which publish values or instances for particular topics. Properties of these entities can be configured using combinations of the DDS QoS policies shown in Table 2.2. Moreover, publishers manage groups of data writers and subscribers manage groups of data readers.

Each QoS policy has ~2 parameters, with the bulk of the parameters having a large number of possible values, e.g., a parameter of type long or character string. Section 2.4.1 shows that not all QoS policies are applicable to all DDS entities nor are all combinations of policy values semantically compatible.

DDS provides a wide range of QoS capabilities that can be configured to meet the needs of topic-based distributed systems with diverse QoS requirements. DDS’ flexible configurability, however, requires careful management of interactions between various QoS policies so that the system behaves as expected. It is encumbent upon the developer to use the QoS policies appropriately and judiciously. This section uses our MMS example from Section 2.1 to present the challenges of configuring DDS QoS policies so the system executes as intended.
<table>
<thead>
<tr>
<th>DDS QoS Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline</td>
<td>Determines rate at which periodic data is refreshed</td>
</tr>
<tr>
<td>Destination Order</td>
<td>Sets whether data sender or receiver determines order</td>
</tr>
<tr>
<td>Durability</td>
<td>Determines if data outlives the time when written or read</td>
</tr>
<tr>
<td>Durability Service</td>
<td>Details how durable data is stored</td>
</tr>
<tr>
<td>Entity Factory</td>
<td>Sets enabling of DDS entities when created</td>
</tr>
<tr>
<td>Group Data</td>
<td>Attaches application data to publishers, subscribers</td>
</tr>
<tr>
<td>History</td>
<td>Sets how much data is kept to be read</td>
</tr>
<tr>
<td>Latency Budget</td>
<td>Sets guidelines for acceptable end-to-end delays</td>
</tr>
<tr>
<td>Lifespan</td>
<td>Sets time bound for &quot;stale&quot; data</td>
</tr>
<tr>
<td>Liveliness</td>
<td>Sets liveness properties of topics, data readers, data writers</td>
</tr>
<tr>
<td>Ownership</td>
<td>Controls writer(s) of data</td>
</tr>
<tr>
<td>Ownership Strength</td>
<td>Sets ownership of data</td>
</tr>
<tr>
<td>Partition</td>
<td>Controls logical partition of data dissemination</td>
</tr>
<tr>
<td>Presentation</td>
<td>Delivers data as group and/or in order</td>
</tr>
<tr>
<td>Reader Data Lifecycle</td>
<td>Controls data and data reader lifecycles</td>
</tr>
<tr>
<td>Reliability</td>
<td>Controls reliability of data transmission</td>
</tr>
<tr>
<td>Resource Limits</td>
<td>Controls resources used to meet requirements</td>
</tr>
<tr>
<td>Time Based Filter</td>
<td>Mediates exchanges between slow consumers and fast producers</td>
</tr>
<tr>
<td>Topic Data</td>
<td>Attaches application data to topics</td>
</tr>
<tr>
<td>Transport Priority</td>
<td>Sets priority of data transport</td>
</tr>
<tr>
<td>User Data</td>
<td>Attaches application data to DDS entities</td>
</tr>
<tr>
<td>Writer Data Lifecycle</td>
<td>Controls data and data writer lifecycles</td>
</tr>
</tbody>
</table>

Table 2.2: DDS QoS Policies

Evaluating Common Alternative Solution Techniques

Several alternatives exist to address the challenges of QoS policy configurations described above, including (1) point solutions, which iteratively modify QoS settings based on system feedback, (2) pattern-based solutions, which incorporate documented design expertise, and (3) model-driven engineering (MDE) solutions, which use DSMLs to design and validate configurations and synthesize implementations. Below we evaluate these alternatives in terms of their ability to document and realize proven QoS policy configurations robustly.

**Point solutions.** This approach involves the three step process of (1) making modifications to the existing system’s QoS policies, (2) gathering feedback, and (3) making further modifications based on the feedback. This iterative process can be done either at (1) run-time, i.e., while the system is executing, or (2) development time, i.e., while the system is being developed. In either case, developers must design a proper QoS policy configuration and ensure correct configuration transformation from design to implementation.

Point solutions work best when a configuration expert is available, the configuration is simple, and the configuration need not be maintained or enhanced. Under these circumstances the problem is simplified and the overhead of training others, codifying the expertise, or otherwise developing for modifiability may not be needed.

Point solutions make it hard, however, to capture proven QoS policy configurations or leverage from the expertise of others. The configuration solutions that are designed often need the help and advice of human experts, which can create productivity bottlenecks. If there are no experts available developers must generate expertise “on-the-fly” while solving configuration problems, which is
tedious and error-prone. Moreover, point solutions do not support automated transformation of configuration solutions from design to implementation.

We now describe two types of point solutions.

**(1a) Run-time point solutions:** To modify and evaluate system behavior at run-time, DDS provides run-time mechanisms that notify a system when QoS policies between DDS entities are incompatible or when QoS policies for a given DDS entity are inconsistent. A subsystem for the application could therefore be developed to monitor when incompatibility between DDS entities and inconsistency for a particular DDS entity occurs and make adjustments accordingly while the application is running.

This subsystem would necessarily be fairly complex and intelligent to support management of the different QoS policies and make compatible and consistent QoS policy changes while the system is running. For example, with the MMS example described above, the spacecraft would need to include monitoring software to determine the lack of compatibility and consistency as well as logic to determine the appropriate QoS settings that should be used. Since the spacecraft would at times not be within contact of a ground station, all of this policy configuration management software would need to be autonomous. Additionally, for distributed systems, determining a consensus of appropriate QoS settings can be difficult with the plethora of monitors needed for each subsystem or computing node. The system itself may become unstable as various policies are modified, feedback is gathered, and additional modifications are made. The system may not be able to reach a stable state and would execute in an unpredictable manner.

This approach would incur additional expense, code complexity, and development time. Additionally, this type of solution is unacceptable for some systems such as hard real-time and mission critical systems which are highly sensitive to jitter and latency. It is also unacceptable for systems that require proof of the system properties.

**(1b) Development-time point solutions:** The development-time point solution approach involves an iterative process of coding, compiling, running, and checking compatibility and consistency, which is tedious and error-prone. Moreover, this approach introduces accidental complexity since there is no separation between the code that configures the QoS policies and the code that implements the application. A change to the code to alleviate QoS incompatibilities and inconsistencies can change code affecting other QoS settings or the code that affects the logic of the application. Additionally, testing can be problematic since proving the interaction of the QoS policies to be correct is hard in a non-trivial system. In the MMS example, for instance, the QoS policies for a given configuration must be proven to interact correctly (i.e., be compatible and consistent) so that data will flow as intended.

One modification to the point solutions outlined above is to decouple the QoS policy configuration settings from the application code by using QoS policy configuration files that are read by the application when it is started. This would allow a QoS monitoring and modification harness to be overlaid on top of an existing system. It also addresses some of the accidental complexity of in-
advertently modifying business code logic while changing QoS policy settings. However, since these policy configuration files are edited manually there is still the problem of accidental complexity occurring in the creation and modification of the files themselves. QoS policy names, parameters, and values could be mistyped. Additionally, there remains the accidental complexity of mistakenly modifying a QoS policy unrelated to the current development iteration. This approach may be feasible for simple systems where all the configuration paths can be rigorously tested and the system will not be expanded or enhanced at a later point. However, this is not the case for most systems interested in incorporating QoS properties.

**Pattern-based solutions.** In this approach configuration patterns are used to address QoS policy configuration challenges. The patterns document the use of QoS policies that provide shaping, prioritization, and management of a dataflow in a network [15]. For example, developers of DDS-based systems could limit access to certain data by using the DDS Controlled Data Access pattern, which utilizes the DDS Partition and User Data QoS Policies along with other DDS elements to provide the desired QoS.

Configuration patterns enable the codification of configuration expertise so that it is clearly documented and can be broadly reused. These patterns address the problems of human expert availability by making the configuration policy expertise generally available. However, a drawback with a pattern-based approach is the responsibility developers have for correctly transferring the configuration design into implementation manually, which can be tedious and error-prone. Moreover, various developers may implement the patterns in different ways, which can impede reuse and large-scale system integration.

Moreover, DDS is a relatively new technology and there are currently a limited set of patterns that have been documented. The catalogue of available patterns may not address a given configuration scenario. It will take some time to build up the catalog of DDS patterns and to fully understand their ramifications. In short, pattern-based solutions address the design challenges for non-trivial QoS policy configurations but do not address the implementation challenges.

**DSML-based solutions.** This approach to addressing the complexity of managing QoS policy configurations involves the use of DSMLs that codify configuration expertise in the metamodels developed for a particular domain. DSMLs also use an executable form of that expertise to synthesize part or all of an implementation. For example, DSMLs can generate valid QoS policy configuration files from valid QoS policy configurations modeled in the DSMLs.

DSMLs can also ensure (1) proper semantics for specifying QoS policies and (2) all parameters for a particular QoS policy are properly specified and used correctly, as described in Section 1. At design time, therefore, they can detect many types of QoS policy configuration problems, such as invalid parameter values for a QoS policy and conflicting QoS policies. They can also automate the generation of implementation artifacts (e.g., source code and configuration files) that reflect design intent. Due to these benefits, this paper focuses on DSML-based solutions.
DDS QoS Policy Configuration: Challenges and DSML-based Solutions

In the context of DDS and the MMS case study, we developed a DSML-based solution to four types of challenges that arise when creating QoS policy configurations. We chose a DSML-based solution over other common solution techniques (such as manually-implementing point- and pattern-based [15] solutions) since DSMLs can ensure (1) proper semantics for specifying QoS policies and (2) all parameters for a particular QoS policy are properly specified and used correctly, as described in Section 1. DSMLs can also detect many types of QoS policy configuration problems at design time and can automatically generate implementation artifacts (e.g., source code and configuration files) that reflect design intent.

MMS Challenge 1: Managing QoS Policy Configuration Variability.

Context. DDS provides three points of variability with respect to QoS policy configurations: (1) the associations between a single DDS entity and two or more QoS policies, (2) the associations between two or more entities, and (3) the number and types of parameters per QoS policy.

Problem. When creating a DDS QoS policy configuration, associations are made between various entities, e.g., between a data writer sending collected data from an MMS satellite and the publisher that manages the data writer. Not all possible associations are valid, however. For example, the association between a data writer and a subscriber is invalid since a subscriber manages one or more data readers and not data writers. If the rules governing valid associations between entities are not obeyed when associations are created the QoS policy configuration will be invalid.

Associations can be made not only between DDS entities but also between a DDS entity and the QoS policies. Not all QoS policies are valid for all DDS entities, however. For instance, associating a Presentation QoS Policy with an MMS ground station’s data reader is invalid. The rules that determine which QoS policies can be associated with which DDS entities must be considered when creating valid QoS policy configurations.

Finally, the number and types of parameters differ for each QoS policy type. The number of parameters for any one QoS policy ranges from one (e.g., Deadline QoS Policy) to six (e.g., Durability Service QoS Policy). The parameter types for any one QoS policy also differ. The parameter types include boolean, string, long, struct, and seven different types of enums. It is hard to track the number of parameters a particular QoS policy has manually; it is even harder to track the valid range of values that any one single parameter can have.

General DSML-based solution approach. A DSML can ensure that only appropriate associations are made between entities and QoS policies. In addition, a DSML can list the parameters and default values of any selected QoS policy. DSMLs ensure that only valid values are assigned to the QoS policy parameters. For example, a DSML can raise an error condition if a string is assigned to a parameter of type long. Section 2.4.2 describes how DQML addresses the QoS policy configuration variability challenge by allowing only valid values to be as-
signed to parameters and checking for valid associations between QoS policies and entities.

**MMS Challenge 2: Ensuring QoS compatibility.**

*Context.* DDS defines constraints for compatible QoS policies. Table 2.3 lists the QoS policies that can be incompatible and the relevant types of entities for those policies. Incompatibility applies to QoS policies of the same type, e.g., reliability, across multiple types of entities, e.g., data reader and data writer.

<table>
<thead>
<tr>
<th>QoS Policies</th>
<th>Affected DDS Entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline</td>
<td>Topic, data reader, data writer</td>
</tr>
<tr>
<td>Destination</td>
<td>Topic, data reader, data writer</td>
</tr>
<tr>
<td>Order</td>
<td>Topic, data reader, data writer</td>
</tr>
<tr>
<td>Durability</td>
<td>Topic, data reader, data writer</td>
</tr>
<tr>
<td>Latency Budget</td>
<td>Topic, data reader, data writer</td>
</tr>
<tr>
<td>Liveliness</td>
<td>Topic, data reader, data writer</td>
</tr>
<tr>
<td>Ownership</td>
<td>Topic, data reader, data writer</td>
</tr>
<tr>
<td>Presentation</td>
<td>Publisher, subscriber</td>
</tr>
<tr>
<td>Reliability</td>
<td>Topic, data reader, data writer</td>
</tr>
</tbody>
</table>

Table 2.3: Potential Incompatible DDS QoS Policies

*Problem.* When compatibility constraints are violated, data will not flow between DDS data writers and data readers, i.e., compatibility impacts topic dissemination. For example, an incompatibility between deadline QoS policies will occur if an MMS ground station expects data updates at least every 5 seconds but an MMS spacecraft only commits to data updates every 10 seconds. The data will not flow between the spacecraft and the ground station because the values of the QoS policies are incompatible, as shown in Figure 2.2.

![Figure 2.2: Incompatible MMS Ground Station and Spacecraft Deadline QoS](image)

*General DSML-based solution approach.* A DSML can include compatibility checking in the modeling language itself. A DSML user can invoke compatibility checking to make sure that the QoS policy configuration specified is valid. If incompatible QoS policies are detected the user is notified at design time and given details of the incompatibility. Section 2.4.2 describes how DQML addresses the QoS compatibility challenge by providing compatibility constraint checking on
QoS policy configurations.

**MMS Challenge 3: Ensuring QoS consistency.**

*Context.* The DDS specification defines when QoS policies are inconsistent, i.e., when multiple QoS policies associated with a single DDS entity are not valid. Table 2.4 describes the consistency constraints for QoS policies associated with a single DDS entity. For example, an inconsistency between the *Deadline* and *Time-based Filter* QoS policies occurs if an MMS ground station tries to set the *Deadline* QoS Policy’s deadline period to 5 ms and the *Time-based Filter* QoS Policy’s minimum separation between incoming pieces of data to 10 ms, as shown in Figure 2.3. This invalid configuration violates the DDS constraint of deadline period \( \geq \) minimum separation.

<table>
<thead>
<tr>
<th>Consistency Constraints for QoS Policies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline.period ( \geq ) Time-based Filter.minimum_separation</td>
</tr>
<tr>
<td>Resource.Limits.max_samples ( \geq ) Resource.Limits.max_samples_per_instance</td>
</tr>
<tr>
<td>Resource.Limits.max_samples_per_instance ( \geq ) History.depth</td>
</tr>
</tbody>
</table>

Table 2.4: DDS QoS Consistency Constraints

![Figure 2.3: Inconsistent QoS Policies for an MMS Ground Station](image)

**Problem.** Manually checking for all possible consistency constraint violations is tedious and error-prone for non-trivial pub/sub systems.

**General DSML-based solution approach.** A DSML can include consistency checking in the modeling language itself. As with compatibility checking, DSML users can invoke consistency checking to ensure that the QoS policy configuration is valid. If inconsistent QoS policies are found, users are notified at design time with detailed information to help correct the problem. Section 2.4.2 describes how DQML addresses the QoS consistency challenge by providing consistency constraint checking on QoS policy configurations.

**MMS Challenge 4: Ensuring Correct QoS transformation.**

*Context.* After a valid QoS policy configuration has been created it must be correctly transformed from design to implementation.

*Problem.* A conventional approach is to (1) document the desired QoS policies, parameters, values, and associated entities often in an *ad hoc* manner (e.g., using handwritten notes or conversations between developers) and then (2) transcribe this information into the source code. This *ad hoc* process creates opportunities for accidental complexities, however, since the QoS policies, parameters, values, and related entities can be misread, mistyped, or misunderstood.
The QoS policy configurations encoded in the system may therefore differ from the valid configurations intended originally.

**General DSML-based solution approach.** A DSML can provide model interpreters to generate correct-by-construction implementation artifacts. The interpreters iterate over the QoS policy configuration model designed in the DSML to create appropriate implementation artifacts (e.g., source code, configuration files) that will correctly recreate the QoS policy configuration as designed. Section 2.4.2 describes how DQML addresses the challenge of correct QoS transformation by providing an interpreter that traverses the model and generates implementation specific artifacts.

**2.4.2 Structure and Functionality of DQML**

The Distributed QoS Modeling Language (DQML) is a DSML that automates the analysis and synthesis of semantically compatible DDS QoS policy configurations. We developed DQML using the Generic Modeling Environment (GME) [16], which is a meta-programmable environment for creating DSMs. This section describes the structure and functionality of DQML and explains how it resolves the challenges from Section 2.4.1 in the context of DDS and the MMS case study.

**Structure of the DQML Metamodel**

The DQML metamodel constrains the possible set of models for QoS policy configurations as described below.

**Scope.** The DQML metamodel includes all DDS QoS policy types shown in Table 2.2, but supports only DDS entity types that have QoS policies associated with them. In addition to topics, data readers, and data writers previously mentioned, DQML can associate QoS policies with (1) publishers, which manage one or more data writers, (2) subscribers, which manage one or more data readers, (3) domain participants, which are factories for DDS entities for a particular domain or logical network, and (4) domain participant factories, which generate domain participants. While other entities and constructs exist in DDS, none directly use QoS policies and are thus excluded from DQML.

As an exemplar, Figures 2.4 and 2.5 illustrate a portion of the DQML metamodel pertaining to the Deadline QoS Policy. Figure 2.4 shows the part of the DQML metamodel relevant to the Deadline QoS Policy and its relationships to applicable DDS entities, i.e., data reader, data writer, and topic. Figure 2.5 shows the part of the DQML metamodel relevant to the OCL constraints placed on the Deadline QoS Policy to ensure semantic compatibility. The compatibility constraints are associated with a topic since compatibility between a data reader and a data writer is determined by a common topic. This figure shows the appropriate relationships and the number of associations. In a manner similar to

---

1In this paper “correct-by-construction” refers to QoS policy configuration artifacts that faithfully transfer design configurations into implementation and deployment.
Figures 2.4 and 2.5 the remainder of the metamodel describes the rest of the QoS policies including the parameters and constraints for each policy.

**Associations between entities and QoS policies.** DQML supports associations between DDS entities and QoS policies rather than having DDS entities contain or own QoS policies. This metamodel design decision allows greater flexibility and ease of constraint error resolution. If QoS policies had been contained by the DDS entities then multiple DDS entities could not share a common QoS policy. Instead, the policy would be manually copied and pasted from one entity to another, thereby incurring accidental complexity when designing a QoS policy configuration.

In contrast, DQML supports multiple DDS entities having the same QoS policy by allowing modelers to create a single QoS policy with the appropriate values. Modelers can then create associations between the applicable DDS entities and the QoS policy. This approach also simplifies constraint errors resolution, e.g., if constraint errors are found, the offending entities can be associated with a common QoS policy to eliminate the compatibility error.

**Constraint definition.** The DDS specification defines constraints placed on QoS policies for compatibility and consistency. The DQML metamodel uses GME’s Object Constraint Language (OCL) [17] implementation to define these constraints. As noted in Section 2.4.1 for challenges 2 and 3, compatibility constraints involve a single type of QoS policy associated with more than one DDS entity, whereas consistency constraints involve a single DDS entity with more than one QoS policy. In particular, Figure 2.5 highlights the OCL constraint that catches the deadline incompatibility of Figure 2.2. Both incompatibility and inconsistency constraints are defined in the metamodel and can be checked when explicitly initiated by users.

To maximize flexibility, DQML does not enforce semantic compatibility constraints automatically in the metamodel since users may only want to model some parts of a DDS application, rather than model all required entities and QoS policies. Only checking constraints when initiated by modelers enables this flexibility. Conversely, association constraints (i.e., the valid associations
Incompatibility caught when data writer deadline > data reader deadline

between DDS entities and QoS policies) are defined in the metamodel and are thus checked automatically when associations are specified.

Functionality of DQML

DQML allows developers to designate any number of DDS entity instances involved with QoS policy configuration. For example, DQML supports seven DDS entity types that can be associated with QoS policies, as shown in Figure 2.6. QoS policies can be created and associated with these entities as described below.

![Deadline QoS Policy Compatibility Constraints](image)

Figure 2.5: Deadline QoS Policy Compatibility Constraints

Speciﬁcation of QoS policies. DQML allows developers to designate the DDS QoS policies involved with a QoS policy conﬁguration. DQML supports all DDS policies, along with their parameters, the appropriate ranges of values,
and the default parameter values. Developers can then change default settings for QoS policy parameters as needed. Moreover, if a QoS policy parameter has a limited range of values, DQML enumerates only these specific values and ensures that only one of these values is assigned to the parameter.

DQML also ensures that the type of value assigned is appropriate. For example it ensures that a character value is not assigned to a parameter that requires an integer value. The DQML interpreter externalizes the parameter values (whether set explicitly or by default) so that no QoS policy has uninitialized parameters.

Figure 2.7 shows an example of how DQML addresses the challenge of managing QoS policy configuration variability as outlined in Section 2.4.1. In this example DQML displays the parameters for the History QoS Policy along with the default values for the parameters in grey, i.e., history_depth = 1 and history_kind = KEEP_LAST. Since history_kind is an enumerated type, DQML lists the valid values when the user selects the parameter. Only one of the valid values can be assigned to the parameter.

Figure 2.7: Example of DQML QoS Policy Variability Management

Association between entities and QoS policies. DQML supports generating associations between the DDS entities themselves and between a DDS entity and the QoS policies. DQML ensures that only valid associations are created, i.e., where it is valid to associate two particular types of entities or associate a particular DDS entity with a particular type of QoS policy. DQML will notify developers if the association is invalid and disallow the association at design-time.

Checking compatibility and consistency constraints. DQML supports checking for compatible and consistent QoS policy configurations. Users initiate this checking and DQML reports any violations. Constraint checking in DQML uses default QoS parameter values to determine QoS compatibility and consistency if no values are specified. Developers of QoS policy configurations might explicitly associate only a single QoS policy to an entity and assume no checking for compatibility or consistency is applicable. A constraint violation may exist, however, depending on the interaction of the explicit parameter values and the default values for other entities.
For instance, if developers specify only a single Presentation QoS Policy in a configuration, associate it with a single subscriber entity, and change the default access scope value from instance to topic or group, they may assume no constraint violations occur. The explicit access scope value set on the subscriber is incompatible, however, with the implicit (default) value of instance for any publisher associated via a common topic.

The constraint resolution problem is further exacerbated by QoS policies that can be associated with a topic entity and then act as the default QoS policy for data readers or writers. For example, the Reliability QoS Policy can be associated with a data reader, a data writer, or a topic. If the policy is associated with a topic, any data readers or data writers not explicitly associated with a reliability policy will use the topic’s Reliability QoS Policy. DQML can check this type of QoS association for compatibility and consistency.

Figures 2.8 and 2.9 show examples of how DQML addresses the challenges of ensuring QoS compatibility and consistency, respectively, as described in Section 2.4.1.

Figure 2.8: Example of DQML QoS Policy Compatibility Constraint Checking

Figure 2.9: Example of DQML QoS Policy Consistency Constraint Checking
deadline period, i.e., 10 is less than the time based filter’s minimum separation of 15. Both policies are associated with the same MMS Ground Station data reader. DQML checks the consistency of the modeled QoS policies and notifies users of violations.

**Transforming QoS policy configurations from design to implementation.** Figure 2.10 shows how DQML addresses the challenge of correctly transforming QoS policy configurations from design to implementation, as described in Section 2.4.1. In this example, DQML generates the QoS policy configuration file for an MMS satellite data writer as modeled in Figure 2.7. The History QoS Policy associated with the data writer is shown along with values for the policy. This file can then be seamlessly integrated into the MMS implementation to ensure the desired QoS policy configuration.

![History QoS Policy parameters implemented as designed](image)

**Figure 2.10: QoS Policy Configuration File for Figure 2.7**

### 2.4.3 DQML Productivity Analysis for the MMS Case Study

In this section we analyze the pros and cons of DQML by applying it in the context of the DDS Benchmarking Environment (DBE) to evaluate the QoS behavior of the MMS scenario presented in Figure 2.1. DBE is a suite of software tools that can examine and evaluate various DDS implementations [18]. DBE requires correct QoS policy settings so that data will flow as expected. If these policy settings are semantically incompatible QoS evaluations will not run properly. DBE uses a set of Perl scripts that launches executables for the DDS application, e.g., to deploy data readers and data writers onto specified nodes. For each data reader and data writer DBE also deploys a QoS policy settings file that is currently generated manually.

This section presents the results of a productivity analysis using DQML. In particular, we present the productivity benefit and the break-even point of using DQML vs. manually implementing QoS policy configurations for DBE. Manual implementation of configurations is applicable to both the point- and pattern-based solutions presented in previous work [19] since neither approach provides implementation guidance.

**The DQML DBE Interpreter**

To support DBE and its need to generate correct QoS policy configurations we developed a DQML interpreter that generates QoS policy parameter settings files for the data readers and data writers that DBE configures and deploys. This interpreter can also accommodate other DDS entities, e.g., topics, publishers, and subscribers. All QoS policies from a DQML model are output for the data readers and data writers.
The DQML interpreter creates one QoS policy parameter settings file for each data reader or data writer that is modeled. The names of the files are generated by using the name of the data reader or data writer prepended with either “DR” or “DW” plus the current count of data readers or data writers processed (e.g., DR1_Satellite1.txt). The filename prefix is generated to ensure that a unique filename is created since the names of the data readers and data writers modeled in DQML need not be unique.

A common DBE use-case for DQML thus becomes (1) model the desired DDS entities and QoS policies in DQML, (2) invoke the DBE interpreter to generate the appropriate QoS settings files, and (3) execute DBE to deploy data readers and data writers using the generated QoS settings files.

Productivity Analysis

Scope. DBE currently deals only with DDS data readers and data writers. Our productivity analysis therefore focuses on the QoS parameters relevant to data readers and data writers. (Similar analysis can be done for other types of DDS entities associated with QoS policies.) At a minimum, in the MMS scenario each MMS satellite, non-MMS satellite, and ground station will have a data writer and data reader to send and receive data, respectively, which yields seven data readers and seven data writers to configure. This scenario provides the minimal baseline since production satellites and ground stations typically have many data writers and data readers for use in sending and receiving not only to other systems but also for use internally between various subsystems.

<table>
<thead>
<tr>
<th>QoS Policy</th>
<th># of Params</th>
<th>Param Type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>Destination Order</td>
<td>1</td>
<td>enum</td>
</tr>
<tr>
<td>Durability</td>
<td>1</td>
<td>enum</td>
</tr>
<tr>
<td>Durability Service</td>
<td>6</td>
<td>5 ints, 1 enum</td>
</tr>
<tr>
<td>History</td>
<td>2</td>
<td>1 enum, 1 int</td>
</tr>
<tr>
<td>Latency budget</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>Lifespan</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>Liveliness</td>
<td>2</td>
<td>1 enum, 1 int</td>
</tr>
<tr>
<td>Ownership</td>
<td>1</td>
<td>enum</td>
</tr>
<tr>
<td>Ownership Strength</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>Reliability</td>
<td>2</td>
<td>1 enum, 1 int</td>
</tr>
<tr>
<td>Resource Limits</td>
<td>3</td>
<td>3 ints</td>
</tr>
<tr>
<td>Transport Priority</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>User Data</td>
<td>1</td>
<td>string</td>
</tr>
<tr>
<td>Writer Data Lifecycle</td>
<td>1</td>
<td>bool</td>
</tr>
</tbody>
</table>

| Total Parameters    | 25          |

Table 2.5: DDS QoS Policies for data writers

A data writer can be associated with 15 QoS policies with a total of 25 parameters, as shown in Table 2.5. A data reader can be associated with 12 QoS policies with a total of 18 parameters, as shown in Table 2.6. The total number of relevant QoS parameters for DBE is thus $18 + 25 = 43$. Each QoS parameter value for a data reader or writer corresponds to one line in the QoS policy parameter settings file for DBE, as shown in Figure 2.10.
<table>
<thead>
<tr>
<th>QoS Policy</th>
<th># of Params</th>
<th>Param Type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>Destination Order</td>
<td>1</td>
<td>enum</td>
</tr>
<tr>
<td>Durability</td>
<td>1</td>
<td>enum</td>
</tr>
<tr>
<td>History</td>
<td>2</td>
<td>1 enum, 1 int</td>
</tr>
<tr>
<td>Latency budget</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>Liveliness</td>
<td>2</td>
<td>1 enum, 1 int</td>
</tr>
<tr>
<td>Ownership</td>
<td>1</td>
<td>enum</td>
</tr>
<tr>
<td>Reader Data Lifecycle</td>
<td>2</td>
<td>2 ints</td>
</tr>
<tr>
<td>Reliability</td>
<td>2</td>
<td>1 enum, 1 int</td>
</tr>
<tr>
<td>Resource Limits</td>
<td>3</td>
<td>3 ints</td>
</tr>
<tr>
<td>Time Based Filter</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>User Data</td>
<td>1</td>
<td>string</td>
</tr>
<tr>
<td><strong>Total Parameters</strong></td>
<td><strong>18</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.6: DDS QoS Policies for data readers

**Interpreter development.** DQML’s DBE interpreter was developed using GME’s Builder Object Network (BON2) framework, which generates C++ code using the Visitor pattern [20]. Within BON2, developers of the DQML DBE interpreter need only modify and add certain portions to the framework that are called to process the particular DSML. In particular, BON2 provides a C++ visitor class with virtual methods (e.g., visitModelImpl, visitConnectionImpl, visitAtomImpl) that the developer subclasses and then overrides the virtual methods.

In BON2, the DDS entities supported in DQML are referred to as model implementations. The DBE interpreter is thus only concerned with overriding the visitModelImpl method. When the BON2 framework invokes this method it passes as an argument a model implementation. A model implementation has methods to (1) traverse the associations a DDS entity has using the getConnEnds method and specifying the relevant QoS policy association as an input parameter (e.g., the association between a data reader and a Reliability QoS Policy), (2) retrieve the connected QoS policy, and (3) obtain the attributes of the associated QoS policy using the policy’s getAttributes method.

The DQML-specific code for the DBE interpreter contains 160 C++ statements that were developed specifically for DQML and DBE. The C++ development effort for the DBE interpreter need only occur once. In particular, no QoS policy configuration for DBE incurs this development overhead since the interpreter already exists. The development effort is included only for comparison with manually implemented QoS policy configurations.

The interpreter code is fairly straightforward once developers understand how to navigate the model in the BON2 framework and access the appropriate information. Although developers should be familiar with the Visitor pattern [20] (since the BON2 framework uses it heavily), they only need define the appropriate methods for the automatically generated Visitor subclass. In general, the DQML interpreter code specific to DBE gathers model information, creates the QoS settings files, and outputs the settings into the QoS settings files.

The most challenging part of developing DQML’s DBE interpreter is navigating through the model’s QoS policy elements and related entities using the
BON2 framework. Conversely, the most challenging aspects of handcrafting QoS policy configurations are (1) maintaining a global view of the model to ensure compatibility and consistency and (2) remembering the number of and valid values for the parameters of the various QoS policies. For non-trivial QoS policy configurations, therefore, developing the DQML-specific C++ code for the interpreter is no more complex than manually ensuring that the QoS settings in settings files are valid, consistent, compatible, and correctly represent the designed configuration.

Analysis for the MMS scenario. As a conservative approximation, the creation and use of the DBE interpreter for DQML has its break-even point for a single QoS policy configuration when there are at least 160 QoS policy parameter settings needed, which correlates to the 160 C++ statements for DQML’s DBE interpreter. As shown in Figure 2.11, using the results for QoS parameters in Tables 2.5 and 2.6 for data readers and data writers, the break-even point equates to ~10 data readers, ~7 data writers, or some combination of data readers and data writers where the QoS settings are greater than or equal to 160 (e.g., 5 data readers and 3 data writers = 160 QoS policy parameter settings).

From the analysis above—and using the minimal MMS scenario in Figure 2.1 of 7 data writers and 7 data readers—the total number of QoS parameters to consider is 7 * 25 (for data writers) + 7 * 18 (for data readers) = 301. This number exceeds the 160 lines of C++ code developed for the DBE interpreter and shows that the minimal MMS scenario warrants the use of DQML and the creation and use of the DBE interpreter. Using DQML for this scenario provides a (301 - 160) ÷ 301 = 47% reduction in development effort as compared to manual methods.

![Figure 2.11: Metrics for Manual Configuration vs. DQML’s Interpreter](image)

Generalized analysis. The break-even analysis above is relevant to generating a single QoS policy configuration. The analysis does not consider any subsequent modifications to an existing configuration or development of new
configurations for DBE that would not require any modifications to interpreter code. Changes made to a configuration also require that developers (1) maintain a global view of the model to ensure compatibility and consistency and (2) remember the number of, and valid values for, parameters of the various QoS policies being modified. These challenges still exist when changing an already valid QoS policy configuration.

Moreover, there may be thousands of data readers and writers in large-scale DDS systems, e.g., shipboard computing or air-traffic management environments [21]. Assuming 1,000 data readers and 1,000 data writers, the number of QoS parameters to manage is $17 \times 1000 + 25 \times 1000 = 42,000$. This number does not include QoS parameter settings for other DDS entities such as publishers, subscribers, and topics. For such large-scale DDS systems the development cost of the DQML interpreter in terms of lines of code is amortized by more than 200 times (i.e., $43,000 / 160 = 268.75$).

2.5 Lessons Learned

DQML is a DSML we developed to address key challenges of pub/sub middleware, including (1) managing QoS policy configuration variability, (2) developing semantically compatible configurations, and (3) correctly transforming QoS policy configurations from design to implementation. In particular, DQML addresses the challenge of QoS policy compatibility by allowing only valid connections between DDS entities and QoS policies. It also provides compatibility constraint checking on a QoS policy configuration model as it’s being designed. In addition, it addresses the challenge of QoS policy consistency by providing consistency constraint checking during QoS policy configuration design time. Finally, it addresses the challenge of QoS policy configuration transformation by providing interpreters that generate “correct-by-construction” implementation and deployment artifacts that can be incorporated into the system implementation.

DQML currently does not attempt to address other areas of interest for a system which uses DDS such as deployment of DDS entities onto computer nodes. While this is an interesting and needed area for research and development, it falls outside the initial objectives of DQML and is an area for future work. The current focus is intentionally limited to modeling compatible and consistent QoS policies for DDS entities.

The following lessons learned summarize our experience using DQML to model QoS policy configurations for the OMG Data Distribution Service (DDS) in the context of the MMS mission.

• OCL presents a significant learning curve for typical application developers. Many application developers who are accustomed to using a functional or object-oriented language, such as Java, C, or C++, are not familiar with rule-based constraint languages, such as OCL. Moreover, tool support for OCL is often rudimentary, e.g., limited debugging support, which impedes productivity. In future work we plan to address enforcing constraints by evaluating other
constraint solving technologies, such as the Constraint Logic Programming Finite Domain (CLP(FD)) solver [22].

- **Run-time feedback provides crucial system performance insight.** While DQML ensures valid QoS policy configurations, some system properties (e.g., latency and CPU resource utilization) are best evaluated at run-time. Incorporating this type of dynamic information back into a QoS policy configuration model helps increase overall development productivity and system robustness. We are evaluating ways to incorporate runtime and emulation feedback [23] into DQML to enhance QoS policy configuration development.

GME can be downloaded from [www.isis.vanderbilt.edu/Projects/gme](http://www.isis.vanderbilt.edu/Projects/gme). DQML is part of the Component Synthesis Model-Integrated Computing (CosMIC) tool suite which can be downloaded from [http://www.dre.vanderbilt.edu/cosmic](http://www.dre.vanderbilt.edu/cosmic).
Chapter 3


Chapter 1 presented an overview of the need for run-time evaluation of QoS mechanisms for pub/sub DRE systems. This chapter presents more in-depth information by (1) detailing a motivating example, (2) outlining existing research in the field of empirical evaluations of QoS mechanisms for pub/sub DRE systems, (3) enumerating unresolved challenges with current research, and (4) resolving the challenges via a solution approach. This chapter also presents empirical metrics data obtained and evaluated using the solution approach.

3.1 Motivating Example: Search and Rescue (SAR) Operations for Disaster Recovery

To highlight the challenges of providing timely and reliable event stream processing for QoS-enabled pub/sub DRE applications, we present our work in the context of supporting search and rescue (SAR) operations. These operations help locate and extract survivors in a large metropolitan area after a regional catastrophe, such as a hurricane, earthquake, or tornado. SAR operations can use unmanned aerial vehicles (UAVs), existing operational monitoring infrastructure (e.g., building or traffic light mounted cameras intended for security or traffic monitoring), and (temporary) datacenters to receive, process, and transmit event stream data from various sensors and monitors to emergency vehicles that can be dispatched to areas where survivors are identified.

Figure 3.1 shows an example SAR scenario where infrared scans along with GPS coordinates are provided by UAVs and video feeds are provided by existing infrastructure cameras.
These infrared scans and video feeds are then sent to a datacenter, where they are processed by fusion applications to detect survivors. Once survivors are detected, the application will develop a three-dimensional view and highly accurate position information so that rescue operations can commence.

A key requirement of the data fusion applications within the datacenter is tight timing bounds on correlated event streams such as the infrared scans coming from UAVs and video coming from cameras mounted atop traffic lights. The event streams need to match up closely so the survivor detection application can produce accurate results. If an infrared data stream is out of sync with a video data stream, the survivor detection application can generate a false negative and fail to initiate needed rescue operations. Likewise, without timely data coordination, the survivor detection software can generate a false positive, expending scarce resources such as rescue workers, rescue vehicles, and data center coordinators unnecessarily.

Meeting the requirements of SAR operations as outlined in Section 3.1 is hard due to the inherent complexity of synchronizing multiple event data streams. These requirements are exacerbated since SAR operations will run in varying environments where resource availability changes from one disaster environment to another. One operating environment might only provide a very restrictive set of resources and conditions (e.g., highly unreliable, low-bandwidth networks with many senders and receivers of data utilizing the network). Another operating environment might provide a relative surfeit of resources and conditions (e.g., relatively reliable, high-bandwidth networks with few senders and receivers) where more fine-grained data can be accommodated (e.g., higher resolution video). The remainder of this section describes four challenges that FLEXMAT addresses to support the communication requirements of the SAR operations presented above.

**SAR Challenge 1: Maintaining Data Timeliness and Reliability** SAR operations must receive sufficient data reliability and timeliness so that multiple data streams can be fused appropriately. For example, the SAR operation example described above highlights the exploitation of data streams (such as infrared scan and video streams) by several applications simultaneously in a datacenter. Figure 3.2 shows how fire detection applications and power grid assessment
applications can use infrared scans to detect fires and working HVAC systems respectively. Likewise, Figure 3.3 shows how security monitoring and structural damage applications can use video stream data to detect looting and unsafe buildings respectively. Section 3.4.1 describes how FLEXMAT addresses this challenge by incorporating transport protocols that balance reliability and low latency.

Figure 3.2: Uses of Infrared Scans during Disaster Recovery

Figure 3.3: Uses of Video Stream during Disaster Recovery

**SAR Challenge 2: Managing Subscription of Event Data Streams Dynamically** SAR operations must seamlessly incorporate and remove particular event data streams dynamically as needed. Ideally, an application for SAR operations should be shielded from the details of when other applications begin to use common event data streams. Moreover, applications should be able to switch to higher fidelity streams as they become available. Section 3.4.1 describes how we address this challenge by using anonymous QoS-enabled pub/sub middleware that seamlessly manages subscription and publication of data streams as needed.

**SAR Challenge 3: Providing Predictable Performance in Varying Environment Configurations** In scenarios where operating environments vary, such as with regional disasters, the performance of SAR operations must be known a priori. SAR operations tested only under a single environment configuration may not perform as needed when introduced to a new environment. The operations could unexpectedly shut down at a time when they are needed most
due to changes in the environment. Section 2.4.3 describes how we determine application performance behavior for varying environments.

**SAR Challenge 4: Adapting to Changing Environments** SAR operations not only must understand their behavior in a single environment configuration, they must also adjust to different operating environments. If SAR operations across different disaster scenarios cannot adjust then they will fail to perform adequately for different operating environments presented by various disaster situations. If resources change from one operating environment to another, the SAR operations must be configured to accommodate fewer resources while maintaining a minimum level of service. If resources are added, the operations should use them to provide higher fidelity or more expansive coverage. Section 3.4.1 describes how we are incorporating flexible transport protocols that can be easily adjusted for reliability, latency, and/or network bandwidth usage.

### 3.2 Related Research

Evaluation of QoS mechanisms for pub/sub DRE systems enables developers to understand the impact of various QoS mechanisms upon the QoS of the DRE pub/sub system. Existing techniques that enable developers to evaluate QoS mechanisms can be classified as follows:

**Performance evaluation of network transport protocols.** Much prior work has evaluated network transport protocols, e.g., Balakrishnan *et al.* [24] evaluate the performance of the Ricochet transport protocol with the Scalable Reliable Multicast (SRM) protocol [25]. Bateman *et al.* [26] compare the performance of TCP variations both using simulations and in a testbed. Cheng *et al.* [27] provide performance comparisons of UDP and TCP for video streaming in multihop wireless mesh networks. Kirschberg *et al.* [28] propose the Reliable Congestion Controlled Multicast Protocol (RCCMP) and provide simulation results for its performance. These evaluations specifically target the protocol level independent of the context of QoS-enabled pub/sub middleware or composite QoS pub/sub concerns such as reliability and low latency.

**Performance evaluation of enterprise middleware.** Xiong *et al.* [18] conducted performance evaluations for three DDS implementations, including OpenDDS. That work highlighted the different architectural approaches taken and trade-offs of these approaches. However, that prior work did not include performance evaluations of various transport protocols as QoS mechanisms for DDS.

Sachs *et al.* [29] present a performance evaluation of message-oriented middleware (MOM) in the context of the SPECjms2007 standard benchmark for MOM servers. The benchmark is based on the Java Message Service (JMS). In particular, the work details performance evaluations of the BEA WebLogic server under various loads and configurations. However, that work did not integrate various transport protocols as QoS mechanisms for the middleware to evaluate its performance.

Tanaka *et al.* [30] developed middleware for grid computing called Ninf-G2. In addition, they evaluate Ninf-G2's performance using a weather forecasting
system. The evaluation of the middleware does not integrate various protocols as pub/sub QoS mechanisms and evaluate performance in this context.

Tselikis et al. [31] conduct performance analysis of a client-server e-banking application. They include three different enterprise middleware platforms each based on Java, HTTP, and Web Services technologies. The analysis of performance data led to the benefits and disadvantages of each middleware technology apart from measuring the impact of various network protocols integrated with QoS-enabled pub/sub middleware.

Performance evaluation of embedded middleware. Bellavista et al. [32] describe their work called Mobile agent-based Ubiquitous multimedia Middleware (MUM). MUM has been developed to handle the complexities of wireless hand-off management for wireless devices moving among different points of attachment to the Internet. However, this work does not focus on the performance or flexibility of QoS mechanisms in QoS-enabled anonymous pub/sub middleware.

TinyDDS [33] is an implementation of DDS specialized for the demands of wireless sensor networks (WSNs). TinyDDS defines a subset of DDS interfaces for simplicity and efficiency within the domain of WSNs. TinyDDS includes a pluggable framework for non-functional properties, e.g., event correlation and filtering mechanisms, data aggregation functionality, power-efficient routing capability. However, this work does not focus on properties of various transport protocols that can be leveraged to support QoS in pub/sub middleware.

3.3 Unresolved Challenges

Existing approaches for incorporating and evaluating QoS mechanisms for pub/sub DRE systems focus on various individual pieces of the problem. For example, some approaches focus only on a particular implementation. Other approaches focus only on components or objects which are subsets of the more generalized pub/sub paradigm. Still other approaches do not focus on QoS aspects and managing the richness of QoS-enabled pub/sub middleware for DRE systems.

The following challenges represent a gap in the current research regarding empirical evaluation of QoS mechanisms for pub/sub DRE systems:

1. Traditionally, QoS mechanisms such as transport protocols are evaluated in isolation apart from pub/sub DRE QoS concerns and outside of the context of pub/sub DRE systems. The delivered QoS of a system is dependent not only upon QoS mechanisms but incorporation of those mechanisms into the supporting system middleware. Therefore, the trade-offs of transport protocols and the QoS properties they support in various operating environments are not highlighted.

2. Pub/Sub middleware usually leverages a single or a very small handful of transport protocols (e.g., UDP for low latency and TCP for reliability). Pub/Sub DRE systems often need to address multiple QoS aspects which
can be contentious such as low latency and reliability which typically impacts latency. Therefore, the impact of the QoS properties that transport protocols support for multiple, especially contentious, pub/sub QoS concerns are not quantified.

3. Pub/sub middleware is generally not designed to easily modify transport protocol parameters or to transition from one protocol to another to ease empirical evaluation of different transport protocols as pub/sub QoS mechanisms. Moreover, pub/sub middleware lacks support for incorporating custom and novel transport protocols that can provide desirable QoS properties for pub/sub middleware in specific operating environments.

Our solution approach integrates and enhances QoS-enabled pub/sub middleware with a flexible transport protocol framework to easily support empirical evaluations of transport protocols as QoS mechanisms for pub/sub middleware. Our approach also incorporates composite QoS metrics that ease evaluation of multiple QoS concerns such as reliability and low latency. Moreover, we provide empirical results and analysis of QoS-enabled pub/sub middleware leveraging multiple transport protocols in varying operating environments.

3.4 Solution Approach: FLEXible Middleware and Transports

This section describes the structure and functionality of FLEXible Middleware And Transports (FLEXMAT), which integrates and enhances QoS-enabled pub/sub middleware with a flexible transport protocol framework. FLEXMAT also includes the ReLate2 composite QoS metrics to add in evaluating the QoS properties of transport protocols. This section also includes empirical evaluations of several transport protocols utilizing FLEXMAT and ReLate2.

3.4.1 The Structure and Functionality of FLEXMAT and ReLate2

This section presents an overview of FLEXMAT, including the OpenDDS and ANT transport protocols it uses. We then describe the ReLate2 metric created to evaluate the performance of FLEXMAT in various environment configurations to support RT-ESP application requirements for data reliability and timeliness.

Design of FLEXMAT and Its Transport Protocols

FLEXMAT integrates and enhances QoS-enabled pub/sub middleware with adaptive transport protocols to provide the flexibility needed by RT-ESP applications. FLEXMAT helps resolve Challenge 2 in Section 3.1 by providing anonymous
publication and subscription via the OMG Data Distribution Service (see Sidebar 2.4.1 for a brief summary of DDS). FLEXMAT is based on the OpenDDS implementation of DDS and incorporates several standard and custom transport protocols.

We chose OpenDDS as FLEXMAT’s DDS implementation due to its (1) open source availability, which facilitates modification and experimentation, and (2) support for a pluggable transport framework that allows RT-ESP application developers to create custom transport protocols for sending/receiving data. OpenDDS’s pluggable transport framework uses patterns (e.g., Strategy [34] and Component Configurator [35]) to provide flexibility and delegate responsibility to the protocol only when applicable.

Overview of Transport Protocols Used in FLEXMAT

OpenDDS currently provides several transport protocols. Other protocols for the FLEXMAT prototype are custom protocols (described below) that we integrated with OpenDDS using its pluggable transport framework.

**OpenDDS Transport Protocols.** By default, OpenDDS provides four transport protocols in its transport protocol framework: TCP, UDP, IP multicast (IP Mcast), and a NAK-based reliable multicast (RMcast) protocol, as shown in Figure 3.4. OpenDDS TCP is a reliable unicast protocol, whereas UDP is an unreliable unicast protocol. IP Mcast can send data to multiple receivers.

![OpenDDS and its Transport Protocol Framework](image)

Figure 3.4: **OpenDDS and its Transport Protocol Framework**

While TCP, UDP, and IP Mcast are standard protocols, RMcast warrants more description. It is a negative acknowledgment (NAK) protocol that provides reliability. For example, the sender sends four data packets, but the third data packet is not received by the receiver. The receiver realizes this packet has not been received when the fourth data packet is received. At this point the receiver sends a NAK to the sender and the sender retransmits the missing data packet. The receiver sends a unicast message to the sender for loss notification and the sender retransmits the missing data packet to the receiver.

In addition to providing reliability, the RMcast protocol orders data packets. When the protocol for a receiver detects a packet out of order it waits for the
missing packet before passing the data up to the middleware. The receiver must buffer any packets that have been received but have not yet been sent to the middleware. RMcast helps resolve Challenge 1 in Section 3.1 by providing reliability and timeliness for certain environment configurations.

**Adaptive Network Transport Protocols.**

The ANT transport protocol framework supports various transport protocol properties, including multicast, packet tracking, NAK-based reliability, ACK-based reliability, flow control, group membership, and membership fault detection. These properties can be composed dynamically at run-time to achieve greater flexibility and support adaptation.

The ANT framework originally was developed from the Ricochet [24] transport protocol. Ricochet uses a bi-modal multicast protocol and a novel type of forward error correction (FEC) called lateral error correction (LEC) to provide QoS and scalability guarantees. Ricochet supports (1) time-critical multicast for high data rates with strong probabilistic delivery guarantees and (2) low-latency error detection along with low-latency error recovery.

We included ANT’s Ricochet transport protocol, ANT’s NAKcast protocol, which is a NAK-based multicast protocol, and ANT’s baseline transport protocol in FLEXMAT. The ANT Baseline protocol mirrors the functionality of IP Mcast as described in Section 3.4.1. Using ANT’s baseline protocol helps quantify the overhead imposed by the ANT framework since similar functionality can be achieved using the OpenDDS IP Mcast pluggable transport protocol.

**Forward Error Correction (FEC).** Ricochet is based on the concepts of FEC protocols. FEC protocols are designed with reliability in mind. They anticipate data loss and proactively send redundant information to recover from this loss. Sender-based FEC protocols have the sender send redundant information, as shown in Figure 3.5. In contrast, receiver-based FEC (a.k.a. Lateral Error Correction (LEC)) have receivers send each other redundant information as shown in Figure 3.6. The Ricochet protocol we employ in FLEXMAT is an example of an LEC protocol.

![Figure 3.5: FEC Reliable Multicast Protocol - Sender-based](image1)

![Figure 3.6: FEC Reliable Multicast Protocol - Receiver-based (LEC)](image2)
**Lateral Error Correction (LEC).** LEC protocols have the same tunable $R$ and $C$ rate of fire parameters as sender-based FEC protocols. Unlike sender-based FEC protocols, however, the recovery latency depends on the transmission rate of receivers. As with gossip-based protocols, LEC protocols have receivers send out to a subset of the total number of receivers to manage scalability and network bandwidth. Moreover, the $R$ and $C$ parameters have slightly different semantics for LEC protocols than for sender-based FEC protocols.

The $R$ parameter determines the number of packets a receiver, rather than the sender, should receive before it sends out a repair packet to other receivers. The $C$ parameter determines the number of receivers that will be sent a repair packet from any single receiver. As described in Section 2.4.3, we hold the value of $C$ constant (i.e., the default value of 3) while modifying the $R$ parameter.

The Ricochet protocol helps resolve Challenge 1 in Section 3.1 by providing high probabilistic reliability and low latency error detection and recovery. Ricochet also helps resolve Challenge 4 in Section 3.1 by supporting tunable parameters that affect reliability, latency, and bandwidth usage. We designed the ANT framework so that different transport protocols can be switched dynamically.

Table 3.1 presents a summary of all protocols we included in our experiments in Section 3.4.2.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Integrator</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>OpenDDS</td>
<td>unicast, reliable, packet ordering, flow control</td>
</tr>
<tr>
<td>UDP</td>
<td>OpenDDS</td>
<td>unicast, unreliable</td>
</tr>
<tr>
<td>IP Mcast</td>
<td>OpenDDS</td>
<td>multicast, unreliable</td>
</tr>
<tr>
<td>RMcast</td>
<td>OpenDDS</td>
<td>multicast, reliable, packet ordering, NAK-based</td>
</tr>
<tr>
<td>ANT Baseline</td>
<td>ANT</td>
<td>multicast, unreliable</td>
</tr>
<tr>
<td>ANT NAKcast</td>
<td>ANT</td>
<td>multicast, reliable, NAK-based</td>
</tr>
<tr>
<td>ANT Ricochet</td>
<td>ANT</td>
<td>multicast, probabilistically reliable</td>
</tr>
</tbody>
</table>

Table 3.1: **Transport Protocols Evaluated**

**Evaluation Metric for Reliability and Latency**

We now describe considerations for evaluating FLEXMAT’s latency and reliability. We present guidelines for unacceptable percentages of packet loss for multimedia applications. We also introduce the ReLate2 metric used to evaluate FLEXMAT empirically in Section 3.4.2.

One way to evaluate the effect of transport protocols with respect to both overall latency and reliability would be simply to compare the latency times of protocols that provide reliability. Since some reliability would be provided these protocols would presumably be preferred over protocols that provide no reliability. The reliability provided by the reliable protocols in our experiments, however, deliver different percentages of reliability. Moreover, depending upon the environment configuration the average data latency between protocols differs as well. To compare results, the level of reliability must also be quantified.
For RT-ESP applications involving multimedia, such as our motivating example of SAR operations in Section 3.1, over 10% loss is generally considered unacceptable. Bai and Ito [36] limit acceptable MPEG video loss at 6% while stating that a packet loss rate of more than 5% is unacceptable for Voice over IP (VoIP) users [37]. Ngatman et al. [38] define consistent packet loss above 2% as unacceptable for videoconferencing. We use these values as guidelines to develop the ReLate2 metric that balances reliability and latency.

The 10% loss unacceptability for multimedia is due to the interdependence of packets. As shown in Figure 3.7, for example, MPEG frames are interdependent such that P frames are dependent on previous I or P frames while B frames are dependent on both preceding and succeeding I or P frames. The loss of an I or P frame therefore results in unusable dependent P and B frames, even if these frames are delivered reliably and in a timely manner.

![MPEG Frame Dependencies](image)

Figure 3.7: MPEG Frame Dependencies

We conservatively state that a 10% packet loss should result in an order of magnitude increase in any metric value generated. We therefore developed our ReLate2 metric to multiply the average latency by the percent packet loss as follows:

\[ \text{ReLate}_2 = \frac{\sum_{i=1}^{r} l_i}{r} \times \left( \frac{t - r}{t} \times 100 + 1 \right) \]

where \( p \) is the protocol being evaluated,
\( r \) = number of packets received,
\( l_i \) = latency of packet \( i \),
and \( t \) = total number of packets sent.

We add 1 to the percent packet loss to normalize for any loss less than 1% where the metric would otherwise yield a value lower than the average latency, specifically the value 0 where all packets are delivered. This adjustment produces a ReLate2 value equal to the average latency when there is no packet loss which still accommodates meaningful comparisons for protocols that deliver all packets. Section 3.4.2 uses the ReLate2 metric to determine the transport protocols that best balance reliability and latency.
3.4.2 Experimental Setup, Results, and Analysis

The section presents the results of experiments we conducted to determine the performance of FLEXMAT in a representative RT-ESP environment. The experiments include FLEXMAT using multiple transport protocols with varying numbers of receivers, percentage data loss, and sending rates as would be expected with SAR operations in a dynamic environment as described in Section 3.1.

Experimental Setup

We conducted our experiments using two network testbeds: (1) the Emulab network emulation testbed and (2) the ISISlab network emulation testbed. Emulab provides computing platforms and network resources that can be easily configured with the desired computing platform, OS, network topology, and network traffic shaping. ISISlab uses Emulab software and provides much of the same functionality, but does not (yet) support traffic shaping. We used Emulab due to its ability to shape network traffic and ISISlab due to the availability of computing platforms.

As outlined in Section 3.1, we are concerned with the distribution of data for SAR datacenters, where network packets are dropped at end hosts [39]. The Emulab network links for the receiving data readers were configured appropriately for the specified percentage loss. The experiments in ISISlab were conducted with modified source code to drop packets when received by data readers since ISISlab does not yet support network traffic shaping.

The Emulab network traffic shaping was mainly needed when using TCP. OpenDDS does not support programmatically dropping a percentage of packets in end hosts for TCP. We therefore used network traffic shaping for TCP which only Emulab provides.

Using the Emulab environment and the ReLate2 metric defined in Section 3.4.1, we next determined the protocols that balanced latency and reliability well, namely RMcast, ANT NAKcast, and ANT Ricochet. Since we could programmatically control the loss of network packets at the receiving end hosts with these protocols, we then used ISISlab due to its availability of nodes to conduct more detailed experiments involving these protocols. We obtained up to 27 nodes fairly easily using ISISlab, whereas this number of nodes was hard to get with Emulab since it is often oversubscribed.

Our experiments using Emulab and ISISlab used the following traffic generation configuration utilizing OpenDDS version 1.2.1: (1) one DDS data writer wrote data, variable number of DDS data readers read data, (2) the data writer and each data reader ran on its own computing platform, and (3) the data writer sent 12 bytes of data 20,000 times at a specified sending rate. To account for experiment variations we ran 5 experiments for each configuration, e.g., 5 receiving data writers, 50 Hz sending rate, 2% end host packet loss. We used Ricochet’s default $C$ value of 3 for both Emulab and ISISlab experiments.

**Emulab configuration.** For Emulab, the data update rates were 25 Hz and 50 Hz for general comparison of all the protocols. We varied the number of
receivers from 3 up to 10. We used Ricochet’s default $R$ value of 8. As defined in Section 3.4.1, the $R$ value is the number of packets received before sending out recovery data.

We used the Emulab pc850 hardware platform, which includes an 850 MHz processor and 256 MB of RAM. We ran the Fedora Core 6 operating system with real-time extensions on this hardware platform, using experiments consisting of between 5 and 12 pc850 nodes. The nodes were all configured in a LAN configuration. We utilized the traffic shaping feature of Emulab to run experiments with network loss percentages between 0 and 3 percent. Table 3.2 outlines the points of variability for the Emulab experiments.

<table>
<thead>
<tr>
<th>Point of Variability</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of receiving data writers</td>
<td>3 - 10</td>
</tr>
<tr>
<td>Frequency of sending data</td>
<td>25 Hz, 50 Hz</td>
</tr>
<tr>
<td>Percent end-host network loss</td>
<td>0 to 3 %</td>
</tr>
</tbody>
</table>

Table 3.2: Emulab Variables

ISISlab configuration. We used ISISlab for experiments involving transport protocols where we could programmatically affect the loss of packets in the end hosts. By modifying the source code, we could discard packets based on the desired percentage. In particular, we focused the ISISlab experiments on the ANT NAKcast and Ricochet protocols since from the initial experiments these protocols showed the ability to balance latency and reliability. At times, OpenDDS RMcast showed the ability to balance reliability and low latency. Since its behavior was erratic for a NAK-based protocol, however, we excluded it from the detailed experiments. Table 3.3 outlines the points of variability for the ISISlab experiments.

ISISlab provides a single type of hardware platform: the pc8832 hardware platform with a dual 2.8 GHz processor and 2 GB of RAM. We used the same Fedora Core 6 OS with real-time extensions as for Emulab. We ran experiments using between 5 and 27 computing nodes which map to between 3 and 25 data readers respectively. All nodes were configured in a LAN as was done for Emulab. We ran experiments using Ricochet’s $R$ value of 8 and 4, as explained in Section 3.4.2.
Results and Analysis of Experiments

This section presents and analyzes the results from our experiments, which resolves Challenge 3 in Section 3.1 by characterizing the performance of the transport protocols for various environment configurations.

The Baseline Emulab Experiments.

The initial set of experiments for the FLEXMAT prototype included all the OpenDDS protocols as enumerated in Section 3.4.1. These experiments used Emulab as described in Section 3.4.2. Our baseline experiments used 3 data readers, 0% loss, and 25 and 50 Hz update rates. As expected, all protocols delivered all data to all data readers, i.e., 3 receivers * 20,000 updates = 60,000 updates.

As shown in Figures 3.8 and 3.9, the latency at times was lowest with protocols that do not provide any reliability, i.e., OpenDDS UDP, OpenDDS IP Mcast, and ANT Baseline). The OpenDDS RMcast and ANT Ricochet protocols were the only ones that never produced the lowest overall average latency. As expected, average latency times decreased as the sending rate increased from 25 Hz to 50 Hz.

The next set of experiments added 1% network packet loss for the receiving end hosts. We do not include figures for the 50 Hz update rate as the data are comparable to that seen with a sending rate of 25 Hz. As shown in Figure 3.10, there is a clear delineation between the protocols that provide reliability and those that do not.

TCP received all updates sent, whereas ANT NAKcast and ANT Ricochet received a high percentage of updates with ANT NAKcast receiving all updates except for one experiment run where it received 59,999 out of the 60,000 updates. Both configurations of ANT Ricochet delivered a consistently high percentage of updates between 99.95% and 99.99%. UDP, IP Mcast, and ANT Baseline group together in the figure with low reliability.

We were unable to configure OpenDDS IP Mcast to use Emulab’s network traffic shaping. Instead we calculated the amount of packet loss that is comparable to the other unreliable transports, i.e., 1% loss. We are confident this...
calculation does not invalidate the values seen and used for OpenDDS IP Mcast as the values for ANT’s version of IP Mcast, i.e., ANT Baseline, produces similar results.

Figure 3.11 shows the erratic behavior of RMcast. At times RMcast received all updates and other times it received all updates only up to a certain number and then received no additional updates. The cause of this problem was not explained by the RMcast developers. We therefore removed RMcast from further consideration.

Figure 3.12 highlights the latency overhead incurred by TCP. This latency is due to TCP’s use of positive acknowledgments. Moreover, TCP’s latency overhead increases as the amount of loss increases. All other protocols are fairly comparable with respect to latency for this environment configuration.

Figure 3.13 shows the ReLate2 values for all the protocols considered. We see that using ReLate2 splits the protocols that support both reliability and low latency from those that do not. The separation of the protocols using ReLate2 is more pronounced with higher levels of network loss and number of receivers.

For 1% network loss, TCP and NAKcast deliver all the packets for every experiment. OpenDDS Reliable Mcast delivers all the packets for some experiments but not all. ANT Ricochet always delivers the second most highest number of updates with the percentage delivered being between 99.5% and 99.6%.

We now analyze the results of the Emulab experiments, which involved all the transport protocols presented in Section 3.4.1. We utilize the ReLate2 metrics defined in Section 3.4.1 to evaluate the results from the initial Emulab experiments. The results show that ANT NAKcast and ANT Ricochet always produced the lowest ReLate2 values even for multiple configurations of the protocols, i.e., NAKcast timeout values of 0.05 and 0.025 and Ricochet R values of 4 and 8. The protocols that support reliability but unbounded latency and the protocols that support low latency but no reliability are clearly separated from the protocols that support both low latency and reliability.

Moreover, the ReLate2 value is equal to the average latency when there is no
Figure 3.12: Emulab: 3 readers, 1% loss, 25Hz

Figure 3.13: Emulab: 3 readers, 1% loss, 25Hz

Figure 3.14: Emulab: 3 readers, 3% loss, 25Hz

Figure 3.15: Emulab: 3 readers, 3% loss, 50Hz

loss, as is the case for TCP and the majority of cases for NAKcast. When NAKcast does not receive all updates, it is only missing some of the very last updates which could not be detected since no packets were received after them. The data and figures show that the ReLate2 metric is useful for evaluating protocols that balance reliability and latency.

We compare the values from the ReLate2 metric as shown in Figure 3.18 with the values in Figure 3.17 which were only based on the original Relate metric. The results show that OpenDDS RMcast and ANT Ricochet always produce the lowest ReLate2 value. Moreover, when there is no loss, the ReLate2
value is equal to the average latency as is the case for TCP. This comparison shows that the ReLate2 metric is useful for evaluating protocols that balance reliability and latency.

The NAKcast and Ricochet Experiments.

Our next set of experiments focused on the protocols that are best suited for balancing reliability and latency based on the ReLate2 metric (i.e., ANT NAKcast and ANT Ricochet). We focus on these protocols for comparison to gain a better understanding of trade-offs between them. We provide experimental results and analyze the results. We note that if RMcast’s behavior would stabilize it would also be a protocol worth evaluating for reliability and low latency.

In particular, for comparison we focused on specific configurations of NAKcast and Ricochet, i.e., NAKcast with a timeout period of 0.05 seconds and Ricochet with an $R$ value of 4. We constrained the protocols in this way because configured correctly either protocol can generally provide lower ReLate2 values than the other. However, we are interested in a relative comparison of the protocols themselves rather than reconfigurations that can make the one protocol outperform the other for a particular environment.

As noted in Section 3.4.2, we used the ISISlab testbed for experiments involving only ANT NAKcast and ANT Ricochet due to the availability of a larger number of hardware nodes. We were able programmatically to induce packet loss at the end hosts for these two protocols since the ANT source code is avail-
As with the Emulab experiments in Section 3.4.2, we began with experiments where the number of receivers and packet loss were low. We also expanded the sending rates to include 10Hz and 100Hz along with the original rates of 25Hz and 50Hz. Adding sending rates made sense as the packet loss recovery times for both of these protocols are sensitive to the update rate.

The packet loss recovery time for NAKcast is sensitive to the update rate since loss is only discovered when packets are received. If packets are received faster then packet loss is discovered sooner and recovery packets can be requested, received, and processed sooner. Likewise, the packet loss recovery time for Ricochet is sensitive to the update rate since recovery data is only sent out after R packets have been received. When packets are received sooner, recovery data is sent, received, and processed sooner.

Moreover, our results and analysis are focused on environment configurations with relatively low (i.e., 1%) and high (i.e., 5%) network loss combined with relatively few (i.e., 3) and many (i.e., 20) receivers. While we ran experiments that ran the spectrum of configurations between these bounds, the particular experiments at these limits are useful for understanding the behavior of the protocols. We show data collected while using 10Hz and 100Hz sending rates to highlight the behavioral distinctions of the protocols.

Figures 3.19 and 3.20 show that for a low number of receivers (i.e., 3), a low loss percentage (i.e., 1%), and low sending rate (i.e., 10Hz), NAKcast, in general, has lower ReLate2 values. In fact, NAKcast 0.05 provided the lowest ReLate2 values for all of the ISISlab protocol configurations tried, i.e., NAKcast with timeout values of 0.05 and 0.025 seconds and Ricochet with R values of 4 and 8. Ricochet provided lower average update latency as the sending rate increases. We discuss this observation in more detail at the end of this section. The number of updates received remains constant across various sending rates for both protocols and we do not include those figures here.

Figures 3.19 and 3.20 also show the reliability of Ricochet at low loss rates. This reliability can be seen by comparing the figures and noticing that the graphs appear very similar. This similarity points out that Ricochet is almost as reliable

---

**Figure 3.18: ReLate2 Metrics for Emulab Experiment: 3 readers, 3% loss, 50 Hz update rate**

---

44
as NAKcast with reliability rates ranging from 99.97% to 99.99%. This reliability is fairly constant across the different sending rates.

Figures 3.21 and 3.22 show the effect on the protocols of increasing packet loss. In this environment configuration we have changed the network loss from 1% to 5%. We see that NAKcast performed best not only for a sending rate of 10 Hz as was the case for 1% loss but also for 25 Hz. Ricochet still provided the best ReLate2 values for sending rates of 50 Hz and 100 Hz. Moreover, while Ricochet average update latency improved over NAKcast the ReLate2 values don’t reflect this as Ricochet only had better ReLate2 values for sending rates of 50 and 100 Hz. This is due to Ricochet’s reliability ranging from 99.42% to 99.56% which has decreased from the experiments with 1% loss.

Figures 3.23 and 3.24 show the effect on the protocols of increasing the number of receivers. In this environment configuration we increased the number of receivers from 3 to 20. We see that now Ricochet and NAKcast performed equally well at 10 Hz where NAKcast always performed best at that rate
with only 3 receivers. Ricochet provided the best ReLate2 values for the other sending rates. Moreover, Ricochet’s reliability is almost as high as with only 3 receivers ranging from 99.94% to 99.96% of updates received.

Finally, Figures 3.25 and 3.26 show the effect on the protocols of increasing the number of receivers and loss rate. In this environment configuration we had 20 receivers and 5% network loss.

We see that while Ricochet had a noticeable improvement in average update latency compared to NAKcast, NAKcast offset this discrepancy with its higher reliability. For higher rates, i.e., 25, 50, and 100 Hz, the ReLate2 values for Ricochet and NAKcast are comparable. NAKcast always provided the lowest ReLate2 values for 10 and 25 Hz while Ricochet always provided the lowest ReLate2 values for 50 and 100 Hz. Moreover, Ricochet’s reliability is in the same range as for 3 receivers with 5% loss ranging from 99.46% to 99.55% of updates received.

The results above show that for a set protocol configuration there are perfor-
mance trade-offs between NAK-based and LEC protocols. In general, NAK-based protocols performed better with a lower network loss percentage, lower sending rates, and few receivers. In this environment configuration there is no concern for NAK storms where receivers flood the sender with requests for retransmissions. Moreover, NAK-based protocols only needed to receive one update that is out of sequence to determine loss whereas LEC protocols need to receive $R$ updates before error detection and correction information is sent among the receivers. NAK-based protocols also delivered consistently high reliability, at the cost of higher latency for higher sending rates.

LEC protocols, however, provided better performance when network loss was higher and sending rates increased. LEC protocols did not incur increasingly more network usage as network loss and number of receivers increased. LEC protocols scaled well in the number of receivers and in network loss. LEC protocols also generally provided lower latency at the cost of small decreases in reliability.

NAKcast 0.05 provided the lowest ReLate2 values and lowest average latency for 3 receivers, 1% loss, and 10 Hz sending rate. The data make sense since the sending rate was less than the timeout period and the loss rate and number of receivers were low. If the network drops a packet the packet is as likely to be discovered in the same amount of time by NAKcast with a timeout of 0.05 as it is with a higher timeout. The sending rate is so low that increasing the NAKcast timeout to 0.025 seconds provided no benefit and indeed added overhead as timeouts are generated and checked more frequently.

3.5 Lessons Learned

Developers of RT-ESP systems face a number of challenges when developing their applications for dynamic environments. To address these challenges, we have developed FLEXMAT to integrate and enhance QoS-enabled pub/sub middleware with flexible transport protocols to support RT-ESP applications. This paper defines the ReLate2 metric to empirically measure the reliability and latency of FLEXMAT as a first step to having QoS-enabled pub/sub middleware autonomically adapt transport protocols as the changing environment dictates.

The following is a summary of lessons learned from our experience evaluating FLEXMAT’s performance with various transport protocols:

- Exploring a configuration space for trade-offs requires a disciplined approach with analysis to guide the exploration. Depending on the number of dimensions involved in the search space there can be many configurations to explore. In our case, we had multiple variables, e.g., update rate, % loss, number of data readers, NAKcast’s timeout value, and Ricochet’s $R$ value. Since the number of potential experiments was large, we found it helpful to make coarse-grained adjustments for initial experiments. We would then analyze the results to guide areas of refinement to find trade-offs between transport protocols. For example, varying Ricochet’s $R$ value (see Section 3.4.2) occurred as a result of analyzing early experimental results.
Integrating pub/sub middleware with transport protocols exacerbates the challenge of pinpointing the source of problems and anomalies. Certain experiments incurred unexpected behavior, such as RMcast at times only providing a small percentage of updates. With the integration of middleware and transport protocols, determining where deficiencies lie can be hard since problems could be in the middleware, the protocol, or the combination of both. In addition to individually testing protocols and the middleware, therefore, it was helpful to compare the anomalous behavior of a protocol with other protocols keeping the same configuration environment. For example, Section 3.4.2 described how we used these comparisons to determine unexpected behavior coming from RMcast rather than the OpenDDS transport protocol framework or pub/sub middleware.

The manual integration of QoS with pub/sub middleware and transport protocols is tedious and error-prone. Currently, pub/sub middleware and transport protocols integrators must manually manage QoS properties specified in the middleware with QoS properties provided by a transport protocol. For example, an integrator could mistakenly select a transport protocol with no reliability support even though application developers specified reliable communication. The middleware does not help in determining the mismatch between QoS properties and transport protocol properties. Our future work is investigating ways to manage this complexity via domain-specific modeling languages (DSMLs) that provide profiles for certain types of applications, such as RT-ESP applications. Once a profile is selected, the DSML could automatically generate correct implementation artifacts for the application.

High-level metrics are useful to quickly differentiate the performance of various configurations. The use of metrics—even if coarse-grained—helps explore a large configuration space. Part of the impetus in developing the ReLate and ReLate2 metrics (see Section 3.4.1) is to ameliorate navigating a configuration space with several points of variability.

Specifying unacceptable loss for RT-ESP is hard to generalize. The amount of acceptable loss is specific to a particular application or application type. However, a general acceptability guideline of 10% loss or less for multimedia applications has been helpful in making initial evaluations of protocols that balance reliability and latency. Additional composite metrics would be helpful for measuring and evaluating additional areas of interest, e.g., jitter and network bandwidth usage. We plan to fine tune ReLate2 and develop additional metrics as needed.

Flexible transport protocols make manual management and tuning of the protocols hard. Our experiments show the flexibility of the NAKcast and Ricochet transport protocols. Modifying NAKcast’s timeout value and Ricochet’s R value affects the average overall latency, as shown by our results in Section 3.4.2. Likewise, the modification of Ricochet’s C value can affect the percentage of recovered packets with a corresponding impact on bandwidth. Keeping protocol parameter settings optimized in a turbulent environment can quickly become overwhelming if done manually. Reaction time needed can swiftly surpass those of humans. We are researching the use of machine learning...
to automatically adjust parameter settings appropriately based on the environment and the QoS specified by the application. We anticipate our experimental data to be used for supervised machine learning to dynamically optimize parameter settings.

- **Multicast with NAK-based reliability and LEC protocols balance reliability and latency.** After conducting the experiments and using our ReLate2 metric we determined that when combining low latency and reliability, multicast with NAK-based reliability and LEC protocols deliver the best performance. NAK-based protocols have fairly low overhead and low bandwidth usage for low loss rates since only the detected loss of a packet triggers recovery actions. Moreover, we found that Ricochet is consistently reliable with a high probability. Ricochet also provides consistent bandwidth usage for $R$ and $C$ settings which can be important for network constrained environments.

The latest information and source-code for FLEXMAT and related research can be obtained at [www.dre.vanderbilt.edu/~jhoffert/FLEXMAT](http://www.dre.vanderbilt.edu/~jhoffert/FLEXMAT).
Chapter 4

Autonomic Adaptation of QoS-enabled Pub/Sub Middleware

In Chapter 1, an overview of the need for autonomic adaptation of QoS-enabled pub/sub DRE middleware was presented. This chapter first presents an overview of existing research in the field of autonomic adaptation of QoS-enabled pub/sub middleware. Next, this chapter enumerates and illustrates unresolved challenges. Finally, this chapter describes the solution approach to resolving the challenges identified in Section 1.2.

4.1 Motivating Example: Ambient Assisted Living in Smart Cities

With an aging population that is quickly outpacing the rate of new care providers for the elderly technology is needed to aid the current care givers and to enhance the independence and quality of life for the aging. Smart home technology is an area that has been researched in recent years which can in part aid care givers. [40–42] However, technology centered on a localized area such as a home does not provide the independence that many elderly people desire and need to maintain or increase quality of living.

The goal of Ambient Assisted Living (AAL) is to prolong and enhance independent living for the elderly. [43] The goal of smart cities (SC) is to dissolve the computational infrastructure and establish ubiquitous, context-aware services in metropolitan areas. [44] Combining the technologies of AAL and SC yields a Smart City Ambient Assisted Living (SCAAL) environment where senior citizens can increase their independence, mobility, and autonomy while care givers for the aged are aided and empowered to adequately support the number of people under their supervision.
Figure 4.1 shows an example AAL-SC scenario where an elderly person is independently navigating in a large metropolitan area. The person is equipped with multiple technological devices that aid in various aspects of the person’s mobility, sensory enhancement, communication, and monitoring. A aged person would normally be overwhelmed with controlling and responding to all the individual equipment. However, coordination among the devices is supported by supplemental equipment.

In particular, a personal data center (PDC) publishes and subscribes to the data that is being managed by the personal devices and interfaces with the smart city by publishing and subscribing to data from the ambient environment. Additionally, coordinating and managing equipment can subscribe to the data being managed by the PDC to filter, fuse, and coalesce the flood of data available so that the elderly person only needs to be contacted when necessary and can be notified in a simple and clear manner.

The environment in which the PDC operates is very dynamic as (1) the elderly person is moving through space in the smart city and updating personal information in time and (2) as the smart city enhances and updates the amount and kind of data that it provides as it moves through time. Our research focuses on the QoS properties that a PDC device must manage in a SCAAL application. To support multiple QoS aspects of the ambient data such as emergency responder update information and personal data such as 3-dimensional health monitoring information the PDC presents the following challenges:
Challenge 1: Timely Adaptation to Dynamic Environments  Due to the dynamic environment inherent in SCAAL applications, the PDC must adjust in a timely manner as the environment changes. If the PDC cannot adjust quickly enough it will fail to perform adequately given a shift in data load and requirements. As the types and the amount of data relevant to the SCAAL application fluctuates and the demand for information varies the PDC must be configured to accommodate these changes with appropriate responsiveness to maintain a minimum level of service. Manual modification is not feasible for SCAAL applications as the elderly person will not possess the technical ability to make modifications. Moreover, manual modifications even by a sufficient trained technical person can be too slow and error-prone to support QoS especially for critical data such health ad safety monitoring.

Challenge 2: Managing Interacting QoS Requirements  The PDC must manage multiple QoS requirements that interact with each other, e.g., data reliability so that enough data is received to be useful and low latency for soft realtime data so that detailed 3-dimensional health monitoring information or video from personal surveillance cameras do not arrive after they are needed. The streamed data must be received soon enough so that successive dependent data can be used as well. For example, MPEG I frame data must be received in a timely manner so that successive dependent B and P frame data can be used before the next I frame makes them obsolete. Otherwise, not only is the data unnecessary, but sending and processing the data has consumed limited resources.

Challenge 3: Scaling to Large Numbers of Senders and Receivers  Within the environment of a smart city, a multitude of organizations such as those relevant to healthcare and safety would not only register interest in the individual’s video and health monitoring scans, but also would supply data to the SCAAL application. For example, hospitals and doctors would not only monitor health information from the elderly individual but would also send out location information for the closest EMS staff. Likewise, police and security forces would not only monitor safety alerts from the SCAAL application in cases of harrassment, stalking, or robbery but would also disseminate information on locations to avoid due to large traffic accidents or current unsafe city locations.

Challenge 4: Specifying Standardized and Robust QoS  SCAAL applications in general and the PDC in particular should be developed with the focus on application logic rather than on complex or custom formats for specifying QoS. Time spent learning a customized or complex format for QoS is time taken from developing the SAR application itself. Moreover, learning a custom format will not be applicable for other applications that use a different QoS format. Application developers also need support for a wide range of QoS to handle dynamic environments.
4.2 Related Research

**Support for adaptive middleware.** The Mobility Support Service (MSS) [45] provides a software layer on top of pub/sub middleware to enable endhost mobility. The purpose of MSS is to support the movement of clients between access points of a system using pub/sub middleware. In this sense, MSS adapts the pub/sub middleware used in a mobile environment. Mobile clients notify MSS when mobility starts and ends. MSS buffers messages and manages connections while the client moves to a different access point. MSS is designed to support multiple pub/sub technologies, e.g., implementations of JMS, and adapt to the technology-specific characteristics.

MSS is solely focused on supporting mobility of pub/sub, however, and therefore does not address the challenge of managing contentious QoS Requirements (i.e., Challenge 2 in Section 4.1). Moreover, MSS fails to address the challenge of specifying standardized and robust QoS (i.e., Challenge 4 in Section 4.1) since it does not present a standardized and robust interface for QoS.

Gridkit [46] is a middleware framework that supports reconfigurability of applications dependent upon the condition of the environment and the functionality of registered components. Gridkit focuses on grid applications which are highly heterogeneous in nature. For example, these applications will run on many types of computing devices and across different types of networks.

To register components, application developers use Gridkit’s API which is based on binding contracts. Gridkit then uses the contract information along with a context engine to determine which components to include in the application. The context engine takes into account the context of the host machines, e.g., battery life, network connectivity.

Gridkit focuses on reconfiguration for installing an application and does not address the challenge of timely adaptation. (i.e., Challenge 1 in Section 4.1). Within Gridkit no consideration is given to making timely adaptations based on the environment changing for a single application installation. Moreover, Gridkit fails to address the challenge of specifying standard and robust QoS. (i.e., Challenge 4 in Section 4.1) as it provides no standardized QoS specification.

David and Ledoux have developed SAFRAN [47] to enable applications to become context-aware themselves so that they can adapt to their contexts. SAFRAN provides reactive adaptation policy infrastructure for components using an aspect-oriented approach. SAFRAN follows the structure of a generic AOP system by supporting (1) a base program which corresponds to a configuration of components, (2) point-cuts which are invoked in response to internal events (e.g., invocations on interfaces) and external events (e.g., change in system resources), (3) advices which define functionality to be executed for pointcuts, and (4) adaptation which uses adaptation policies to link join points to advices.

The SAFRAN component framework, however, only provides development support of maintaining specified QoS. The adaptive policies and component implementation are the responsibility of the application developer. Moreover, SAFRAN does not specifically address the challenge of scaling to large numbers
of receivers (i.e., Challenge 3 in Section 4.1) since it does not focus on scalability. SAFRAN also does not address the challenge of specifying standard and robust QoS (i.e., Challenge 4 in Section 4.1) since it provides no standard QoS specification.

**Machine learning in support of autonomic adaptation.** Vienne and Sourrouille [48] present the Dynamic Control of Behavior based on Learning (DCBL) middleware that incorporates reinforcement machine learning in support of autonomic control for QoS management. Reinforcement machine learning not only allows DCBL to handle unexpected changes but also reduces the overall system knowledge required by the system developers. System developers provide an XML description of the system, which DCBL then uses together with an internal representation of the managed system to select appropriate QoS dynamically.

DCBL’s customized QoS specification, however, does not address the challenge of specifying standard and robust QoS (i.e., Challenge 4 in Section 4.1) and DCBL focuses on single computers rather than addressing scalable distributed systems, as outlined with the challenge of scaling to large number of receivers. (i.e., Challenge 3 in Section 4.1). Moreover, DCBL requires developers to specify in an XML file the selection of operating modes given a QoS level along with execution paths, which leaves handling the challenge of managing contentious QoS requirements (i.e., Challenge 2 in Section 4.1) to developers.

Tock et al [49] utilize machine learning for data dissemination in their work on Multicast Mapping (MCM). MCM hierarchically clusters data flows so that multiple topics are mapped onto a single session and multiple sessions are mapped onto a single reliable multicast group. MCM’s approach manages the scarce availability of multicast addresses in large-scale systems. MCM leverages machine learning to adapt as user interest and message rate change during the day. MCM is just designed to address the scarce resource of IP multicast addresses in large-scale systems, however, rather than the challenges of managing contentious QoS requirements and specifying standardized and robust QoS (i.e., Challenge 2 in Section 4.1 or Challenge 4 in Section 4.1).

**Infrastructure for autonomic computing.** Grace et al. [50] describe an architecture metamodel for adapting components that implement coordination for reflective middleware distributed across peer devices. This work also investigates supporting reconfiguration types in various environmental conditions. The proposed architecture metamodel, however, only provides proposed infrastructure for autonomic adaptation and reconfiguration and does not directly address the challenges in Section 4.1.

Valetto et al. [51] developed network features in support of service awareness to enable autonomic behavior. Their work targets communication services within a Session Initiation Protocol (SIP) enabled network to communicate monitoring, deployment, and advertising information. As an autonomic computing infrastructure, however, this work does not directly address any of the challenges in Section 4.1.

**Autonomic adaption of service level agreements.** Herssens et al. [52] describe work that centers around autonomically adapting service level agree-
ments (SLAs) when the context of the specified service changes. This work acknowledges that both offered and the requested QoS for Web services might vary over the course of the interaction and accordingly modifies the SLA between the client and the server as appropriate. This work does not address the challenge of timely adaptation in dynamic environments (i.e., Challenge 1 in Section 4.1), but rather negotiates the QoS agreement to fit the dynamic environment.

**Autonomic adaption of networks.** The Autonomic Real-time Multicast Distribution System (ARMDS) [53] is a framework that focuses on decreasing excessive variance in service quality for multicast data across the Internet. The framework supports the autonomic adaptation of the network nodes forming the multicast graph so that the consistency of service delivery is enhanced. The framework includes (1) high level descriptions of policies and objectives, (2) a multicast topology management protocol supported by network nodes, (3) measurement and monitoring infrastructure, and (4) a control component that autonomously manipulates the protocol and infrastructure to reduce variance. However, ARMDS does not address the challenge of managing contentious QoS requirements or specifying standardized and robust QoS ((i.e., Challenge 2 in Section 4.1 and Challenge 4 in Section 4.1).

### 4.3 Unresolved Challenges

Existing approaches applicable to autonomically adapting QoS mechanisms for pub/sub DRE systems in dynamic environments focus only on individual pieces of the problem. For example, some approaches focus only on checking QoS status at run-time. Other approaches provide a framework for developing adaptation applications but don’t provide implementations themselves. Still other approaches provide adaptation but are not concerned with QoS properties relevant to pub/sub DRE systems such as hard and soft real-timeliness and bounded latencies.

The following challenges taken together represent a gap in the current research regarding autonomic adaptation of QoS mechanisms for pub/sub DRE systems operating in dynamic environments:

1. A pub/sub DRE system needs to monitor QoS to know if QoS is not being met. Since the system is deployed in a dynamic operating environment QoS mechanisms that provided sufficient QoS initially might no longer be adequate. Only ongoing monitoring of the QoS of the system will provide the current status of QoS. However, while monitoring is necessary by itself it is not sufficient for autonomic adaptation.

2. Once QoS monitoring has determined that QoS is not being met the pub/sub DRE system must then reconfigure the QoS mechanisms appropriately. For safety or mission critical DRE pub/sub systems reconfiguration of the QoS mechanisms must occur while the system is running. These kinds of system can not afford the lapse in support that a shutdown and restart
would incur. However, while dynamic reconfiguration of the QoS mechanisms during system execution is necessary for autonomic adaptation it is not sufficient for DRE pub/sub systems which require real-timeliness and bounded latency times.

Our solution approach integrates and enhances (1) QoS-enabled pub/sub middleware, (2) a flexible transport protocol framework to provide appropriate transport protocol behavior, (3) machine learning techniques to determine optimal transport protocol and protocol settings, (4) a monitoring infrastructure to determine when specified QoS is and is not being met, and (5) a controller to manage monitoring, protocol determination, and migration from one protocol to another.

4.4 Proposed Solution Approach: ADAptive Middleware and Network Transports

This section describes the structure and functionality of the proposed solution approach of ADAptive Middleware And Network Transports (ADAMANT), which integrates and enhances QoS-enabled pub/sub middleware, a flexible transport protocol framework, machine learning techniques, monitoring infrastructure, and a controller. This section also includes specific challenges that ADAMANT will need to address as a proposed solution.

4.4.1 Overview of ADAptive Middleware and Network Transports

Our solution approach combines and enhances the following technologies to resolve the challenges presented in Section 4.1 The architecture of the proposed ADAMANT solution approach is illustrated in Figure 4.2.

- Standard QoS-enabled pub/sub middleware as described for FLEXMAT in Chapter 3. Anonymous pub/sub middleware addresses the scalability of Challenge 3 in Section 4.1 by decoupling data senders from data receivers. Standard QoS-enabled middleware also addresses the QoS standardization of Challenge 4 in Section 4.1.

- The Adaptive Network Transports (ANT) framework as described for FLEXMAT in Chapter 3 helps address the timely adaptation of Challenge 1 in Section 4.1.

- Supervised machine learning helps address Challenge 1 in Section 4.1 and Challenge 2 in Section 4.1 by selecting an appropriate transport protocol and protocol parameters in a timely manner given a specified QoS and a particular environment configuration. The machine learning component includes features for several different environment configurations and supervised training techniques, such as decision trees, multilayer perceptrons, and support vector
machines, to learn the correct protocol and parameters. The machine learning interpolates and extrapolates its learning based on the current environment configuration, which may not have been included in the supervised training.

- Environment monitoring helps address Challenge 1 in Section 4.1 by providing environment configuration information. Relevant environment configuration values are monitored as needed such as the number of subscribers, the percentage of network packet loss, and the sending rate of the data. These monitored values are input to the machine learning component to determine an appropriate network transport and accompanying parameters.

- Autonomic adaptation helps address Challenge 1 in Section 4.1 and Challenge 2 in Section 4.1 by (1) querying relevant values from the environment monitoring, (2) activating the machine learning component which will determine an appropriate transport protocol and parameters, (3) retrieving the recommended protocol settings, and (4) transitioning the adaptive network transports to use the recommended settings.

### 4.4.2 Specific Challenges for the Proposed ADAMANT Solution Approach

While ADAMANT proposes addressing all the existing challenges that have been mentioned, ADAMANT has particular challenges it must address due to the nature of DRE systems and the technologies ADAMANT is including. Specifically, the proposed ADAMANT approach must justify the use of machine learning techniques over other approaches (e.g., policy-based, reinforcement learning) available for determining appropriate transport protocol and protocol settings.
for a given operating environment.

Policy-based approaches provide a relatively straightforward way to guide the properties of systems. When certain operating conditions are checked and met then the system is imperatively directed by the policies to alter its behavior. Policy-based approaches can be optimized since the bounded number of both conditions that are checked and the behaviors used to direct the system are explicitly identified. For example, case or switch statements in an imperative programming language such as C++ can be used to implement policy-based approaches. These switch statements can then be optimized to constant time performance.

However, policy-based approaches do not provide robustness in the face of conditions not considered a priori. Policy-based approaches must have complete knowledge of all conditions that can affect the system so that this knowledge can be imperatively codified. If conditions exist that were not anticipated then unexpected system behavior can occur which can be disastrous for mission-critical pub/sub DRE systems. Even when all relevant conditions are considered and all appropriate responses are codified, managing the conditions and responses increases accidental complexity.

Unsupervised machine learning approaches provide robustness and flexibility when not all conditions and appropriate system responses are known a priori. For reinforcement learning in particular, certain system behaviors can be set as goals and positive or negative reinforcements can guide the resolution of system behavior as changes in an operating environment occur. Alternatively, even when all conditions of the operating environment are known and all appropriate responses determined, unsupervised machine learning can be used to manage the accidental complexity of conditions and appropriate responses. However, in general unsupervised machine learning is unbounded in its determination of an appropriate response due to online exploration of the solution space and backtracking when branches of the solution space are determined to be no longer viable. As indicated in the work of Bu et al. [54], reinforcement learning benefits from an additional initialization period before system startup to improve performance.

We propose using supervised machine learning for ADAMANT for DRE systems in dynamic environments to address both the timeliness and robustness concerns. Our hypothesis is that supervised machine learning techniques are robust when not all conditions of dynamic environments are known a priori. Supervised machine learning techniques provide interpolation and extrapolation of known training sets to handle conditions for which the techniques have not been trained. We propose to show that supervised machine learning techniques can provide ADAMANT with appropriate transport protocol guidance for operating environments for which the techniques have not been trained.

Our hypothesis also includes the premise that supervised machine learning techniques provide timely and bounded response. Techniques such as artificial neural networks can be configured in the number of layers and the amount of processing at each layer between the input of operating environment conditions and the output of an appropriate transport protocol and settings. [55].
This configuration allows worst-case execution time analysis to be conducted to determine bounded response times. We propose to show that supervised machine learning techniques provide timely and bounded response for determining appropriate transport protocols and parameters for operating conditions in dynamic environments.

Due to real-timeliness concerns of DRE systems ADAMANT must show that the autonomic adaptation occurs in a timely fashion and that adaptation times are bounded. Bounded adaptation times have implications on ADAMANT in the following areas:

- **ADAMANT must provide a timely determination of current QoS state.** If monitoring takes an unbounded or lengthy amount of time then the approach outlined by ADAMANT will be inappropriate for DRE systems.
- **ADAMANT must provide a timely determination of optimal transport protocol and protocol settings.** If determination of an optimal protocol takes an unbounded or lengthy amount of time ADAMANT will be inappropriate for DRE systems.
- **ADAMANT must provide a timely transition to optimal transport protocol.** If transitioning to the optimal transport protocol takes an unbounded or lengthy amount of time then ADAMANT will be inappropriate solution approach for DRE systems.

These three concerns must all individually provide real-timeliness and bounded latencies since the three concerns involve the steps needed to appropriately perform autonomic adaptation. In proposing ADAMANT we plan to address all three timeliness concerns as well as the robustness and accidental complexity concerns warranted by operating conditions in dynamic environments.
Chapter 5

Concluding Remarks

In this proposal we initially presented the Distributed QoS Modeling Language (DQML), which is domain specific modeling language that provides design-time QoS configuration management. Specifically, DQML (1) allows developers to model desired entities and associated QoS policies for pub/sub DRE middleware, (2) reduces the accidental complexity of QoS variability, (3) checks the semantic compatibility of the modeled QoS configuration, and (4) and automatically generates implementation artifacts for a validated configuration model.

This proposal then described FLEXible Middleware And Transports (FLEXMAT), which is an evaluation framework for transport protocols as QoS mechanisms for QoS-enabled pub/sub middleware. FLEXMAT integrates and enhances (1) QoS-enabled pub/sub middleware and (2) a flexible transport protocol framework. FLEXMAT also provides composite QoS metrics to evaluate multiple QoS concerns as well as empirical evaluations and analysis.

Finally, we proposed an approach for ADAPtive Middleware And Network Transports (ADAMANT) to support QoS of pub/sub DRE systems in dynamic operating environments. ADAMANT combines and enhances (1) QoS-enabled pub/sub middleware, (2) a flexible network transport framework, (3) QoS monitoring infrastructure, (4) machine learning techniques, and (5) a controller to manage the monitoring and adaptation. Moreover, ADAMANT is intended (1) to provide timely and bounded adaptation as is needed for pub/sub DRE systems and (2) robust response as is needed for systems operating in dynamic environments.

Figure 5.1 shows the research progress and the plan for the completion of the proposed research. Table 5.1 presents the summary of research contributions, and is followed by a list research publications thus far.
Table 5.1: Summary of Research Contributions

<table>
<thead>
<tr>
<th>Category</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct QoS Design</td>
<td>DQML: design and implementation of a domain specific modeling tool that (1) manages QoS variability complexity, (2) checks the semantic QoS compatibility, and (4) and automatically generates implementation artifacts for a validated QoS configuration model.</td>
</tr>
<tr>
<td>Evaluation of QoS Mechanisms</td>
<td>FLEXMAT: QoS mechanism evaluation techniques integrating (1) pub/sub middleware, (2) a flexible transport framework, (3) composite QoS metrics, and (4) empirical evaluations and analysis.</td>
</tr>
<tr>
<td>Autonomic Adaptation for Qos</td>
<td>ADAMANT: autonomic adaption to support QoS in dynamic operating environments via (1) QoS-enabled pub/sub middleware, (2) a flexible network transport framework, (3) QoS monitoring infrastructure, (4) machine learning techniques, and (5) a controller to manage the monitoring and adaptation.</td>
</tr>
</tbody>
</table>
Summary of Publications and Presentations


Bibliography


[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995. 2.4.3


[34] E. G. et al., Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995. 3.4.1


