DESIGN AND RUN-TIME QUALITY OF SERVICE MANAGEMENT TECHNIQUES FOR PUBLISH/SUBSCRIBE DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS

By

Joseph William Hoffert II

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May, 2011

Nashville, Tennessee

Approved:

Dr. Aniruddha Gokhale

Dr. Douglas C. Schmidt

Dr. Larry Dowdy

Dr. Yi Cui

Dr. Janos Sztipanovits

To Sophie, William, and Annabelle, and to their terrific mother and my lifemate Kendra.
Above all, to God: Father, Son, and Holy Spirit; who was, and is, and is to come.

ACKNOWLEDGMENTS

This work would not have been possible without the financial support of the National Science Foundation's Team for Research in Ubiquitous Secure Technology (TRUST) program, the Air Force Research Labs, Raytheon, and the Science Applications International Corporation.

I would also like to take up this opportunity to thank some of the people who helped me in coming to this point in my life. First of all, I want to thank my parents, Marvin and Marilyn Hoffert, who have always emboldened me concerning academic pursuits. Growing up, it seemed there was no question too silly to ask. My parents always encouraged my questions with thoughtful adult answers. I hope to impart this wisdom to my children.

Ken Goldman at Washington University gave me my first opportunity as a research staff person at a highly regarded research university. Ken expressed confidence in me and my ability to figure out solutions to problems. Chris Gill also at Washington University encouraged me to consider Vanderbilt University for graduate school when I discussed getting a doctorate with him. His advice helped lead me to where I am today.

I have been privileged to have studied under such a world-class researcher as Dr. Douglas Schmidt. Doug has impressed me with his dedication to quality research and lucid communication. He has shown me the importance of not only doing and knowing the research but also communicating the research clearly to others. Supported by Doug, I have also been able to travel around the world and make connections with a wide array of people. I also appreciate the insight provided by Dr. Aniruddha Gokhale along with his willingness to advise me will Doug is on sabbatical. Andy's lighthearted levity interspersed between thoughtful comments make for an enjoyable work environment.

I want to thank Dr. Larry Dowdy and Dr. Yi Cui for serving as members of my dissertation committee and for their help and insight in the classroom as well as with my research. Dr. Janos Sztipanovits has been very encouraging and supportive with my research work

on TRUST and provided many opportunities for me to broaden my learning experiences. I appreciate the time all my committee members have taken to help improve my work.

I appreciate the help I received from former and current DOC group members. Kitty, Nishanth, Gan, and Jai helped show me "the DOC group ropes." Will Otte was a great help setting up the house and navigating my family and me to it during our U.K. internship. The DOC group in particular and ISIS in general has been a positive part of my doctoral education providing me exposure to new research areas through pizza lectures or advice from students in related areas (*e.g.*, Daniel Mack and his help with machine learning).

I could not have achieved this milestone without the love, respect, and support of my terrific wife of 19+ years, Kendra. She and I left a pretty good situation in St. Louis. We had friends, a nice house, and a fairly comfortable lifestyle provided by the salary I was making at Boeing. She and I had talked about my going back to school years before and when the opportunity came she was on board. Living in Nashville has been tough for both of us at times and I am glad some of it is coming to an end. More than that, I am thankful Kendra is still by my side. Aside from God, she is the greatest blessing I have in my life. My children, Sophie, William, and Annabelle, have also been a tremendous support. They help to make sure I have balance in my life and remind me that there are much more important things in life than schooling.

Last, but certainly not least, I thank the Triune God: Father, Son, and Holy Spirit. God has giving me the enjoyment of learning and the security of trusting Him to come to Vanderbilt and support a wife and three children on a graduate student's stipend. He has always been faithful. I desire above all else to be faithful to Him. Soli Deo Gloria!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

I.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 [49]. 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) [72], Web Services Brokered Notification [66], CORBA Event Service [82], and OMG Data Distribution Service (DDS) [84]. 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.

I.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.

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, unknown operating environments, such as those provided by cloud computing environments, or 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 is initially deployed may be unknown *a priori* or the initial environment configuration 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 determining appropriate QoS mechanisms for a given operating environment.

I.3 Enhancing Developer Productivity and Run-time Support for QoS-enabled Publish-Subscribe Middleware

To address the challenges identified in Section I.2, this dissertation enhances the productivity, evaluation, flexibility, and adaptability of QoS-enabled pub/sub middleware. The novel contributions of this dissertation focus on the following four synergistic areas: (1) a model-driven technique for developing semantically compatible QoS configurations and applicable implementation artifacts; (2) composite QoS metrics to quantitatively evaluate multiple QoS concerns simultaneously, (3) a technique for evaluating QoS mechanisms in specific operating environments and providing guidance for run-time QoS management of multiple QoS concerns within pub/sub systems in static environments; and (4) a technique to support autonomic configuration and adaptation of QoS-enabled pub/sub middleware in flexible and dynamic environments.

We briefly summarize the four 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 II.5 describes the QoS modeling process in detail.

2. **Composite QoS metrics** which include metrics to quantitatively evaluate the QoS concerns of (1) reliability, (2) average latency, (3) jitter (*i.e.*, standard deviation of received data), (4) network bandwidth usage, and (5) network burstiness (*i.e.*, the amount of data received within a specified period of time). Sections III.5.1.3 and IV.1.5.1 provide detailed descriptions of the family of ReLate2 metrics.

3. **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 III.5 describes the enhanced middleware, composite metrics, and quantitative analysis in detail.

4. **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 techniques 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 IV.1.3 describes the autonomically adaptive middleware configured for flexible environments in detail while Section IV.2.4 provides a detailed description of the adaptive middleware operating in dynamically changing environments.

I.4 Thesis 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 II discusses QoS configuration development for pub/sub DRE systems. Chapter III discusses a flexible QoS-enabled pub/sub middleware evaluation framework in varied environment configurations. Chapter IV discusses autonomic adaptation of pub/sub middleware to configure QoS mechanisms in flexible environments and support QoS in dynamic environments. Finally, chapter V provides a summary of the research contributions and publications.

CHAPTER II

DESIGN TIME MANAGEMENT OF QOS CONFIGURATION COMPLEXITY

Chapter I 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 the context of QoS configuration management, (2) presenting 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.

II.1 Context

Emerging trends for publish/subscribe systems. The use of distributed systems based on publish/subscribe (pub/sub) technologies has increased due to the advantages of scale, cost, and performance over single computers [49, 98]. In contrast to distributed object computing middleware (*e.g.*, Java RMI and CORBA) where clients invoke point-to-point methods on distributed objects, pub/sub middleware disseminates data from suppliers to one or more consumers. Examples of pub/sub middleware include Web Services Brokered Notification [66], the Java Message Service (JMS) [72], the CORBA Event Service [82], and the Data Distribution Service (DDS) [84]. These technologies support data propagation throughout a system using an anonymous subscription model that decouples event suppliers and consumers.

Pub/sub middleware is applicable to a broad range of application domains, such as

satellite coordination and shipboard computing environments. This middleware provides policies that affect end-to-end system QoS. Common policies include *persistence* (*i.e.*, saving data for current subscribers), *durability* (*i.e.*, saving data for subsequent subscribers), and *grouped data transfer* (*i.e.*, transmitting and receiving a group of data as an atomic unit).

Challenges in configuring pub/sub middleware. While tunable policies enable fine-grained control of system QoS, a number of challenges arise when developing *QoS policy configurations*, which are combinations of QoS properties that affect overall system QoS. For example, each QoS policy may have multiple parameters associated with it, such as the data topic of interest, data filter criteria, and the maximum number of messages to store when transmitting data. Each parameter can also be assigned 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 regular expressions used as filtering criteria.

The QoS policies associated with individual suppliers or consumers collectively determine the overall observed QoS of suppliers and consumers. Not all combinations of QoS policies/parameters deliver the required system QoS, however, and many combinations may not be semantically compatible. It is tedious and error-prone to transform a valid QoS policy configuration design manually to its implementation for a middleware platform.

Solution approach $\rightarrow$ Model-driven QoS policy configuration. We have developed a domain-specific modeling language (DSML) called the *Distributed QoS Modeling Language* (DQML) to address the challenges described above. In particular, DQML helps developers (1) choose valid sets of values for QoS policies in pub/sub middleware, (2) ensure that these QoS policy configurations are semantically compatible (*i.e.*, they do not conflict with each other), and (3) automate the transformation of a QoS policy configuration design into the correct pub/sub middleware-specific implementation.

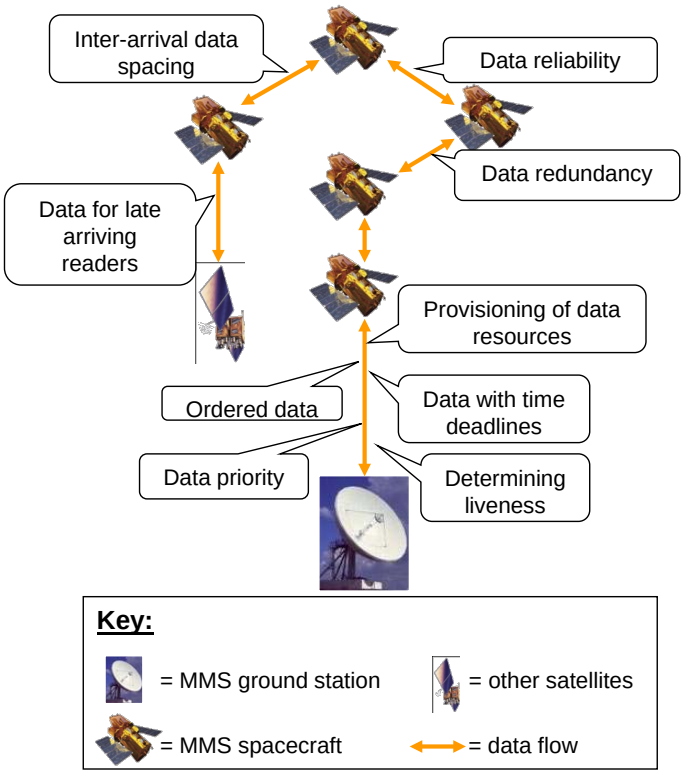

Figure 1: Example MMS Mission Scenario with QoS Requirements

II.2 Motivating Example: NASA's Magnetospheric Multiscale Mission

We chose NASA's *Magnetospheric Multiscale* (MMS) Mission [94] 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, 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 1. 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.

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

Table 1: MMS pub-sub QoS Policy Requirements

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 1.

A challenge for MMS developers is to determine how the interaction of the QoS policies listed in Table 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.

II.3 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) [107] 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 adaptation strategies as modeling elements to be incorporated into an existing DSML.

The *Options Configuration Modeling Language* (OCML) [62] 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 end points. For instance, these endpoints could be processes, objects, or components.

The MicroQoSCORBA [71] 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 [23] and the Publish/-subscribe Applied to Distributed REsource Scheduling (PADRES) [65] 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) [58] provides a profile for modeling QoS properties and mechanisms [24]. 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.

II.4 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 intertwined 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.

II.5 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 II.2) implemented using the OMG Data Distribution Service (DDS) [81] pub/sub middleware (see Section II.5.1.1). 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.

II.5.1 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.

II.5.1.1 Overview of the OMG Data Distribution Service (DDS)

The OMG DDS specification defines a standard pub/sub architecture and runtime capabilities that enables applications to exchange data in event-based distributed systems. DDS provides efficient, scalable, predictable, and resource-aware data distribution via its Data-Centric Publish/Subscribe (DCPS) layer, which supports a global data store where publishers write and subscribers read data, respectively. Its modular structure, power, and flexibility stem from its support for (1) location-independence, via anonymous pub/sub, (2) redundancy, by allowing any numbers of readers and writers, (3) real-time QoS, via its 22 QoS policies, (4) platform-independence, by supporting a platform-independent model for data definition that can be mapped to different platform-specific models (*e.g.,* C++ running on VxWorks or Java running on Real-time Linux), and (5) interoperability, by specifying a standardized protocol for exchanging data between distributed publishers and subscribers that allows implementations from different DDS vendors to interact.

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 research focuses on DCPS since it is better specified and supported than the DLRL.

As shown in Figure 2, several types of DCPS entities are specified for DDS. A *domain*

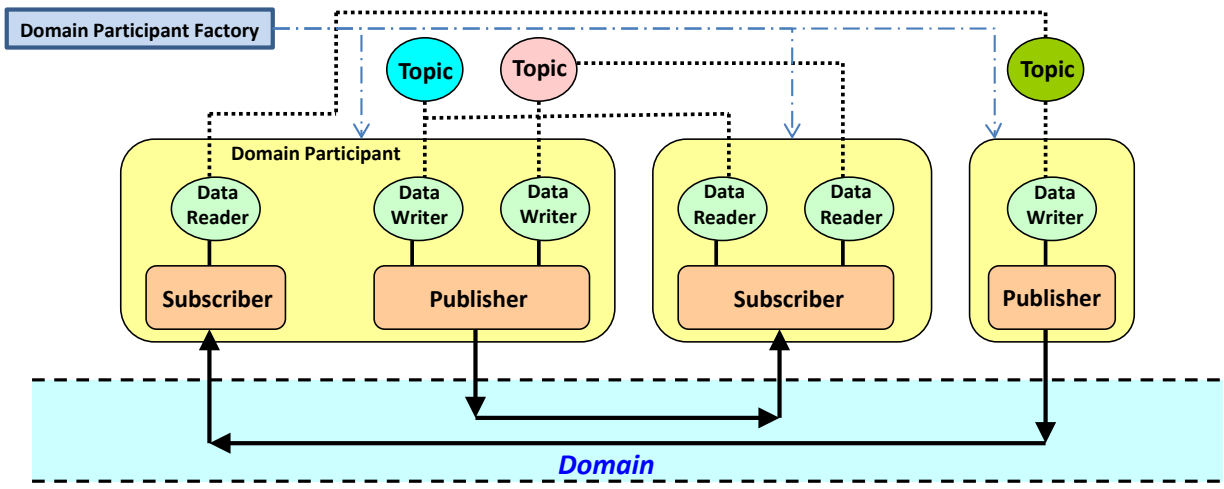

Figure 2: Architecture of the DDS Data-Centric Publish/Subscribe (DCPS) Layer

represents the set of applications that communicate with each other. A domain acts likes a virtual private network so that DDS entities in different domains are completely unaware of each other even if on the same machine or in the same process. A *domain participant factory*'s sole purpose is to create and destroy domain participants. The factory is a pre-existing singleton object that can be accessed by means of the `get_instance()` class operation on the factory. A *domain participant* provides (1) a container for all DDS entities for an application within a single domain, (2) a factory for creating publisher, subscriber, and topic entities, and (3) administration services in the domain, such as allowing the application to ignore locally any information about particular DDS entities.

DDS is topic-based, which allows strongly typed data dissemination since the type of the data is known throughout the entire system. In contrast, content-based pub/sub middleware, such as Siena [23] and the Publish/subscribe Applied to Distributed REsource Scheduling (PADRES) [65], examine events throughout the system to determine data types. A DDS *topic* describes the type and structure of the data to read or write, a *data reader* subscribes to the data of particular topics, and a *data writer* publishes data for particular topics. Various properties of these entities can be configured using combinations of the 22 QoS policies that are described in Table 2. In addition, *publishers* manage one or more

data writers while *subscribers* manage one or more data readers. Publishers and subscribers can aggregate data from multiple data writers and readers for efficient transmission of data across a network.

Topic types are defined via the OMG Interface Definition Language (IDL) that enables platform-independent type definition. An IDL topic type can be mapped to platform-specific native data types, such as C++ running on VxWorks or Java running on real-time Linux. Below we show an example topic definition in IDL that defines an analog sensor with a sensor id of type string and a value of type float.

```
struct AnalogSensor {
   string sensor\_id; // key
   float value; // other sensor data
};
```

DDS provides a rich set of QoS policies, as illustrated in Table 2. Each QoS policy has $\sim$2 attributes, with most attributes having an unbounded number of potential values (*e.g.*, an attribute of type character string or integer). The DDS specification defines which QoS policies are applicable for certain entities, as well as which combinations of QoS policy values are semantically compatible as outlined in Section II.5.1.3. For example, if a data reader and data writer associated via a common topic want data to flow reliably, they must both specify reliable transmission via the reliability QoS policy.

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.

The extensive QoS support of DDS and the flexibility of the QoS policies present the challenges of appropriately managing the policies to form the desired QoS configuration. These challenges not only include ensuring valid QoS parameter types and values but also ensuring valid interactions between the policies and the DDS entities. Moreover, managing

DDS QoS Policy	Description
Deadline	Determines rate at which periodic data is refreshed
Destination Order	Sets whether data sender or receiver determines order
Durability	Determines if data outlives the time when written or read
Durability Service	Details how durable data is stored
Entity Factory	Sets enabling of DDS entities when created
Group Data	Attaches application data to publishers, subscribers
History	Sets how much data is kept to be read
Latency Budget	Sets guidelines for acceptable end-to-end delays
Lifespan	Sets time bound for "stale" data
Liveliness	Sets liveness properties of topics, data readers, data writers
Ownership	Controls writer(s) of data
Ownership Strength	Sets ownership of data
Partition	Controls logical partition of data dissemination
Presentation	Delivers data as group and/or in order
Reader Data Lifecycle	Controls data and data reader lifecycles
Reliability	Controls reliability of data transmission
Resource Limits	Controls resources used to meet requirements
Time Based Filter	Mediates exchanges between slow consumers and fast producers
Topic Data	Attaches application data to topics
Transport Priority	Sets priority of data transport
User Data	Attaches application data to DDS entities
Writer Data Lifecycle	Controls data and data writer lifecycles

Table 2: DDS QoS Policies

semantic compatibility increases the accidental complexity of creating valid QoS configuration since not all valid combinations of QoS policies will produce the desired system behavior as outlined above with the flow of reliable data. It is incumbent upon the developer to use the QoS policies appropriately and judiciously.

DSMLs can help address these challenges. DSMLs can reduce the variability complexity of managing multiple QoS policies and their parameters by presenting the QoS policies as modeling elements that are automatically checked for appropriate associations and whose parameters are automatically typed and checked for appropriate values. DSMLs can also codify constraints for semantic compatibility to ensure that data flows as intended. Moreover, DSMLs can automatically generate implementation artifacts that accurately reflect the design. This section uses our MMS example from Section II.2 to present the challenges of configuring DDS QoS policies so the system executes as intended.

II.5.1.2 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) *runtime* (*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 inadvertently 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 the not 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 [51]. 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. Various developers may also 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 catalog 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 I. 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 research focuses on DSML-based solutions.

II.5.1.3 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 [51] 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 I. 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 II.5.2.2 describes how DQML addresses the QoS policy configuration variability challenge by allowing only valid values to be assigned to parameters and checking for valid associations between QoS

QoS Policies	Affected DDS Entities
Deadline	Topic, data reader, data writer
Destination Order	Topic, data reader, data writer
Durability	Topic, data reader, data writer
Latency Budget	Topic, data reader, data writer
Liveliness	Topic, data reader, data writer
Ownership	Topic, data reader, data writer
Presentation	Publisher, subscriber
Reliability	Topic, data reader, data writer

Table 3: Potential Incompatible DDS QoS Policies

policies and entities.

MMS Challenge 2: Ensuring QoS compatibility.

Context. DDS defines constraints for compatible QoS policies. Table 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*).

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 3.

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 II.5.2.2 describes how DQML addresses the QoS compatibility challenge by providing

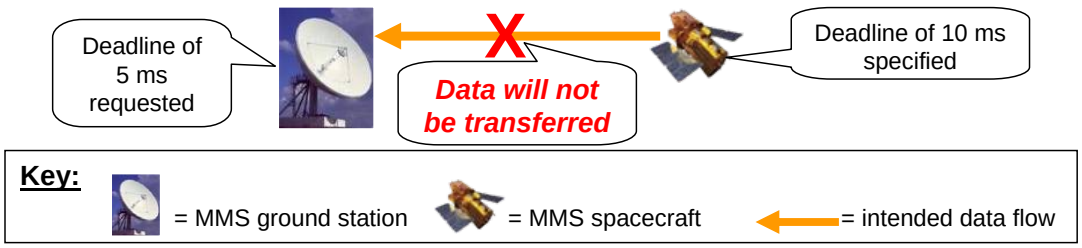

Figure 3: Incompatible MMS Ground Station and Spacecraft Deadline QoS

Consistency Constraints for QoS Policies
Deadline.period $\geq$ Time_Based_Filter.minimum separation
Resource_Limits.max_samples $\geq$ Resource_Limits.max_samples_per_instance
Resource_Limits.max_samples_per_instance $\geq$ History.depth

Table 4: DDS QoS Consistency Constraints

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 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 4. This invalid configuration violates the DDS constraint of deadline period $\geq$ minimum separation.

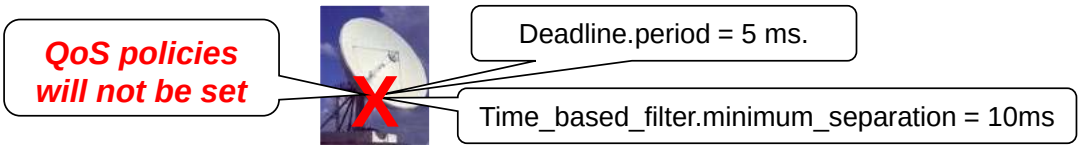

Figure 4: Inconsistent QoS Policies for an MMS Ground Station

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 II.5.2.2 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[1] 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 II.5.2.2 describes how DQML addresses the challenge of correct QoS transformation by providing an interpreter that traverses the model and generates implementation specific artifacts.

[1] In our research "correct-by-construction" refers to QoS policy configuration artifacts that faithfully transfer design configurations into implementation and deployment.

II.5.2 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) [64], which is a meta-programmable environment for creating DSMLs. This section describes the structure and functionality of DQML and explains how it resolves the challenges from Section II.5.1.3 in the context of DDS and the MMS case study.

II.5.2.1 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, 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 5 and 6 illustrate a portion of the DQML metamodel pertaining to the *Deadline QoS Policy*. Figure 5 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 6 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

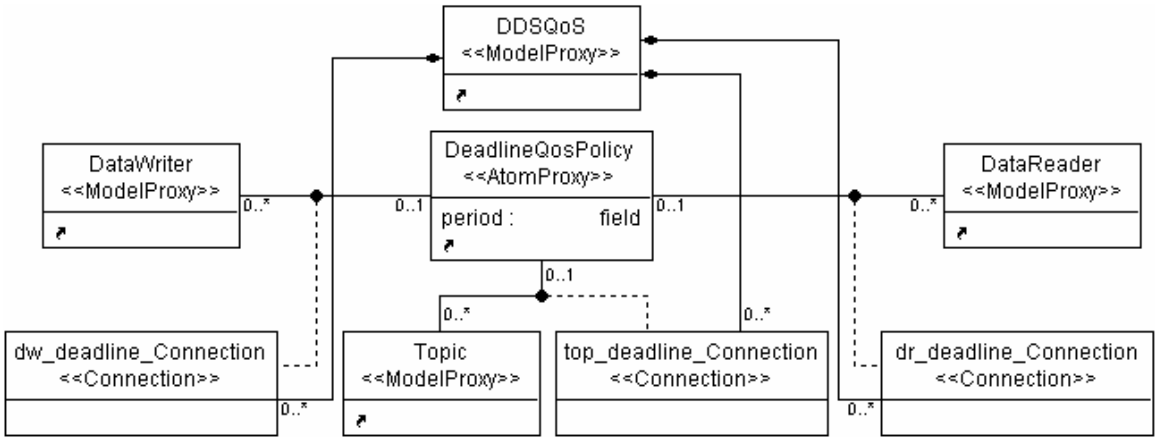

Figure 5: Deadline QoS Policy Relationships (UML notation)

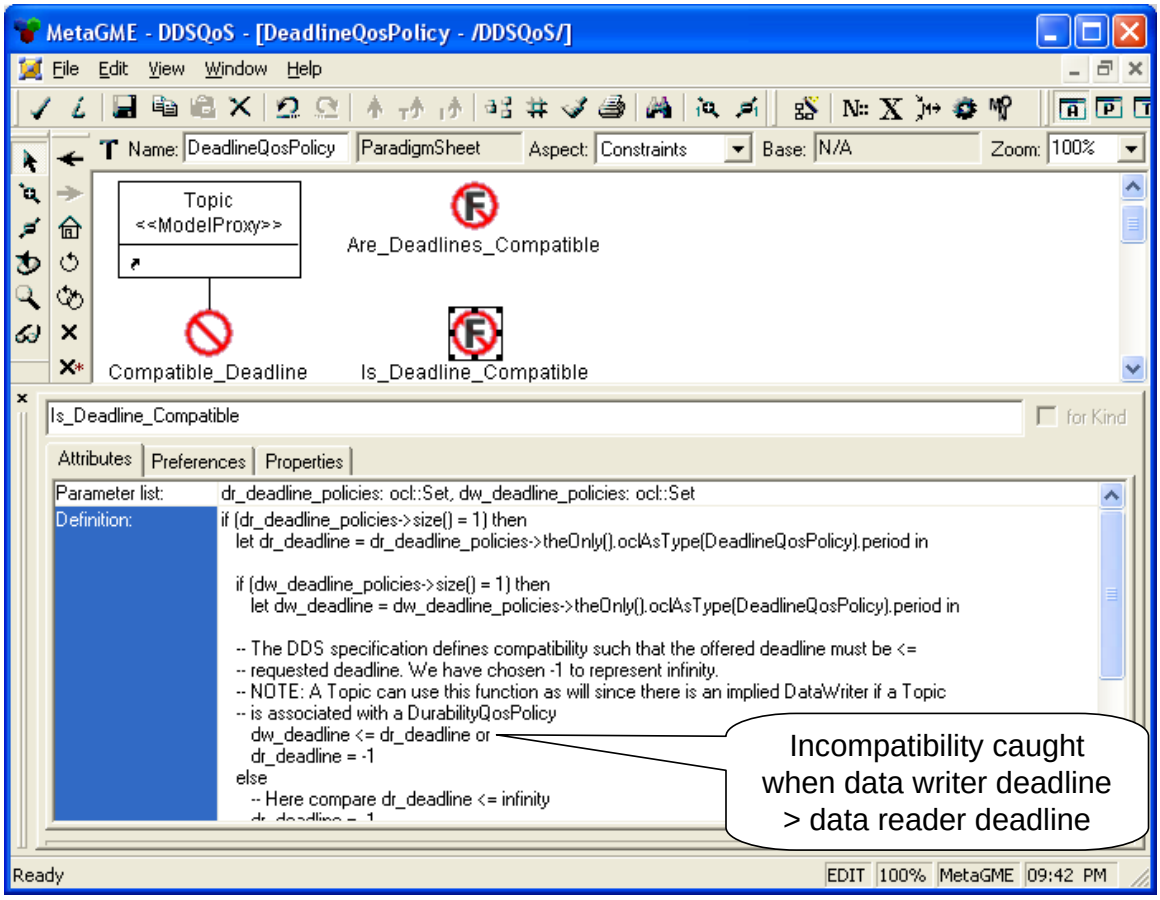

Figure 6: Deadline QoS Policy Compatibility Constraints

manner similar to Figures 5 and 6 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) [105] implementation to define these constraints. As noted in Section II.5.1.3 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 6 highlights the OCL constraint that catches the deadline incompatibility of Figure 3. 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

Figure 7: DDS Entities Supported in DQML

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

II.5.2.2 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 7. QoS policies can be created and associated with these entities as described below.

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

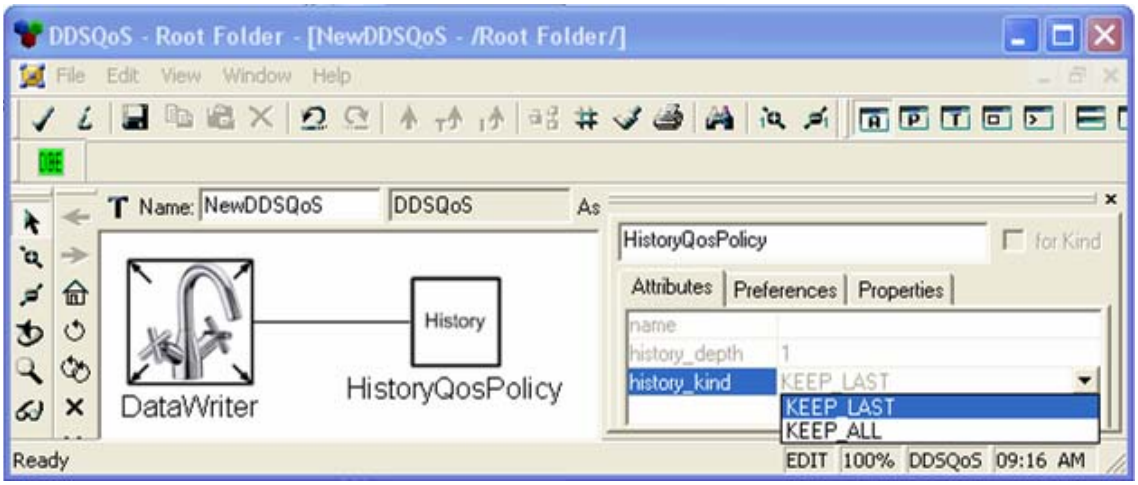

Figure 8: Example of DQML QoS Policy Variability Management

DQML interpreter externalizes the parameter values (whether set explicitly or by default) so that no QoS policy has uninitialized parameters.

Figure 8 shows an example of how DQML addresses the challenge of managing QoS policy configuration variability as outlined in Section II.5.1.3. 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.

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 9 and 10 show examples of how DQML addresses the challenges of ensuring QoS compatibility and consistency, respectively, as described in Section II.5.1.3. Figure 9 shows how DQML detects and notifies users of incompatible reliability QoS policies. Likewise, Figure 10 shows an incompatible 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.

Finally, the design decision was made to models DDS entities as GME models and QoS policies as GME atoms. Within GME the use of models allows containment of other GME objects while atoms do not allow this. Currently, DDS entities do not need to contain anything else but initially this was not clear. In hindsight, the DDS entities may have been

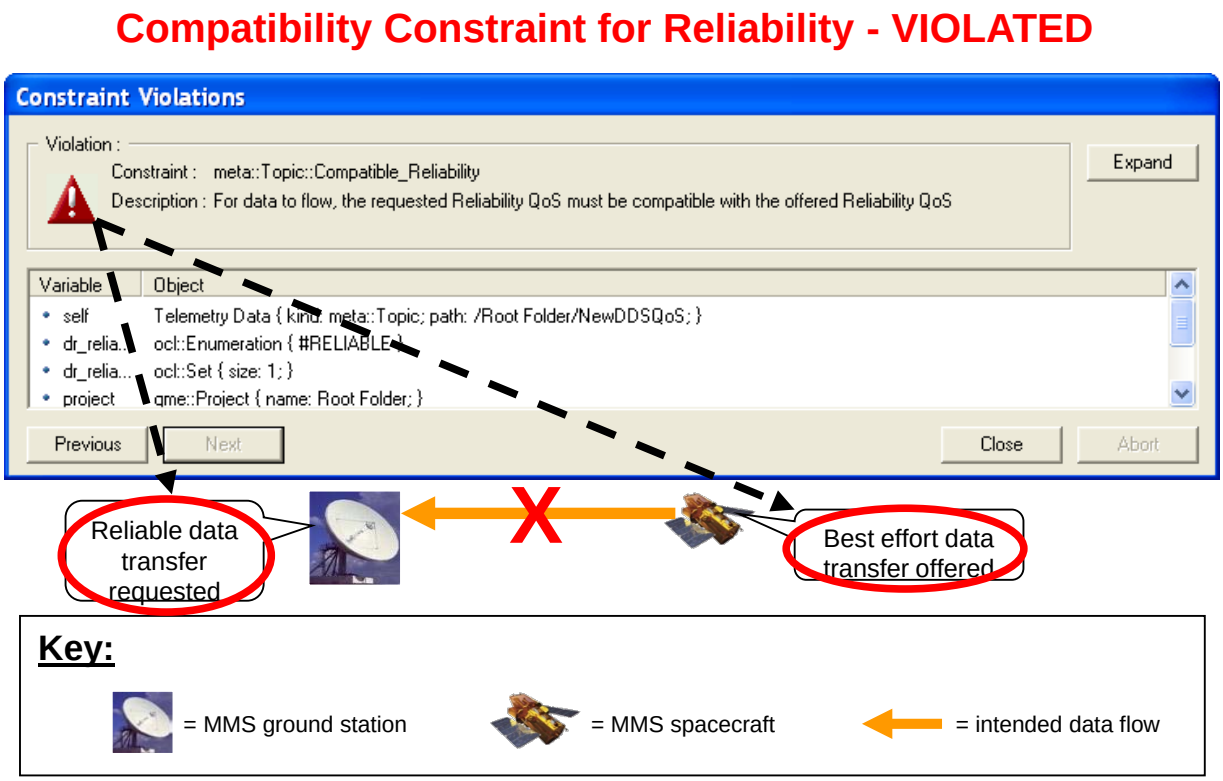

Figure 9: Example of DQML QoS Policy Compatibility Constraint Checking

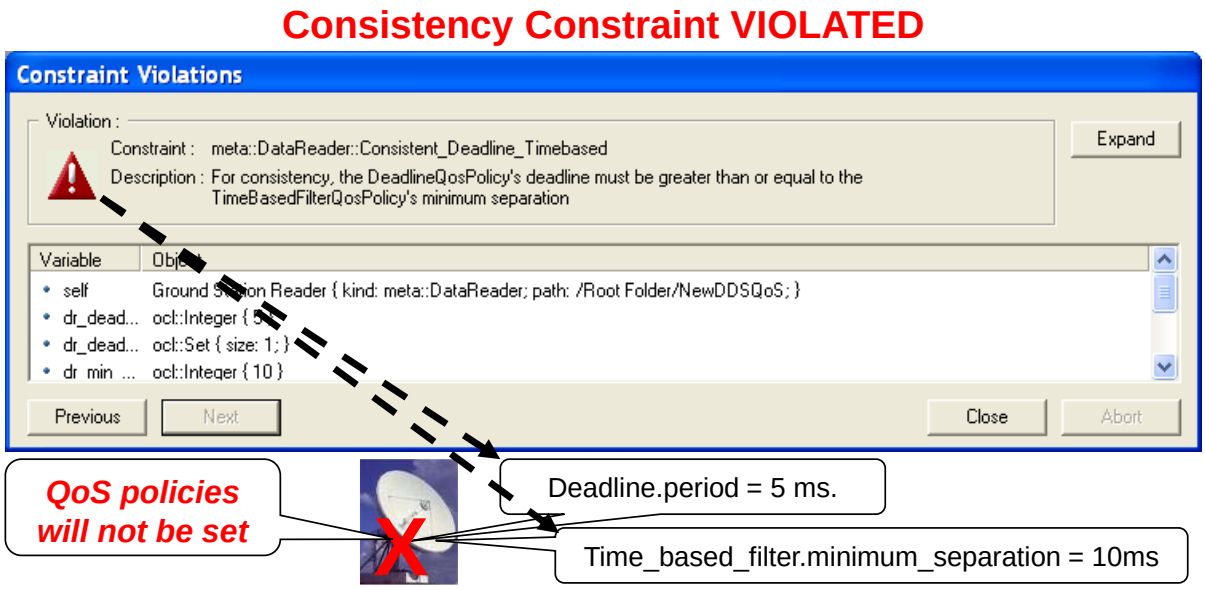

Figure 10: Example of DQML QoS Policy Consistency Constraint Checking

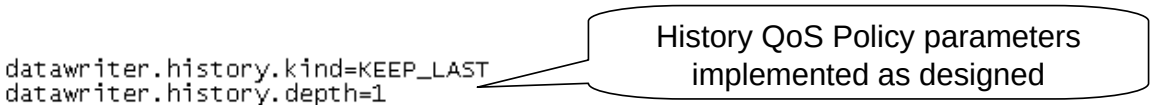

Figure 11: QoS Policy Configuration File for Figure 8

modeled as atoms although there may still be a need for containment at some point in the future. It seems prudent to leave this option open for the time being.

Transforming QoS policy configurations from design to implementation. Figure 11 shows how DQML addresses the challenge of correctly transforming QoS policy configurations from design to implementation, as described in Section II.5.1.3. In this example, DQML generates the QoS policy configuration file for an MMS satellite data writer as modeled in Figure 8. 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.

II.5.3 DQML Productivity Analysis for the MMS Case Study

Model-driven engineering (MDE) helps address the problems of designing, implementing, and integrating applications [6, 40, 59, 91]. MDE is increasingly used in domains involving modeling software components, developing embedded software systems, and configuring quality-of-service (QoS) policies. Key benefits of MDE include (1) raising the level of abstraction to alleviate accidental complexities of low-level and heterogeneous software platforms, (2) more effectively expressing designer intent for concepts in a domain, and (3) enforcing domain-specific development constraints. Many documented benefits of MDE are qualitative (*e.g.*, use of domain-specific entities and associations that are familiar to domain experts, visual programming interfaces where developers can manipulate icons representing domain-specific entities to simplify development). There is a lack of documented quantitative benefits for DSMLs, however, that show how developers are more productive using MDE tools and how development using DSMLs yields fewer bugs.

Conventional techniques for quantifying the benefits of MDE in general (*e.g.*, comparing user-perceived usefulness of measurements for development complexity [3, 4]) and DSMLs in particular (*e.g.*, comparing elapsed development time for a domain expert with and without the use of the DSML [69]) involve labor-intensive and time-consuming experiments. For example, control and experimental groups of developers may be tasked to complete a development activity during which metrics are collected (*e.g.*, number of defects, time required to complete various tasks). These metrics also often require the analysis of domain experts, who may be unavailable in many production systems.

Even though DSML developers are typically responsible for showing productivity gains, they often lack the resources to demonstrate the quantitative benefits of their tools. One way to address this issue is via productivity analysis, which is a lightweight approach to quantitatively evaluating DSMLs that measures how productive developers are, and quantitatively exploring factors that influence productivity [14, 87]. We apply quantitative productivity measurement using a case study of DQML.

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 1. DBE is a suite of software tools that can examine and evaluate various DDS implementations [106]. 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 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 previously

in Section II.5.1.2 [46] since neither approach provides implementation guidance. Our productivity analysis shows significant productivity gains compared with common alternatives, such as manual development using third-generation programming languages. While this section focuses on DQML, in general the productivity gains and analysis presented are representative of DSMLs' ability to reduce accidental complexity and increase reusability.

II.5.3.1 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) modeling the desired DDS entities and QoS policies in DQML, (2) invoking the DBE interpreter to generate the appropriate QoS settings files, and (3) executing DBE to deploy data readers and data writers using the generated QoS settings files.

II.5.3.2 Productivity Analysis

Productivity Analysis Approach.

When analyzing productivity gains for a given DSML, analysts can employ several different types of strategies, such as

• **Design development effort**, comparing the effort (*e.g.*, time, number of design steps [11], number of modeling elements [56, 104]) it takes a developer to generate a design using traditional methods (*e.g.*, manually) versus generating a design using the DSML,

• **Implementation development effort**, comparing the effort (*e.g.*, time, lines of code) it takes a developer to generate implementation artifacts using traditional methods (*i.e.*, manual generation versus generating implementation artifacts using the DSML),

• **Design quality**, comparing the number of defects in a model or an application developed traditionally to the number of defects in a model or application developed using the DSML,

• **Required developer experience**, comparing the amount of experience a developer needs to develop a model or application using traditional methods to the amount of experience needed when using a DSML, and

• **Solution exploration**, comparing the number of viable solutions considered for a particular problem in a set period of time using the DSML as compared to traditional methods or other DSMLs.

Our focus is on the general area of quantitative productivity measurement-specifically on implementation development effort in terms of lines of code. The remainder of this section compares the lines of configuration code manually generated for DBE data readers and data writers to the lines of C++ code needed to implement the DQML DBE interpreter, which in turn generates the lines of configuration code automatically.

Metrics for DQML Productivity Analysis.

Below we analyze the effect on productivity and the breakeven point of using DQML as opposed to manual implementations of QoS policy configurations for DBE. Although configurations can be designed using various methods as outlined previously in Section II.5.1.2, manual implementation of configurations is applicable to these other design solutions since these solutions provide no guidance for implementation.

Within the context of DQML, we developed an interpreter specific to DBE to support

DBE's requirement of correct QoS policy configurations. The interpreter generates QoS policy parameter settings files for the data readers and data writers that DBE configures and deploys. All relevant QoS policy parameter settings from a DQML model are output for the data readers and data writers including settings from default as well as explicitly assigned parameters.

As appropriate for DBE, the interpreter generates a single QoS policy parameter settings file for every data reader or data writer modeled. Care is taken 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. Moreover, the interpreter's generation of filenames aids in QoS settings files management (as described in Section II.5) since the files are uniquely and descriptively named. The following subsections detail the scope, development effort, and productivity analysis of DQML's DBE interpreter versus manual methods.

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.

A data writer can be associated with 15 QoS policies with a total of 25 parameters, as shown in Table 5. A data reader can be associated with 12 QoS policies with a total of 18 parameters, as shown in Table 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 11.

Interpreter development. We developed the DBE interpreter for DQML using GME's

QoS Policy	# of Params	Param Type(s)
Deadline	1	int
Destination Order	1	enum
Durability	1	enum
Durability Service	6	5 ints, 1 enum
History	2	1 enum, 1 int
Latency budget	1	int
Lifespan	1	int
Liveliness	2	1 enum, 1 int
Ownership	1	enum
Ownership Strength	1	int
Reliability	2	1 enum, 1 int
Resource Limits	3	3 ints
Transport Priority	1	int
User Data	1	string
Writer Data Lifecycle	1	bool
Total Parameters	**25**	

Table 5: DDS QoS Policies for Data Writers

QoS Policy	# of Params	Param Type(s)
Deadline	1	int
Destination Order	1	enum
Durability	1	enum
History	2	1 enum, 1 int
Latency budget	1	int
Liveliness	2	1 enum, 1 int
Ownership	1	enum
Reader Data Lifecycle	2	2 ints
Reliability	2	1 enum, 1 int
Resource Limits	3	3 ints
Time Based Filter	1	int
User Data	1	string
Total Parameters	**18**	

Table 6: DDS QoS Policies for Data Readers

Builder Object Network (BON2) framework, which provides C++ code to traverse the DQML model utilizing the Visitor pattern [32]. When using BON2, developers of a DSML interpreter only need to modify and add a small subset of the framework code to traverse and appropriately process the particular DSML model. More specifically, the BON2 framework supplies a C++ visitor class with virtual methods (*e.g.*, `visitModelImpl`, `visitConnectionImpl`, `visitAtomImpl`). The interpreter developer then subclasses and overrides the applicable virtual methods.

The DDS entities relevant to DQML are referred to as model implementations in BON2. Therefore, the DBE interpreter only needs to override the `visitModelImpl()` method and is not concerned with other available virtual methods. When the BON2 framework invokes `visitModelImpl()` it passes a model implementation as an argument. A model implementation includes methods to (1) traverse the associations a DDS entity has (using the `getConnEnds()` method) and specify the relevant QoS policy association as an input parameter (*e.g.*, the association between a data writer and a deadline QoS Policy), (2) retrieve the associated 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 utilizes 160 C++ statements within the BON2 framework. We stress that any interpreter development is a one-time cost; specifically there is no development cost for the DBE interpreter since it is already developed. The main challenge in using BON2 is understanding how to traverse the model and access the desired information. After interpreter developers are familiar with BON2, the interpreter development is fairly straightforward. We detail the steps of developing the DBE interpreter below.

Figure 12 outlines the visitor class that has been created for the DBE interpreter for use within the BON2 framework. This class is the only class that needs to be implemented for the DBE interpreter. Line 1 determines the class name and its derivation from the BON2 Visitor class. Lines 3 and 4 declare the default constructor and destructor respectively.

```
1  class DDSQoSVisitor : public Visitor {
2  public:
3      DDSQoSVisitor ();
4      ~DDSQoSVisitor ();
5
6  protected :
7      virtual void visitAtomImpl (const Atom& atom);
8      virtual void visitModelImpl (const Model& model);
9      virtual void visitConnectionImpl ( const Connection& connection);
10
11     void processDataReaderQos (const Model& dataReader);
12     void processDataWriterQos (const Model& dataWriter);
13
14     void outputDDSEntityQos (const Model& dds_entity,
15                              const std::string &entity_name,
16                              const std::string &entity_abbrev,
17                              const std::string &qos_connection_name,
18                              const std::string &qos_name,
19                              const std::map<std::string, std::string> &attribute_map,
20                              int entity_count,
21                              bool &file_opened,
22                              std::ofstream &out_file);
23
24 private:
25     std::ofstream out_file_;
26 };
```

Figure 12: Visitor Class for DBE Interpreter

Lines 7 - 9 declare the abstract methods `visitAtomImpl`, `visitModelImpl`, and `visitConnectionImpl` inherited from the BON2 Visitor class that need to be defined for the DBE interpreter. Lines 11 and 12 declare methods to process data readers and data writers respectively. Lines 14 - 22 declare the main method that processes the QoS properties for a data reader or data writer and writes the QoS parameters to the appropriate file. Line 25 defines the debugging output file that had been used for debugging the DBE interpreter.

As is shown in Figure 12, the structure of the DBE visitor class is fairly simple and straightforward. Moreover, of the three methods inherited from the BON2 Visitor class and declared on lines 7 - 9 only the `visitModelImpl` method declared on line 8 is a non-empty method. For DBE, the only DQML entities of interest are what GME terms the model elements which for DBE's interests are the data readers and data writers. The

```
1  void DDSQoSVisitor::visitModelImpl( const Model& model )
2  {
3      if (model->getModelMeta().name() == "DataReader")
4      {
5          out_file_ << "DDS DataReader Name: " << model->getName() << std::endl;
6          processDataReaderQos(model);
7          out_file_ << "...Done DDS DataReader Name: " << model->getName() << std::endl;
8      }
9      else if (model->getModelMeta().name() == "DataWriter")
10     {
11         out_file_ << "DDS DataWriter Name: " << model->getName() << std::endl;
12         processDataWriterQos(model);
13         out_file_ << "...Done DDS DataWriter Name: " << model->getName() << std::endl;
14     }
15 }
```

Figure 13: visitModelImpl Method

DBE interpreter is not concerned with traversing atom or connection elements since these elements will be addressed by processing the model elements.

We now focus on the implementations of the relevant methods particularly as they relate to complexity and required background knowledge. The default constructor and destructor simply open and close the file used for debugging which is not required functionality for the DBE interpreter. Therefore the implementations of these two methods (which total two C++ statements) are excluded to save space. The `visitAtomImpl` and `visitConnectionImpl` methods are defined (since the inherited methods are abstract) but empty (since they are not needed).

As shown in Figure 13, the `visitModelImpl` method determines the type of model element currently being processed and calls the appropriate method (*i.e.*, `processDataReaderQos` for a data reader on line 6 and `processDataWriterQos` for a data writer on line 12). The lines written to `out_file_` are simply for debugging purposes and are not required by DBE. The DBE interpreter developer required familiarity with the DQML metamodel to know the names of the model elements of interest but the model elements in the metamodel were given intuitive names to reduce accidental complexity (*e.g.*, `DataReader` and `DataWriter` on lines 3 and 9 respectively).

```
1  void DDSQoSVisitor::processDataWriterQos( const Model& dataWriter )
2  {
3      static int dw_count = 1;
4      const std::string dw_name("DataWriter");
5      const std::string dw_prefix("DW");
6      std::ofstream output_file;
7      bool file_opened = false;
8      std::map<std::string, std::string> attrib_map;
9
10     // Handle Deadline QoS Policy
11     attrib_map.clear ();
12     attrib_map["period"] = "datawriter.deadline.period=";
13     outputDDSEntityQos (dataWriter,
14                         dw_name,
15                         dw_prefix,
16                         "dw_deadline_Connection",
17                         "Deadline",
18                         attrib_map,
19                         dw_count,
20                         file_opened,
21                         output_file);
22
23     // Handle History QoS Policy
24     attrib_map.clear ();
25     attrib_map["history_kind"] = "datawriter.history.kind=";
26     attrib_map["history_depth"] = "datawriter.history.depth=";
27     outputDDSEntityQos (dataWriter,
28                         dw_name,
29                         dw_prefix,
30                         "dw_history_Connection",
31                         "History",
32                         attrib_map,
33                         dw_count,
34                         file_opened,
35                         output_file);
       •
       •
       •
```

Figure 14: processDataWriterQos Method

Figure 14 outlines the `processDataWriterQos` method. For each QoS policy applicable to a data writer this method sets up a mapping of DQML QoS parameter names to DBE QoS parameters names. Then the method calls the `outputDDSEntityQoS` method to write the QoS parameter values to the appropriate file. The interpreter developer needed to have an understanding of the QoS parameter names for DBE, the QoS parameter names in the DQML metamodel, and the names of the associations between data readers/writers and QoS policies in the DQML metamodel. However, as with the model elements in the DQML metamodel, the QoS parameters were given intuitive names to reduce accidental complexity (*e.g.*, `history_kind` and `history_depth` on lines 25 and 26 respectively) as were the connection names (*e.g.*, `dw_deadline_Connection` and `dw_history_Connection` on lines 16 and 30 respectively).

Figure 14 shows the source code for processing the deadline and history QoS policies. The rest of the method, which has been elided for brevity, handles all the other QoS policies relevant to data writers. Finally, the method closes the QoS parameter file if one has been opened previously and increments the count of data writers processed so that unique filenames can be generated. Likewise, the `processDataReaderQos` method provides the same functionality for QoS policies and parameters relevant to data readers. Its source code is not included due to space constraints.

Figure 15 presents the `outputDDSEntityQos` method which traverses the connection that a data reader or data writer has to a particular QoS policy (*e.g.*, connections to QoS policies for data readers or data writers) and writes the QoS parameters out to the QoS settings file for that data reader or writer. Lines 14 - 21 and 54 - 57 provide error checking for the BON2 framework and have been elided for space considerations. Line 11 retrieves the associations that the data reader or writer has with a particular QoS policy (*e.g.*, all the associations between a data reader and the reliability QoS policy). Lines 24 - 27 retrieve the endpoint of the connection which will be the associated QoS policy of the type specified as the input parameter of line 4. Lines 29 and 30 retrieve the parameters of

the associated QoS policy, lines 31 - 41 open a uniquely named DBE QoS settings file if one is not currently open, and lines 42 - 52 iterate through the QoS parameters and write them out to the opened file in the required DBE format using the attribute mapping passed as an input parameter on line 6.

Since the BON2 framework relies on the *Visitor* pattern, familiarity with this pattern can be helpful. This familiarity is not required, however, and developers minimally only need to implement relevant methods for the automatically generated Visitor subclass. In general, the DQML interpreter code specific to DBE (1) traverses the model to gather applicable information, (2) creates the QoS settings files, and (3) outputs the settings into the QoS settings files.

The C++ development effort for DQML's DBE interpreter is only needed one time. In particular, no QoS policy configuration developed via DQML for DBE incurs this development overhead since the interpreter has already been developed. The development effort metrics of 160 C++ statements are included only to be used in comparing manually implemented QoS policy configurations.

Analysis for the MMS scenario. The hardest aspect of developing DQML's DBE interpreter is traversing the model's data reader and data writer elements along with the associated QoS policy elements using the BON2 framework. Conversely, the most challenging aspects of manually implementing QoS policy configurations are (1) maintaining a global view of the model to ensure compatibility and consistency, (2) verifying the number, type, and valid values for the parameters of the applicable QoS policies, and (3) faithfully transforming the configuration design into implementation artifacts. On average, implementing a single C++ statement for the DBE interpreter is no harder than implementing a single parameter statement for the DBE QoS settings files. When implementing a non-trivial QoS policy configuration, therefore, development of the C++ code for the DBE interpreter is no more challenging than manually ensuring that the QoS settings in settings files are valid,

```
1  void DDSQoSVisitor::outputDDSEntityQos (const Model& dds_entity,
2                                          const std::string &entity_name, // e.g., "DataReader"
3                                          const std::string &entity_abbrev, // e.g., "DR"
4                                          const std::string &qos_connection_name,
5                                          const std::string &qos_name,
6                                          const std::map<std::string, std::string> &attribute_map,
7                                          int entity_count,
8                                          bool &file_opened,
9                                          std::ofstream &out_file)
10 {
11     std::multiset<ConnectionEnd> conns = dds_entity->getConnEnds(qos_connection_name);
12     if (conns.size() > 0)
13     {
14         if (conns.size() > 1) { ... }
22         else
23         {
24             std::multiset<ConnectionEnd>::const_iterator iter(conns.begin ());
25             ConnectionEnd endPt = *iter;
26             FCO fco(endPt);
27             if (fco)
28             {
29                 std::set<Attribute> attrs = fco->getAttributes ();
30                 std::set<Attribute>::const_iterator attr_iter(attrs.begin());
31                 if (!file_opened)
32                 {
33                     file_opened = true;
34                     std::string filename;
35
36                     char cnt_buf [10];
37                     ::sprintf_s (cnt_buf, "%d", entity_count);
38                     std::string cnt_str = cnt_buf;
39                     filename = entity_abbrev + cnt_str + "_" + dds_entity->getName () + ".txt";
40                     out_file.open(filename.c_str ());
41                 }
42                 for (; attr_iter != attrs.end (); ++attr_iter)
43                 {
44                     Attribute attr = *attr_iter;
45                     std::string attr_name = attr->getAttributeMeta ().name ();
46                     std::map<std::string, std::string>::const_iterator map_iter =
47                         attribute_map.find (attr_name);
48                     if (map_iter != attribute_map.end ())
49                     {
50                         out_file << map_iter->second << attr->getStringValue () << std::endl;
51                     }
52                 }
53             }
54             else { ... }
58         }
59     }
60 }
```

Figure 15: outputDDSEntityQos Method

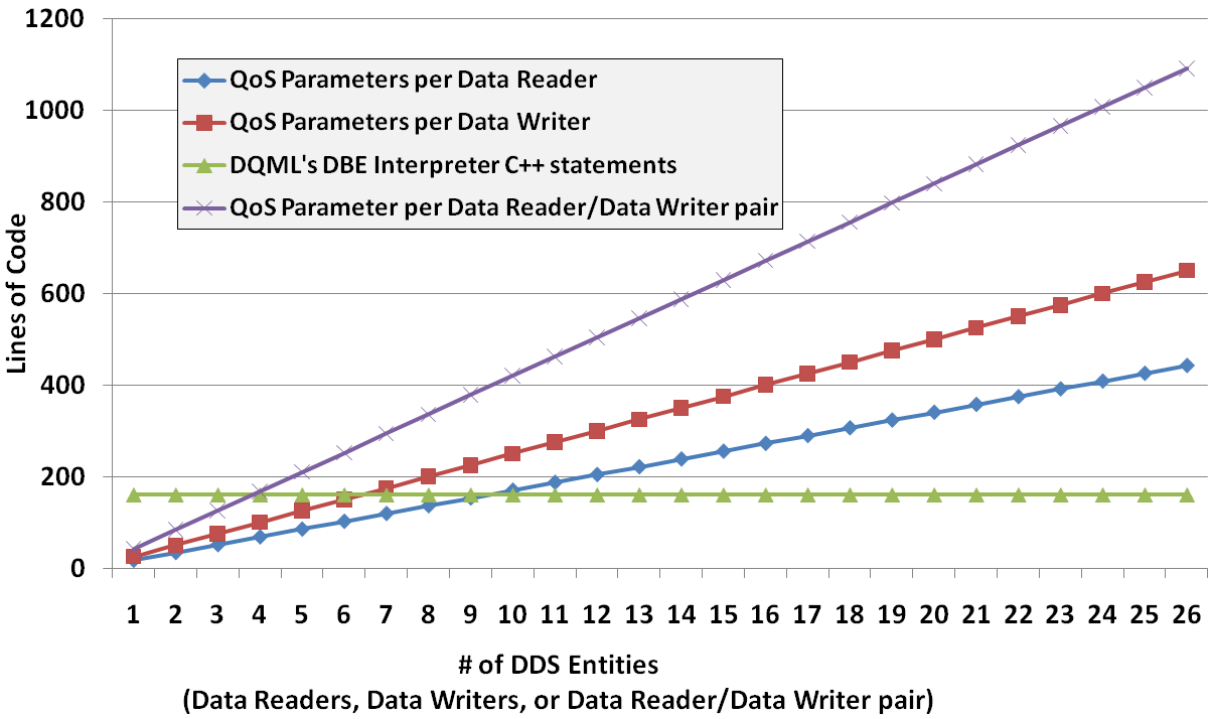

Figure 16: Metrics for Manual Configuration vs. DQML's Interpreter

consistent, compatible, and correctly represent the designed configuration. Below we provide additional detail into what can be considered a non-trivial QoS policy configuration.

The development and use of the DBE interpreter for DQML is justified for a single QoS policy configuration when at least 160 QoS policy parameter settings are involved. These parameter settings correlate to the 160 C++ statements for DQML's DBE interpreter. Using the results for QoS parameters in Table 5 and Table 6 for data readers and data writers, Figure 16 shows the justification for interpreter development. The development is justified with $\sim$10 data readers, $\sim$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 = 165 QoS policy parameter settings). For comparison, the break-even point for data reader/writer pairs is 3.72 (*i.e.*, 160/43).

We also quantified the development effort needed to support topics if the DBE interpreter required that functionality. Table 7 shows the DDS QoS policies and policy parameters applicable to topics. To support topics an additional 59 C++ statements would need to

QoS Policy	# of Params	Param Type(s)
Deadline	1	int
Destination Order	1	enum
Durability	1	enum
Durability Service	6	5 ints, 1enum
History	2	1 enum, 1 int
Latency budget	1	int
Lifespan	1	1 int
Liveliness	2	1 enum, 1 int
Ownership	1	enum
Reliability	2	1 enum, 1 int
Resource Limits	3	3 ints
Transport Priority	1	1 int
Topic Data	1	string
Total Parameters	**23**	

Table 7: DDS QoS Policies for Topics

be added. Conversely, for manual generation 23 more QoS parameters need to be considered for each topic. The break-even point for data reader/writer/topic triplets becomes 3.32 (*i.e.*, (160 + 59)/(43 + 23)) which is less than the break-even point for data reader/writers alone (*i.e.*, 3.72).

This break-even point is less because the additional source code to support topics can leverage existing code, in particular, the `outputDDSEntityQos` method outlined in Figure 15. The break-even point can be applicable for any interpreter that leverages the commonality of formatting regardless of the entity type (*cf.* `outputDDSEntityQos` method). Moreover, the complexity of developing any DQML interpreter is lessened by having the DBE interpreter as a guide. The design and code of the DBE interpreter can be reused by another application-specific interpreter to navigate a DQML model and access the QoS policies.

Table 8 also shows productivity gains as a percentage for various numbers of data readers and data writers. The percentage gains are calculated by dividing the number of parameter values for the data readers and data writers involved by the number of interpreter C++

# of Data Readers and Data Writers (each)	Total # of Params	Productivity Gain
5	215	34%
10	430	169%
20	860	438%
40	1720	975%
80	3440	2050%

Table 8: Productivity Gains using DQML's DBE Interpreter

statements (*i.e.*, 160) and subtracting 1 to account for the baseline manual implementation (*i.e.*, ((# of data reader and writer parameters)/160) - 1). The gains increase faster than the increase in the number of data readers and data writers (*e.g.*, the gain for 10 data readers and data writers is more than twice as much for 5 data readers and data writers) showing that productivity gains are greater when more entities are involved.

The interpreter justification analysis shown relates to implementing a single QoS policy configuration. The analysis includes neither the scenario of modifying an existing valid configuration nor the scenario of implementing new configurations for DBE where no modifications to the interpreter code would be required. Changes made even to an existing valid configuration 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, the parameters of the various QoS policies being modified. These challenges are as applicable when changing an already valid QoS policy configuration as they are when creating an initial configuration. Moreover, the complexity for developing a new interpreter for some other application is ameliorated by having the DBE interpreter as a template for traversing a model in BON2.

In large-scale DDS systems (*e.g.*, shipboard computing, air-traffic management, and scientific space missions) there may be thousands of data readers and writers. As a point of reference with 1,000 data readers and 1,000 data writers, the number of QoS parameters to manage is 43,000 (*i.e.*, 18 * 1,000 + 25 * 1,000). 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).

The productivity analysis approach taken for DQML's DBE interpreter is applicable to other DSMLs since the complexities involved will be similar. A break-even point for the development effort of an interpreter for any DSML will exist. We outline four areas that directly influence this break-even point: number of entities, complexity of the entities, complexity of associations between the entities, and level of maintainability needed.

The number of entities affects the break-even point for interpreter development since the more entities that are to be considered the less likely any one individual will be able to manage these entities appropriately. Miller [33] has shown that humans can process up to approximately 7 items of information at a time. This guideline of 7 can be helpful in exploring the break-even point for interpreter development. If there are more than 7 entities to be considered then the accidental complexity increases since the developer must manage the entities using some tool or device (*e.g.*, a piece of paper, a database) external to the person. With this external management comes the possibility of introducing errors in the use of the management tool (*e.g.*, incorrectly transcribing the entities from the developer's head to the tool).

Likewise, this same analysis holds for the complexity of entities as determined by the number of fields or parameters. If an entity contains more than 7 fields then some external tool should be used to manage this complexity. The use of a tool introduces accidental complexity (*e.g.*, incorrectly transcribing the order, names, or types of the parameters). The same analysis can also be applied to the number of associations made between entities to determine that complexity as well as the number of times a configuration will need to be modified.

If any one of these four areas exceeds the threshold of 7 then an interpreter might be warranted. If more than one of these areas exceeds the threshold (*e.g.*, more than 7

entities with more than 7 associations between the entities) then the break-even point for an interpreter is lowered. The exact determination for justifying interpreter development will vary according to the application but the guidelines presented can provide coarse-grained justification.

II.6 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 is 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)) [41], [54]. There is a fairly steep learning curve for OCL especially since it not a programming language but rather a constraint language where developers write rules that enforce the desired constraints. OCL does not support some of the features with which computer programmers are comfortable. There is no program and so there is no program to debug. Also, different modeling tools may implement OCL in different ways so that a developer using OCL with one modeling tool may need to be aware of subtle differences when using a different modeling tool.

• **Management of QoS policy configurations is essential for large-scale systems.** Management of 1,000s of entities and QoS policies can be tedious and error-prone. Maintaining the overall global perspective and supporting the low-level view of a single entity or QoS policy is critical to understand overall system QoS. We plan to address this topic in future work via automated model scalability [39].

• **DSMLs should build upon pattern knowledge.** A DSML can benefit from the knowledge already documented in configuration patterns by incorporating these patterns into the DSML itself. This approach ensures that different types of patterns provide semantic compatibility and are implemented correctly. We are therefore targeting future DQML enhancements to support patterns and higher level services (*e.g.*, security and fault tolerance [96]).

• **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 [44] into DQML to enhance QoS policy configuration development.

• **Leveraging a DSML into an existing modeling tool chains increases its applicability.** DSMLs can be incorporated into existing modeling tool chains to expand scope and utility. We are incorporating DQML into MDE tool chains [35] that handle packaging and deployment.

• **Trade-offs and the break-even point for DSMLs must be clearly understood and communicated.** There are pros and cons to any technical approach including DSMLs. The use of DSMLs may not be appropriate for every case and these cases must be evaluated to provide balanced and objective analysis. For a DSML product line, the advantages of DSMLs will typically outweigh the development costs. For a one-time point solution the development of a DSML may not be justified, depending on the complexity of the domain.

• **The context for DSML productivity analysis should be well defined.** Broad generalizations of a DSML being "X" times better than some other technology is not particularly helpful for comparison and evaluation. A representative case study can be useful to provide a concrete context for productivity analysis.

• **Provide analysis for as minimal or conservative a scenario as possible.** Using a minimal scenario in productivity analysis allows developers to extrapolate to larger scenarios where the DSML use will be justified.

In summary, our experiences developing and applying DQML showed that DQML is an attractive tool for managing the QoS policy configuration complexity since it detects design mistakes early in the development cycle. DQML is particularly appealing since it alleviates the accidental complexities associated with managing (1) the validity and number of associations among DDS entities, (2) the validity and number of associations between DDS entities and QoS policies, and (3) the number, types, and valid values for QoS policy parameters. DQML also provides automated generation of QoS policy configuration artifacts (*e.g.*, policy configuration files) that can be seamlessly incorporated into system

development, thereby reducing costs and increasing confidence. In particular, a DSML like DQML can be used to ensure that the desired QoS of the system is the implemented QoS of the system, thereby reducing costs and effort while increasing confidence in the system as deployed.

GME can be downloaded from `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`.

CHAPTER III

EMPIRICAL EVALUATION OF QOS MECHANISMS FOR QOS-ENABLED PUB/SUB MIDDLEWARE

Chapter I 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) providing more detailed context, (2) illustrating a motivating example, (3) outlining existing research in the field of empirical evaluations of QoS mechanisms for pub/sub DRE systems, (4) enumerating unresolved challenges with current research, and (5) resolving the challenges via a solution approach. This chapter also presents empirical metrics data obtained and evaluated using the solution approach.

III.1 Context

Emerging trends and challenges. *Real-time Event Stream Processing* (RT-ESP) applications support mission-critical systems (such as collaboration of weather monitoring radars to predict life-threatening weather [86]) by managing and coordinating multiple streams of event data that have (possibly distinct) timeliness requirements. Streams of event data may originate from sensors (*e.g.*, surveillance cameras, temperature probes), as well as other types of monitors (*e.g.*, online stock trade feeds). These continuously generated data streams differ from streaming the contents of a data file (such as a fixed-size movie) since the end of RT-ESP data is not known *a priori*. In general, streamed file data demand less stringent delivery and deadline requirements, instead emphasizing a continuous flow of data to an application.

RT-ESP applications require (1) *timeliness* of the event stream data and (2) *reliability* so that sufficient data are received to make the result usable. Moreover, RT-ESP applications encompass multiple senders and receivers (*e.g.*, multiple continuous data streams

can be produced and multiple receivers can consume the data streams). With the growing complexity of RT-ESP application requirements (*e.g.*, large number of senders/receivers, variety of event types, event filtering, QoS, and platform heterogeneity), developers are increasingly leveraging pub/sub middleware to help manage the complexity and increase productivity [30, 63].

To address the complex requirements of RT-ESP applications, the underlying pub/sub middleware must support a flexible communication infrastructure. This flexibility requirement is manifest in several ways, including the following:

• Large-scale RT-ESP applications require flexible communication infrastructure due to the complexity inherent in the scale involved. As the number and type of event data streams continue to increase, the communication infrastructure must be able to coordinate these streams so that publishers and subscribers are connected appropriately. Flexible communication infrastructure must adapt to fluctuating demands for various event streams and environment changes to maintain acceptable levels of service.

• Certain types of large-scale RT-ESP applications require a flexible communication infrastructure due to their dynamic and *ad hoc* nature. These application environments incur fluctuations in resource availability as they include mobile assets with intermittent connectivity and underprovisioned or temporary assets from emergency responders. Examples of *ad hoc* large-scale RT-ESP applications include tactical information grids, *in situ* weather monitoring for impending hurricanes, and emergency response networks in the aftermath of regional disasters.

For RT-ESP applications requiring synchronization across a variety of event streams (*e.g.*, weather monitoring, online stock trading, homeland security, and humanitarian relief missions), modern datacenters provide a crucial computing platform for centralized processing [9]. Where the number of comparatively costly mainframe servers has decreased in recent years, the installed base of datacenter-class inexpensive commodity servers have

increased [36]. Moreover, modern datacenters, running heterogeneous systems and software [76] are service-oriented and heavily virtualized [55]. Modern datacenters must also increasingly support applications with *real-time* quality-of-service (QoS) requirements [9]. These datacenters typically comprise low-cost, commodity components [36] that operate in *ad hoc*, tumultuous environments with multiple failure modes ranging in scope from loss of packets to node crashes to large-scale local, regional, and national system failures [34]. Such datacenters must support the timeliness properties of RT-ESP applications as well as recover from failures in relatively short timeframes (*e.g.*, within seconds).

Real-time datacenters require a communication infrastructure that is scalable and flexible. The scalability requirement comes from size of the datacenter itself. For any single RT-ESP application, the datacenter needs to support the number and variety of data types being provided and consumed along with the required QoS. Moreover, the datacenter must be able to manage a wide variety of RT-ESP applications concurrently with event streams being used across applications. The flexibility requirement is manifest in many ways, including the following:

- Large-scale datacenters require flexible communication infrastructure due to the many failure modes and complexity inherent in the scale involved. Flexible communication infrastructure can adapt to (1) fluctuating demands for various event streams and (2) environment changes to maintain acceptable levels of service.

- Certain types of large-scale datacenters aggravate the demands of flexible communication infrastructure due to their dynamic and *ad hoc* nature. Examples of *ad hoc* large-scale datacenters include tactical information grids, *in situ* weather monitoring for impending hurricanes, and emergency response networks in the aftermath of a regional or national disaster.

Several pub/sub middleware platforms have been developed to support large-scale data-centric distributed systems, such as the Java Message Service [72], Web Services Brokered

Notification [66], and the CORBA Event Service [82]. These platforms, however, do not support fine-grained and robust QoS. Some large-scale distributed system platforms, such as the Global Information Grid [1] and Network-centric Enterprise Services, [2], require rapid response, reliability, bandwidth guarantees, scalability, and fault-tolerance. Moreover, these systems are required to perform under stressful conditions and over connections with less than ideal behavior, such as latency and bandwidth variability, bursty loss, and routers quickly alternating destinations (*i.e.*, route flaps).

Developing flexible communication infrastructure to address these challenges is hard because it must have a detailed understanding of the capabilities that the underlying transport protocols provide. The infrastructure must also understand how these protocols behave under different operating conditions stemming from both the application-imposed workload changes, as well as system dynamics, such as failures and network congestion. Building on this understanding, QoS-enabled pub/sub middleware can help alleviate the complexity of managing multiple event streams and maintaining real-time QoS for multiple event streams in highly dynamic environments.

Solution approach → A FLEXible Middleware And Transports (FLEXMAT) Evaluation Framework.

This chapter describes the design and capabilities of the *FLEXible and Integrated Middleware and Transport Evaluation Framework* (FLEXMAT) to address these requirements. To evaluate the impact of various transport protocols that can lead to the realization of a QoS-enabled pub/sub middleware we developed *ReLate2*, which is a composite metric for FLEXMAT that considers both reliability and latency. We use the ReLate2 composite QoS metric to evaluate the reliability and latency of transmitted data for various experimental configurations involving parameters such as sending rate, network loss, and number of receivers.

To facilitate the empirical benchmarking environment, and collection of the Relate2 metrics, FLEXMAT integrates and enhances the following capabilities:

• The *Adaptive Network Transports* (ANT) framework, which provides infrastructure for composing transport protocols that builds upon properties provided by the scalable reliable multicast-based Ricochet transport protocol [9]. Ricochet enables trade-offs between latency and reliability, which are needed qualities for pub/sub middleware supporting RT-ESP applications. Ricochet also supports modification of parameters to affect latency, reliability, and bandwidth usage.

• OpenDDS [80], which is an open-source implementation of the OMG Data Distribution Service (DDS) [84] standard that enables applications to communicate by publishing information they have and subscribing to information they need in a timely manner. OpenDDS provides support for various transport protocols, including TCP, UDP, IP multicast, and a reliable multicast protocol. OpenDDS also provides a pluggable transport framework that allows integration of custom transport protocols within OpenDDS.

We apply the ReLate2 metric across various commonly used and custom FLEXMAT transport protocols. We then empirically quantify the results and analyze the pros/cons of various transport protocol configurations in the context of FLEXMAT. By capturing the insights gained from this effort, our goal is to enhance the development and validation of QoS-enabled pub/sub middleware.

III.2 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

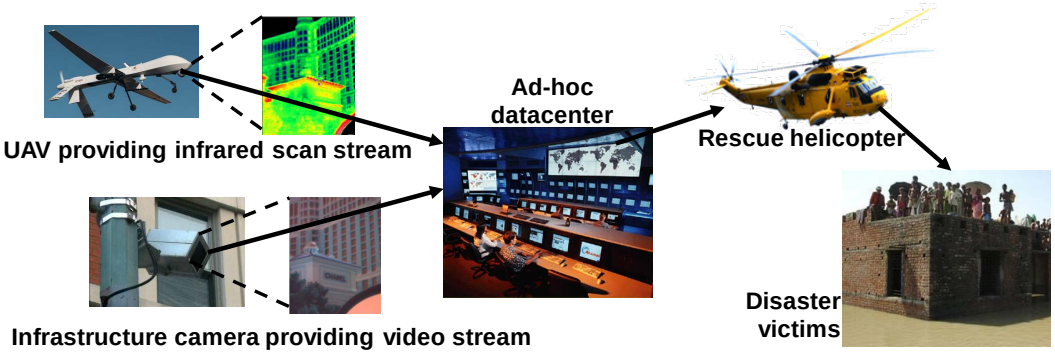

Figure 17: Search and Rescue Motivating Example

event stream data from various sensors and monitors to emergency vehicles that can be dispatched to areas where survivors are identified.

Figure 17 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 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 18 shows how fire detection applications and power grid assessment applications can use infrared scans to detect fires and working HVAC systems respectively. Likewise, Figure 19 shows how security monitoring and structural damage applications can use video stream data to detect looting and unsafe buildings respectively. Section III.5.1.2 describes how FLEXMAT addresses this challenge by incorporating transport protocols that balance reliability and low latency.

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

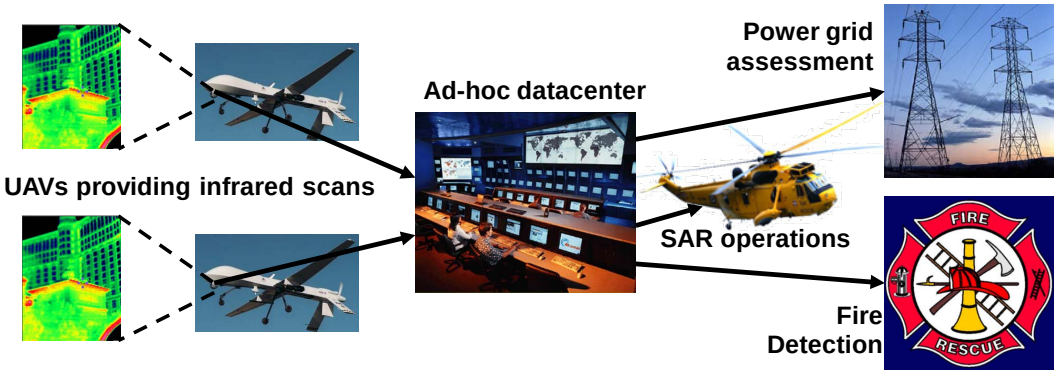

Figure 18: Uses of Infrared Scans during Disaster Recovery

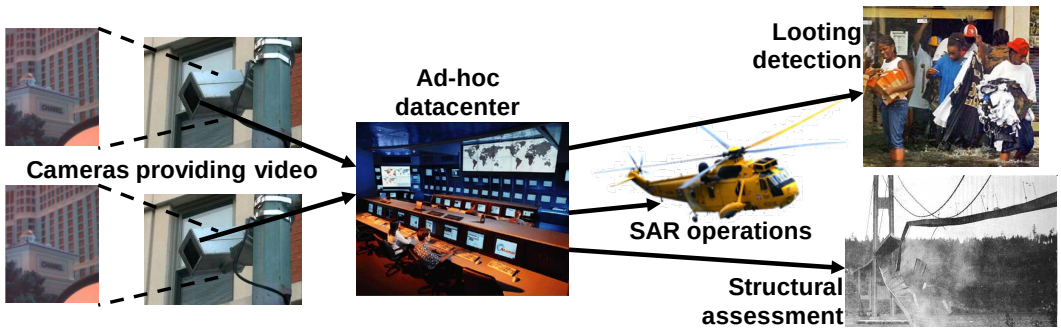

Figure 19: Uses of Video Stream during Disaster Recovery

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 III.5.1.1 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 II.5.3 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 III.5.1.2 describes how we are incorporating flexible transport protocols that can be easily adjusted for reliability, latency, and/or network bandwidth usage.

III.3 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.* [9] evaluate the performance of the Ricochet transport protocol with the Scalable Reliable Multicast (SRM) protocol [31]). Bateman *et al.* [12] compare the performance of TCP variations both using simulations and in a testbed. Cheng *et al.* [26] provide performance comparisons of UDP and TCP for video streaming in multihop wireless mesh networks. Kirschberg *et al.* [61] 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.* [106] 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.* [89] 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 mechanims for the middleware to evaluate its performance.

Tanaka *et al.* [97] 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.* [101] 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.* [13] 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 [15] 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.

III.4 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 quanitfied.

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.

III.5 Solution Approach: FLEXible Middleware And Transports (FLEXMAT)

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 utilizes the ReLate2 composite QoS metrics to aid in evaluating the QoS properties of transport protocols. Moreover, this section includes empirical evaluations of several transport protocols utilizing FLEXMAT and ReLate2.

III.5.1 The Structure and Functionality of FLEXMAT and the ReLate2 Composite QoS Metric

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.

III.5.1.1 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 III.2 by providing anonymous publication and subscription via the OMG Data Distribution Service (see Sidebar II.5.1.1 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 facilities 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 [32] and Component Configurator [92]) to provide flexibility and delegate responsibility to the protocol only when applicable.

III.5.1.2 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.

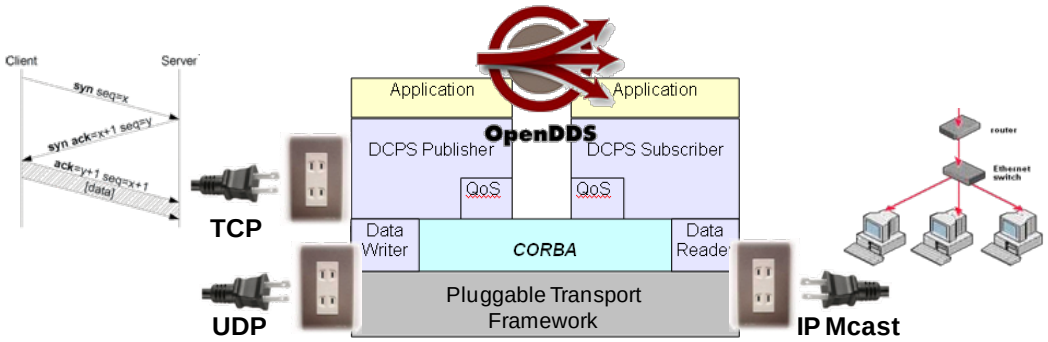

Figure 20: OpenDDS and its Transport Protocol Framework

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 20. OpenDDS TCP is a reliable unicast protocol, whereas UDP is an unreliable unicast protocol. IP Mcast can send data to multiple receivers.

While TCP, UDP, and IP Mcast are standard protocols, RMcast warrants more description. It is a negative acknowledgment (NAK) protocol that provides reliability. as shown in Figure 21. In this 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 III.2 by providing reliability and timeliness for certain environment configurations.

Adaptive Network Transport Protocols. The ANT transport protocol framework

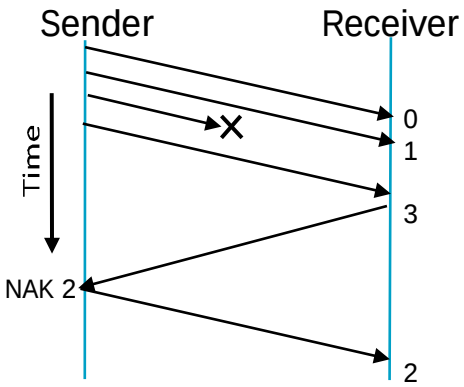

Figure 21: A NAK Based Protocol Discovering Loss

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 [9] 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 III.5.1.2. 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

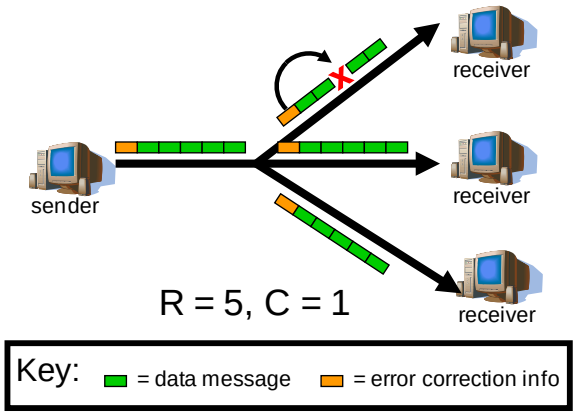

Figure 22: FEC Reliable Multicast Protocol - Sender-based

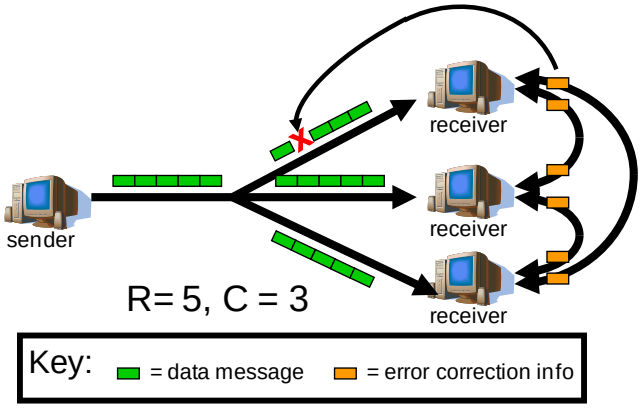

Figure 23: FEC Reliable Multicast Protocol - Receiver-based (LEC)

proactively send redundant information to recover from this loss. Sender-based FEC protocols have the sender send redundant information, as shown in Figure 22. In contrast, receiver-based FEC (a.k.a. Lateral Error Correction (LEC)) have receivers send each other redundant information as shown in Figure 23. The Ricochet protocol we employ in FLEXMAT is an example of an LEC protocol.

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

Protocol	Integrator	Functionality
TCP	OpenDDS	unicast, reliable, packet ordering, flow control
UDP	OpenDDS	unicast, unreliable
IP Mcast	OpenDDS	multicast, unreliable
RMcast	OpenDDS	multicast, reliable, packet ordering, NAK-based
ANT Baseline	ANT	multicast, unreliable
ANT NAKcast	ANT	multicast, reliable, NAK-based
ANT Ricochet	ANT	multicast, probabilistically reliable

Table 9: Transport Protocols Evaluated

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 II.5.3, 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 III.2 by providing high probabilistic reliability and low latency error detection and recovery. Ricochet also helps resolve Challenge 4 in Section III.2 by supporting tunable parameters that effect reliability, latency, and bandwidth usage. We designed the ANT framework so that different transport protocols can be switched dynamically. Table 9 presents a summary of all protocols we included in our experiments in Section III.5.2.

III.5.1.3 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 III.5.2.

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.

We initially designed the *ReLate* metric to account for both latency and reliability in a fairly straightforward manner. ReLate divided the average latency of data packets for an experiment using a particular protocol by the percentage of packets received. This metric then accounted for reliability and latency. In particular, if latencies were equal between two protocols then the protocol that delivered the most packets would have the lowest value. The formula for ReLate is defined as:

$$ReLate_p = \frac{\sum_{i=1}^{r} l_i}{r} \div \frac{r}{t}$$

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.

The initial ReLate metric is helpful only for evaluating protocols that balance reliability and latency. This metric does not help us evaluate all the protocols that we have currently used, in particular the protocols that provide no reliability. Using this initial metric produces values that are lower than those for the reliable multicast and Ricochet protocols even with a significant percentage of network loss (*e.g.*, 3%).

For example, using the values from one of our experiments outlined in Section III.5.2.2,

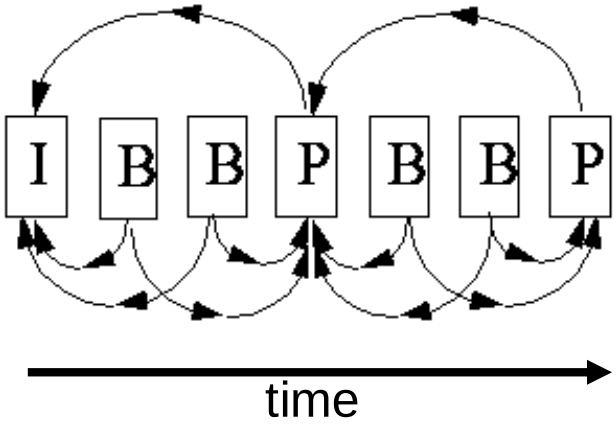

Figure 24: MPEG Frame Dependencies

the ReLate metric produces the lowest values for OpenDDS UDP, OpenDDS IP Mcast, and ANT Baseline even with 3% packet loss. None of these transport protocols with the lowest ReLate value provide reliability. Figure 32 in Section III.5.2.2 presents these results in the context of FLEXMAT.

For RT-ESP applications involving multimedia, such as our motivating example of SAR operations in Section III.2, over 10% loss is generally considered unacceptable. Bai and Ito [8] 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 [7]. Ngatman et al. [78] 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 24, 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.

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:

$$ReLate2_p = \frac{\sum_{i=1}^{r} l_i}{r} \times (\frac{t-r}{t} \times 100 + 1)$$

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 III.5.2.2 uses the ReLate2 metric to determine the transport protocols that best balance reliability and latency.

III.5.2 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 III.2.

III.5.2.1 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 III.2, we are concerned with the distribution of data for SAR datacenters, where network packets are dropped at end hosts [10]. 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 III.5.1.3, 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

Point of Variability	Values
Number of receiving data writers	3 - 10
Frequency of sending data	25 Hz, 50 Hz
Percent end-host network loss	0 to 3 %

Table 10: Emulab Variables

Point of Variability	Values
Number of receiving data writers	3 - 25
Frequency of sending data	10 Hz, 25 Hz, 50 Hz, 100 Hz
Percent network loss	0 to 5 %

Table 11: ISISlab Variables

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 50Hz 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 III.5.1.2, 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 10 outlines the points of variability for the Emulab experiments.

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 11 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 III.5.2.2.

III.5.2.2 Results and Analysis of Experiments

This section presents and analyzes the results from our experiments, which resolves Challenge 3 in Section III.2 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 III.5.1.2. These experiments used Emulab as described in Section III.5.2.1. 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 25 and 26, 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.

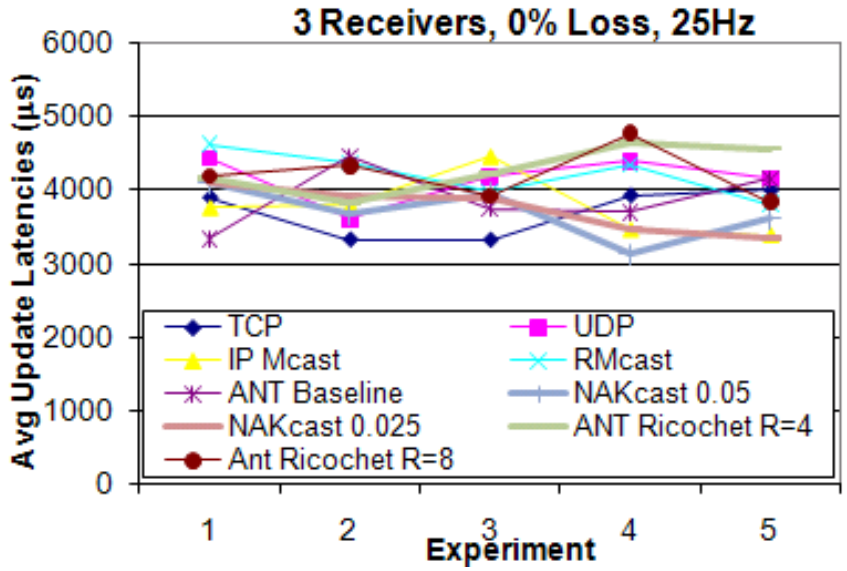

Figure 25: Emulab: Average update latency, 3 readers, 0% loss, 25Hz

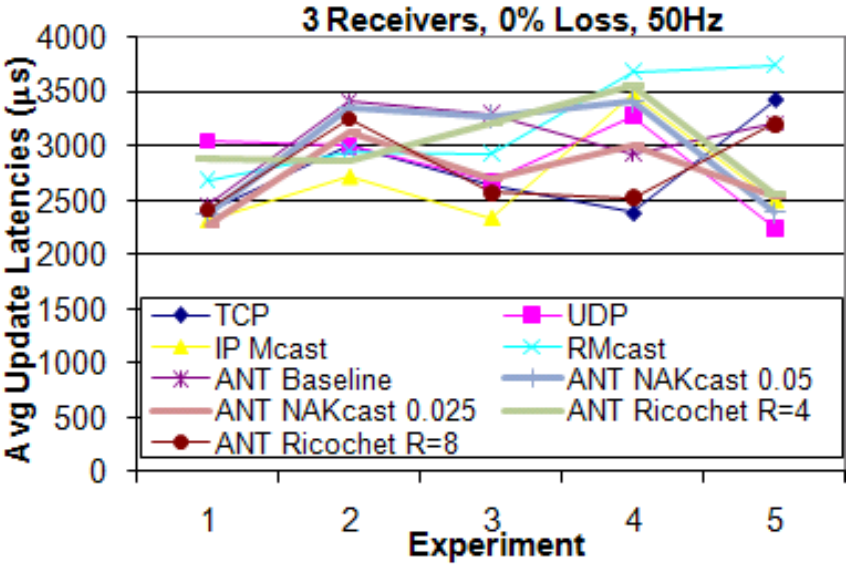

Figure 26: Emulab: Average update latency, 3 readers, 0% loss, 50Hz

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 27, 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 28 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 29 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 30 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.

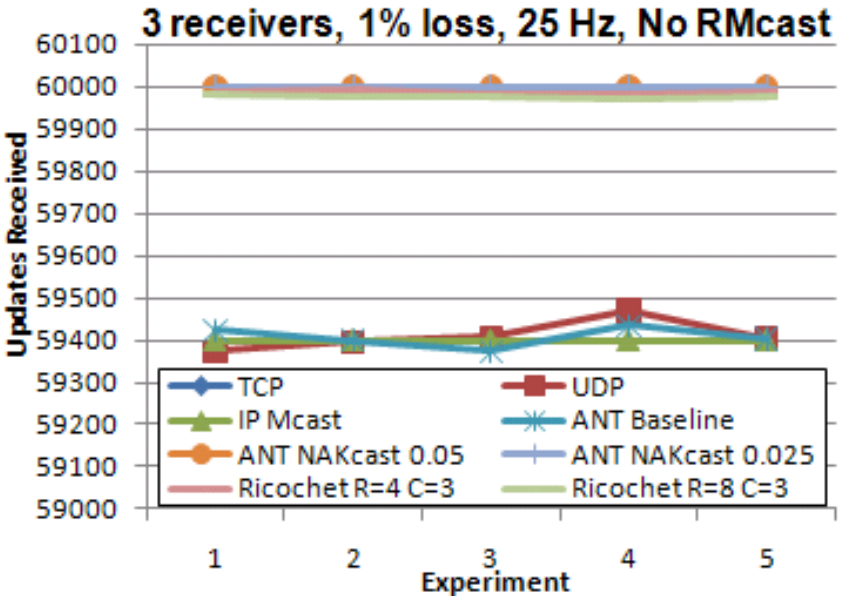

Figure 27: Emulab: Updates received, 3 readers, 1% loss, 25Hz, no RMcast

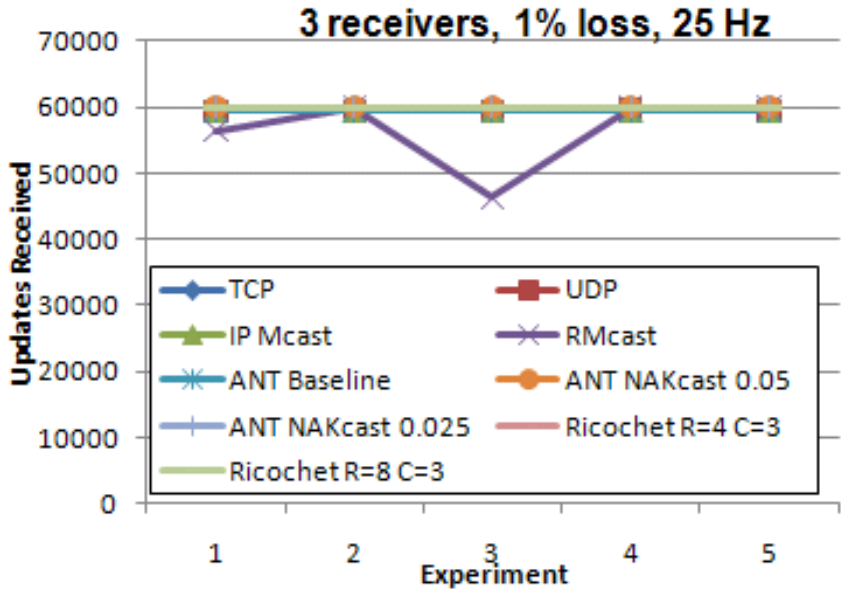

Figure 28: Emulab: Updates received, 3 readers, 1% loss, 25Hz

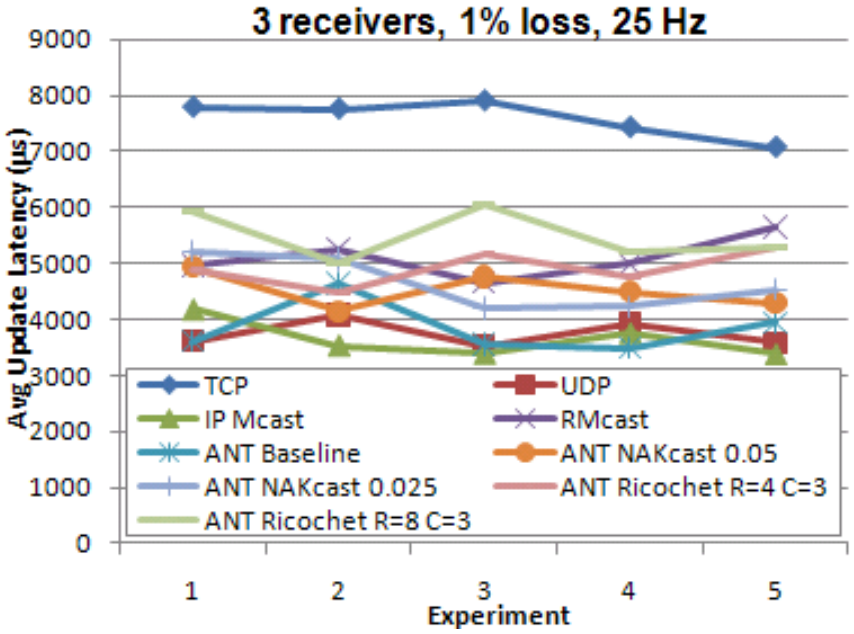

Figure 29: Emulab: Average update latency, 3 readers, 1% loss, 25Hz

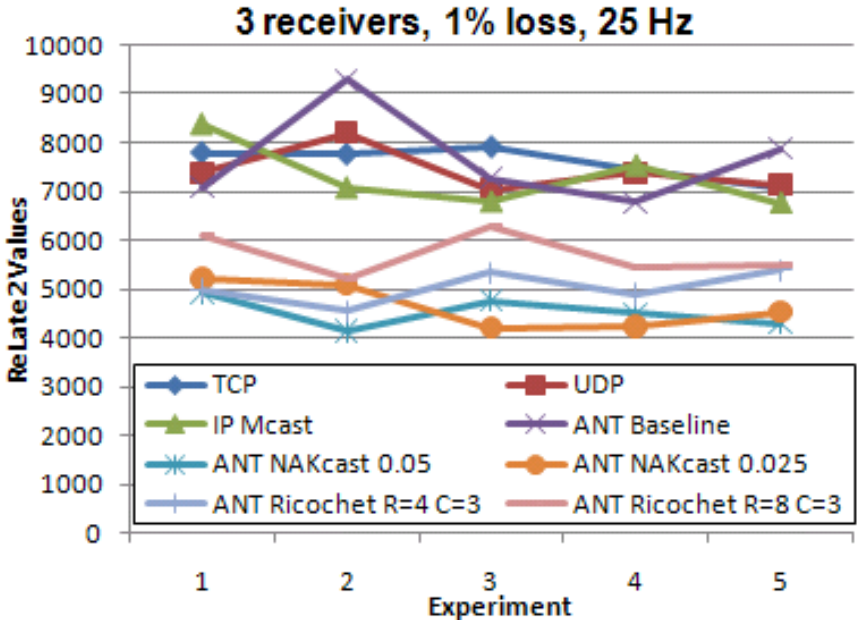

Figure 30: Emulab: Relate2 values, 3 readers, 1% loss, 25Hz

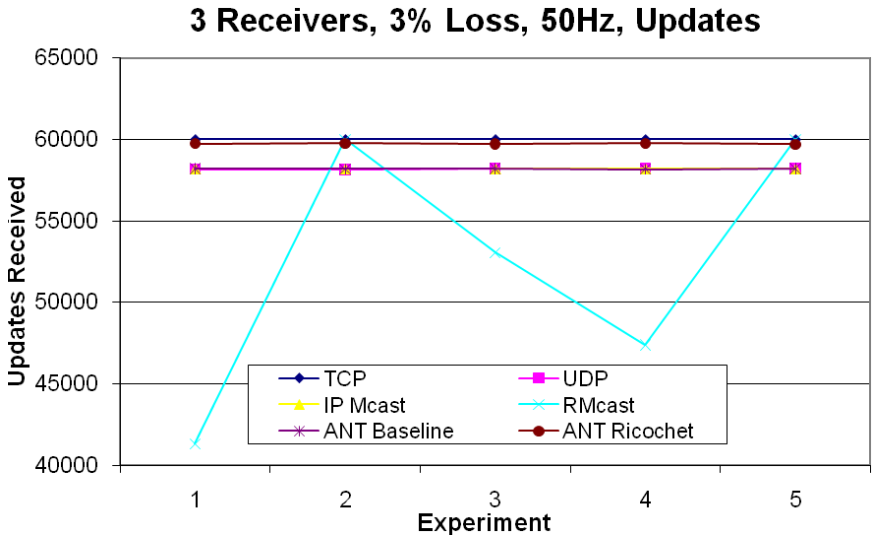

Figure 31: Emulab: Updates Received, 3 readers, 3% loss, 50 Hz

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 III.5.1.2. We utilize the ReLate2 metrics defined in Section III.5.1.3 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 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.

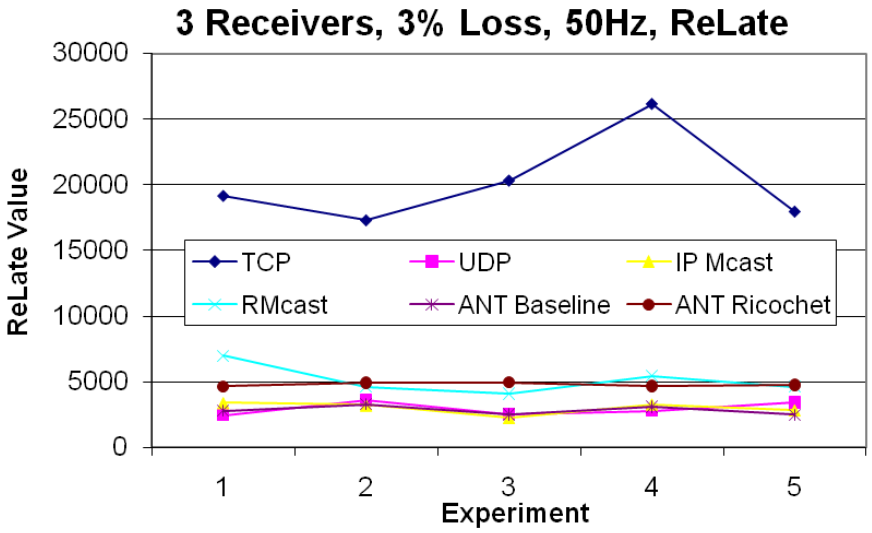

Figure 32: ReLate Metrics for Emulab Experiment: 3 readers, 3% loss, 50 Hz update rate

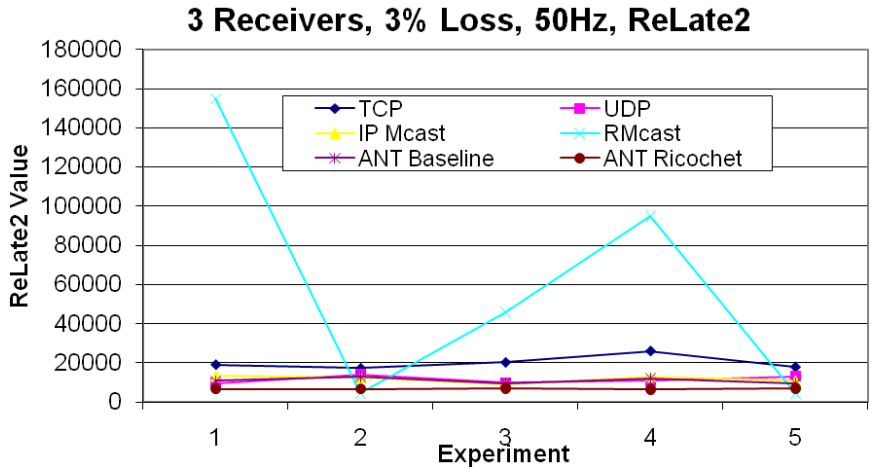

Figure 33: ReLate2 Metrics for Emulab Experiment: 3 readers, 3% loss, 50 Hz update rate

We compare the values from the ReLate2 metric as shown in Figure 33 with the values in Figure 32 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 III.5.2.1, 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 available and thus we did not require Emulab's network traffic shaping capability.

As with the Emulab experiments in Section III.5.2.2, 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 behavorial distinctions of the protocols.

Figures 34 and 35 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 34 and 35 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 as NAKcast with reliability rates ranging from 99.97% to 99.99%. This reliability is fairly constant across the different sending rates.

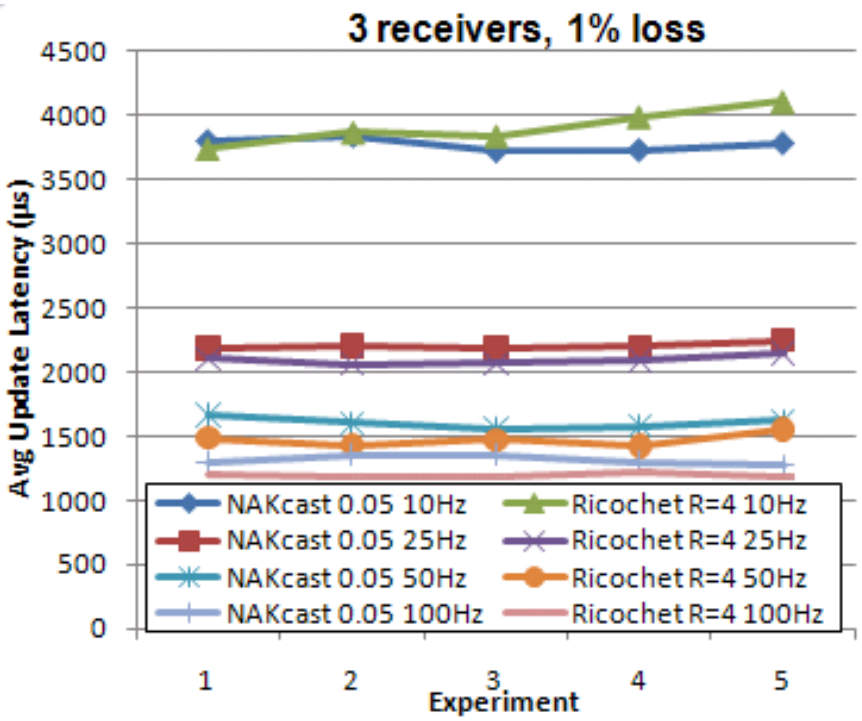

Figure 34: ISISlab: Average update latency, 3 readers, 1% loss

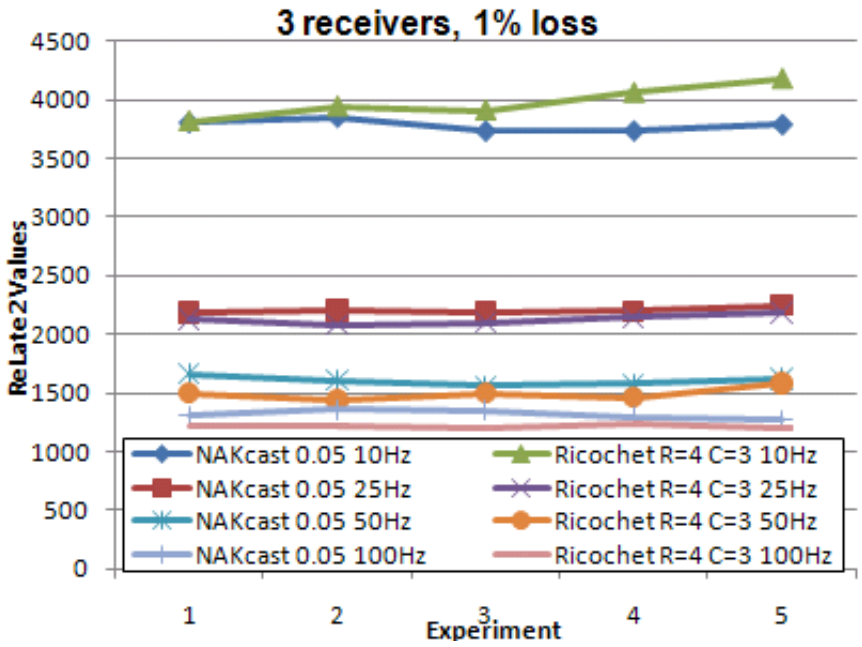

Figure 35: ISISlab: ReLate2 values, 3 readers, 1% loss

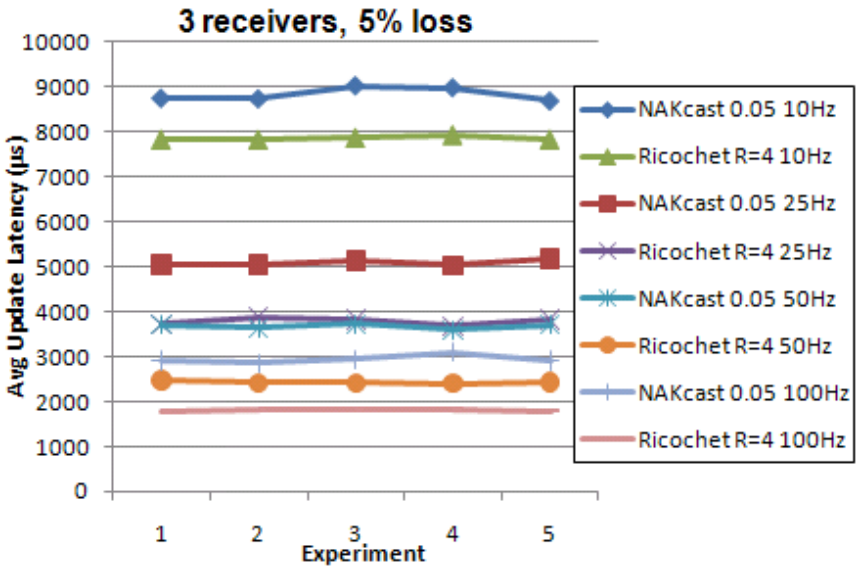

Figure 36: ISISlab: Average update latency, 3 readers, 5% loss

Figures 36 and 37 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 38 and 39 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 40 and 41 show the effect on the protocols of increasing the number of

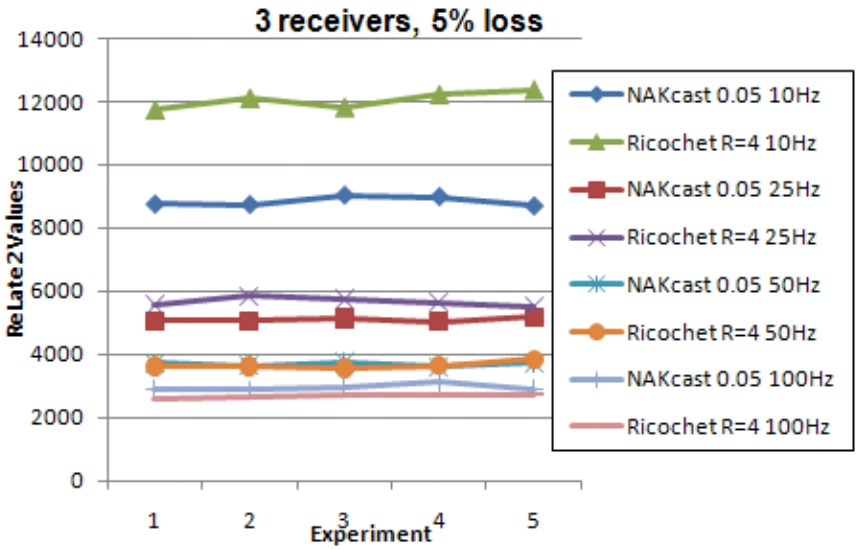

Figure 37: ISISlab: ReLate2 values, 3 readers, 5% loss

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.

Analysis of Experimental Results. The results in this section show that for a set protocol configuration there are performance 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

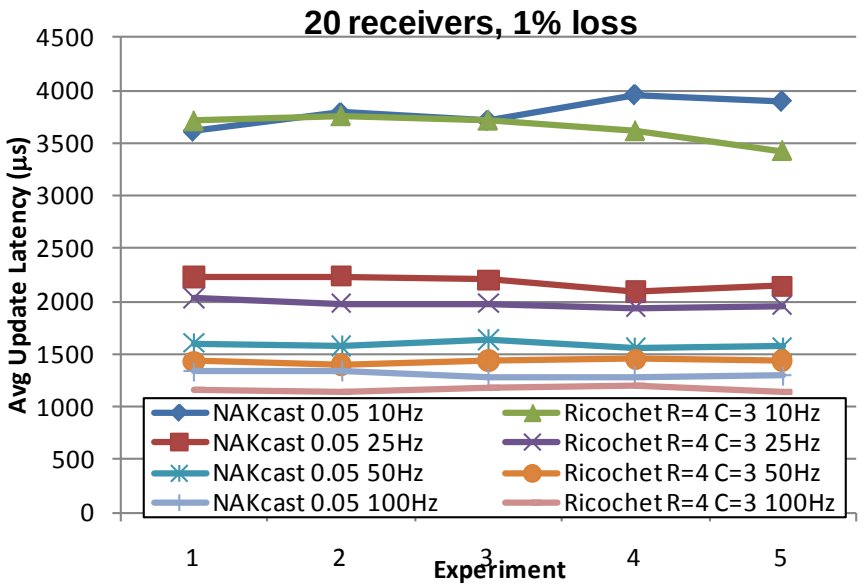

Figure 38: ISISlab: Average update latency, 20 readers, 1% loss

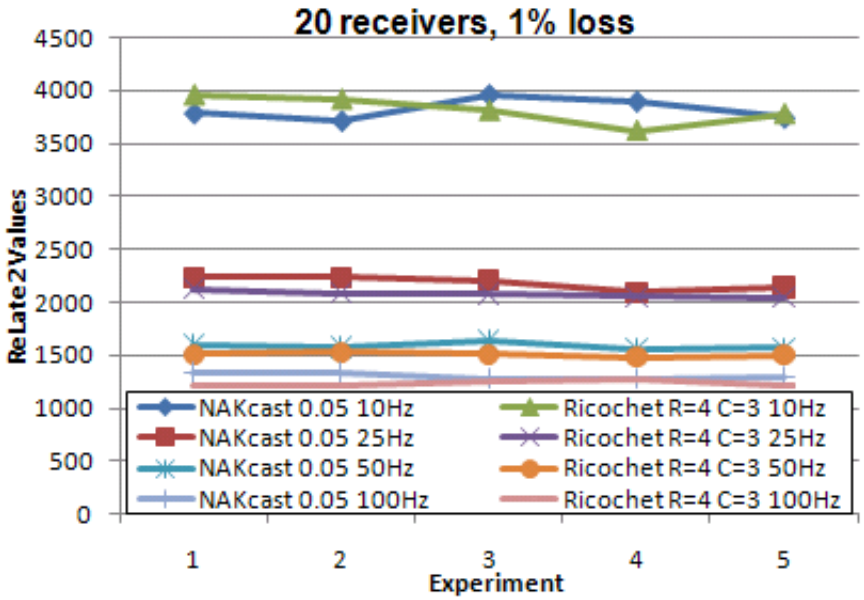

Figure 39: ISISlab: ReLate2 values, 20 readers, 1% loss

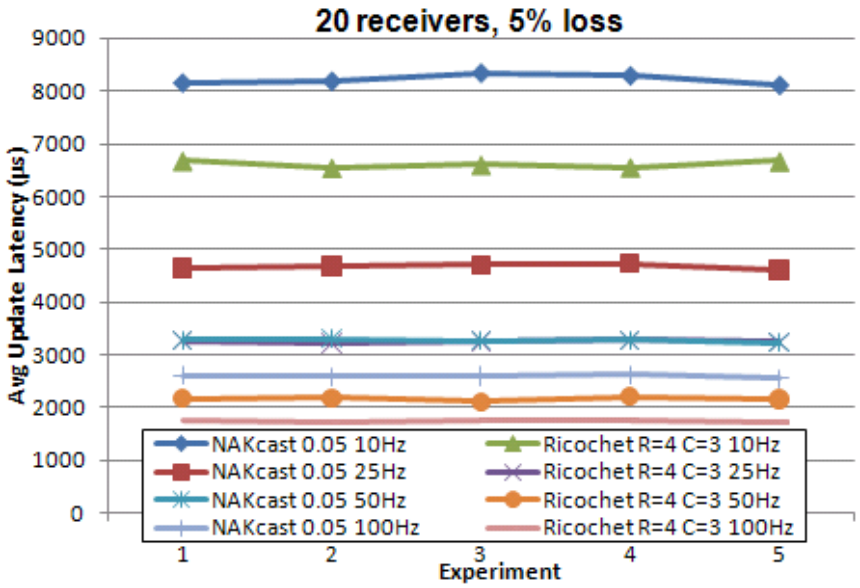

Figure 40: ISISlab: Average update latency, 20 readers, 5% loss

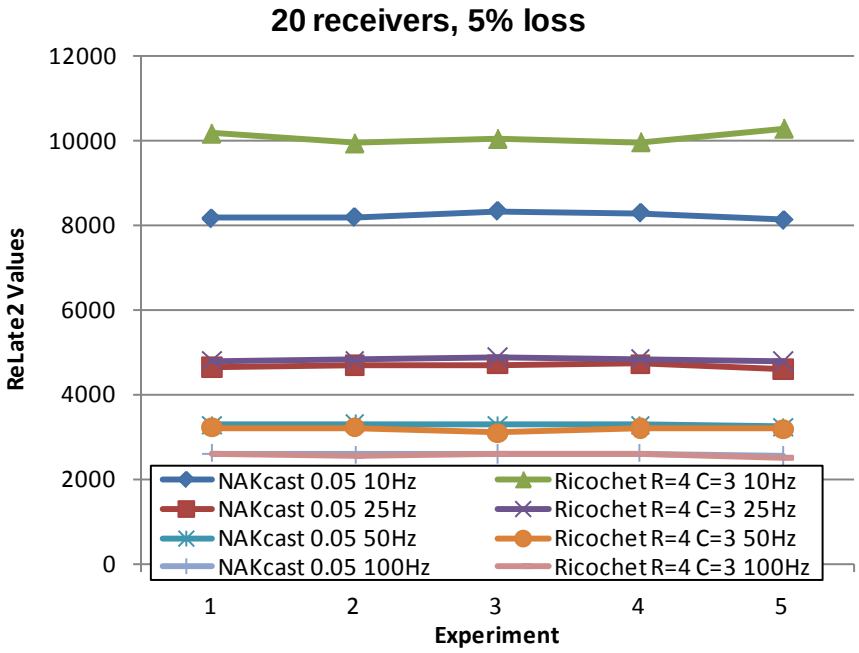

Figure 41: ISISlab: ReLate2 values, 20 readers, 5% loss

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.

III.6 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 section defined 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 III.5.2.2) 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 III.5.2.2 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.

One approach includes guidance to the application developer via patterns such as those proposed by Hunt [51]. Even with the aid of patterns, however, there is no guidance for

implementation. Another approach is to have protocols register their supporting QoS properties (*e.g.*, level of reliability) with the middleware. The middleware can then automatically select transport protocols based on the QoS properties specified by the developer. A third approach involves model-driven engineering [91] where a 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.

• **Determining when one transport protocol provides advantages over another in pub/sub middleware can be challenging.** There are often many factors to consider when selecting a transport protocol to be used with pub/sub middleware. One protocol might produce the lowest packet latency but not deliver all the messages. Additionally, when protocols do not perform as expected, particularly when scaling beyond previously tested bounds, pinpointing the source of the irregularity takes time as the problem could be in the transport protocol, the pub/sub middleware, or the integration between the two.

• **High-level metrics are useful to quickly differentiate the performance of various configurations.** The use of metrics—even if coarse-grained—help explore a large configuration space. Part of the impetus in developing the ReLate and ReLate2 metrics (see Section III.5.1.3) 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 III.5.2.2. 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`.

CHAPTER IV

AUTONOMIC ADAPTATION OF QOS-ENABLED PUB/SUB MIDDLEWARE

Chapter I presented an overview of the need for autonomic adaptation of QoS-enabled pub/sub DRE middleware. In particular, autonomic adaptation is needed for both (1) configuration adaptation within flexible environments where the operating environment (*e.g.*, available resources) is not known until runtime, and (2) ongoing modifications within dynamic environments where the operating environment changes while the system is running. This chapter presents more in-depth information for both of these areas by (1) detailing a motivating example for each type of adaptation to highlight adaptation's broad applicability, (2) outlining existing research in the field of autonomic adaptation of middleware and software system infrastructure, (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.

IV.1 Autonomic Configuration in Flexible Environments

This section details the context, challenges, our solution approach, and results for autonomically configuring QoS-enabled pub/sub middleware for flexible computing environments.

IV.1.1 Context

Emerging trends and challenges. Enterprise distributed real-time and embedded (DRE) publish/subscribe (pub/sub) systems manage data and resources that are critical to the ongoing system operations. Examples include testing and training of experimental aircraft

across a large geographic area, air traffic management systems, and disaster recovery operations. These types of enterprise DRE systems must be configured correctly to leverage available resources and respond to the system deployment environment. For example, search and rescue missions in disaster recovery operations need to configure the image resolution used to detect and track survivors depending on the available resources (*e.g.*, computing power and network bandwidth) [93].

Many enterprise DRE systems are implemented and developed for a specific computing/networking platform and deployed with the expectation of specific computing and networking resources being available at runtime. This approach simplifies development complexity since system developers need only focus on how the system behaves in one operating environment. Thus considerations of multiple infrastructure platforms are ameliorated with respect to system QoS properties (*e.g.*, responsiveness of computing platform, latency and reliability of networked data, etc.). Focusing on only a single operating environment, however, decreases the flexibility of the system and makes it hard to integrate into different operating environments (*e.g.*, porting to new computing and networking hardware).

Cloud computing [21, 77] is an increasingly popular infrastructure paradigm where computing and networking resources are provided to a system or application as a service—typically for a "pay-as-you-go" usage fee. Provisioning services in cloud environments relieve enterprise operators of many tedious tasks associated with managing hardware and software resources used by systems and applications. Cloud computing also provides enterprise application developers and operators with additional flexibility by virtualizing resources, such as providing virtual machines that can differ from the actual hardware machines used.

Several pub/sub middleware platforms (such as the Java Message Service [72], and Web Services Brokered Notification [66]) can (1) leverage cloud environments, (2) support large-scale data-centric distributed systems, and (3) ease development and deployment of these systems. These pub/sub platforms, however, do not support fine-grained and robust

QoS that are needed for enterprise DRE systems. Some large-scale distributed system platforms, such as the Global Information Grid [1] and Network-centric Enterprise Services [2], require rapid response, reliability, bandwidth guarantees, scalability, and fault-tolerance.

Conversely, conventional cloud environments are problematic for enterprise DRE systems since applications within these systems often cannot characterize the utilization of their specific resources (*e.g.*, CPU speeds and memory) accurately *a priori*. Consequently, applications in DRE systems may need to adjust to the available resources supplied by the cloud environment (*e.g.*, using compression algorithms optimized for given CPU power and memory) since the presence/absence of these resources affect timeliness and other QoS properties crucial to proper operation. If these adjustments take too long the mission that the DRE system supports could be jeopardized.

Configuring an enterprise DRE pub/sub system in a cloud environment is hard because the DRE system must understand how the computing and networking resources affect end-to-end QoS. For example, transport protocols provide different types of QoS (*e.g.*, reliability and latency) that must be configured in conjunction with the pub/sub middleware. To work properly, however, QoS-enabled pub/sub middleware must understand how these protocols behave with different cloud infrastructures. Likewise, the middleware must be configured with appropriate transport protocols to support the required end-to-end QoS. Manual or *ad hoc* configuration of the transport and middleware can be tedious, error-prone, and time consuming.

Solution approach → Supervised Machine Learning for Autonomous Configuration of DRE Pub/Sub Middleware in Cloud Computing Environments. This section describes how we are (1) evaluating multiple QoS concerns (*i.e.*, reliability and latency)

based on differences in computing and networking resources and (2) configuring QoS-enabled pub/sub middleware autonomically for cloud environments based on these evaluations. We have prototyped this approach in the *ADAptive Middleware And Network Transports* (ADAMANT) platform, which addresses the problem of configuring QoS-enabled DRE pub/sub middleware for cloud environments. Our approach provides the following contributions to research on autonomic configuration of DRE pub/sub middleware in cloud environments:

• **Supervised machine learning as a knowledge base to provide fast and predictable resource management in cloud environments.** *Artificial Neural Network* (ANN) tools determine in a timely manner the appropriate transport protocol for the QoS-enabled pub/-sub middleware platform given the computing resources available in the cloud environment. ANN tools are trained on particular computing and networking configurations to provide the best QoS support for those configurations. Moreover, they provide predictable response times needed for DRE systems.

• **Configuration of DRE pub/sub middleware based on guidance from supervised machine learning.** Our ADAMANT middleware uses the *Adaptive Network Transports* (ANT) [45] framework to select the transport protocol(s) that best addresses multiple QoS concerns for given computing resources. ANT provides infrastructure for composing and configuring transport protocols such as the scalable reliable multicast-based Ricochet transport protocol [9]. Supported protocols such as Ricochet enable trade-offs between latency and reliability to support middleware for enterprise DRE pub/sub systems in cloud environments.

We have implemented ADAMANT using multiple open-source pub/sub middleware implementations (*i.e.*, OpenDDS [80] and OpenSplice [90]) of the OMG Data Distribution Service (DDS) [84] specification. DDS defines a QoS-enabled DRE pub/sub middleware standard that enables applications to communicate by publishing information they have and subscribing to information they need in a timely manner. The OpenDDS and OpenSplice

implementations of DDS provide pluggable protocol frameworks that can support standard transport protocols (such as TCP, UDP, and IP multicast), as well as custom transport protocols (such as Ricochet and reliable multicast).

We validated ADAMANT by configuring Emulab (`www.emulab.net`) to emulate a cloud environment that allows test programs to request and configure several types of computing and networking resources on-demand. We then applied several composite metrics developed to ascertain how ADAMANT supports relevant QoS concerns for various Emulab-based cloud configurations. These metrics quantitatively measure multiple inter-related QoS concerns (*i.e.*, latency and reliability) to evaluate QoS mechanisms (such as transport protocols) used in QoS-enabled pub/sub DRE systems. Our supervised machine learning tools use the results of these composite metrics to determine the most appropriate transport protocol to apply in the Emulab cloud environment.

IV.1.2 Motivating Example - Search and Rescue Operations in Cloud Computing Environments

This section builds on the Search and Rescue Operations motivating example described in Section III.2 by leveraging resources in a cloud computing environment to motivate the runtime configuration challenges that ADAMANT addresses.

IV.1.2.1 Search and Rescue Operations for Disaster Recovery

To highlight the challenges of configuring enterprise DRE pub/sub systems for cloud environments in a timely manner, our work is motivated in the context of supporting search and rescue (SAR) operations, as previously described, but which are able to leverage cloud infrastructure. In addition to using 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 data from various sensors and

monitors to emergency vehicles that can be dispatched to areas where survivors are identified, the datacenters can be mobile (*e.g.*, in truck trailers or large command-and-control aircraft if roads are damaged) and brought into the disaster area as needed. Moreover, these datacenters can be connected to cloud infrastructure via high-speed satellite links [52] since ground-based wired connectivity may not be available due to the disaster. In particular, our work focuses on configuring the QoS-enabled pub/sub middleware used by the temporary *ad hoc* datacenter for data dissemination.

Figure 17 in Section III.2 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 facilitated by cloud infrastructure where the data are disseminated, received by fusion applications, and processed to detect survivors. Once survivors are detected, the SAR system will develop a three dimensional view and highly accurate position information so that rescue operations can commence.

A key requirement of data fusion applications within the datacenter is the 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 thereby expending scarce resources such as rescue workers, rescue vehicles, and data center coordinators unnecessarily. The timeliness and reliability properties of the data are affected by the underlying hardware infrastructure (*e.g.*, faster processors and networks can decrease latency and allow more error correcting data to be transmitted to improve reliability).

SAR operations in the aftermath of a disaster can be impeded by the lack of computing and networking resources needed for an *ad hoc* datacenter. The same disaster that caused

missing or stranded people also can diminish or completely eliminate local computing resources. Cloud infrastructure located off-site can provide the needed resources to carry out the SAR operations. Applications using cloud resources can be preempted to support emergency systems such as SAR operations during national crises much as emergency vehicles preempt normal traffic and commandeer the use of traffic lights and roadways. The resources that the cloud provides, however, are not known *a priori*. Thus, the effective QoS for the SAR operations are dependent on the computing resources provided.

IV.1.2.2 Key Challenges in Supporting Search and Rescue Operations in Cloud Computing Environments

Meeting the requirements of SAR operations outlined in Section IV.1.2.1 is hard due to the inherent complexity of configuring enterprise DRE pub/sub middleware based on the computing resources the cloud provides. These resources are not known *a priori* and yet the QoS of the system is affected by the specific resources provided. The remainder of this section describes four challenges that ADAMANT addresses to support the communication requirements of the SAR operations presented above.

Challenge 1: Configuring for data timeliness and reliability. SAR operations must receive sufficient data reliability and timeliness so that multiple data streams can be fused appropriately. For instance, the SAR operation example described above shows how data streams (such as infrared scan and video streams) can be exploited by multiple applications simultaneously in a datacenter. The top half of Figure 42 shows how security monitoring and structural damage applications can use video stream data to detect looting and unsafe buildings, respectively. The bottom half of Figure 42 shows how fire detection applications and power grid assessment applications can use infrared scans to detect fires and working HVAC systems, respectively.

Likewise, the SAR systems must be configured to best use the computing and networking resources from the cloud to address data timeliness and reliability. These systems must

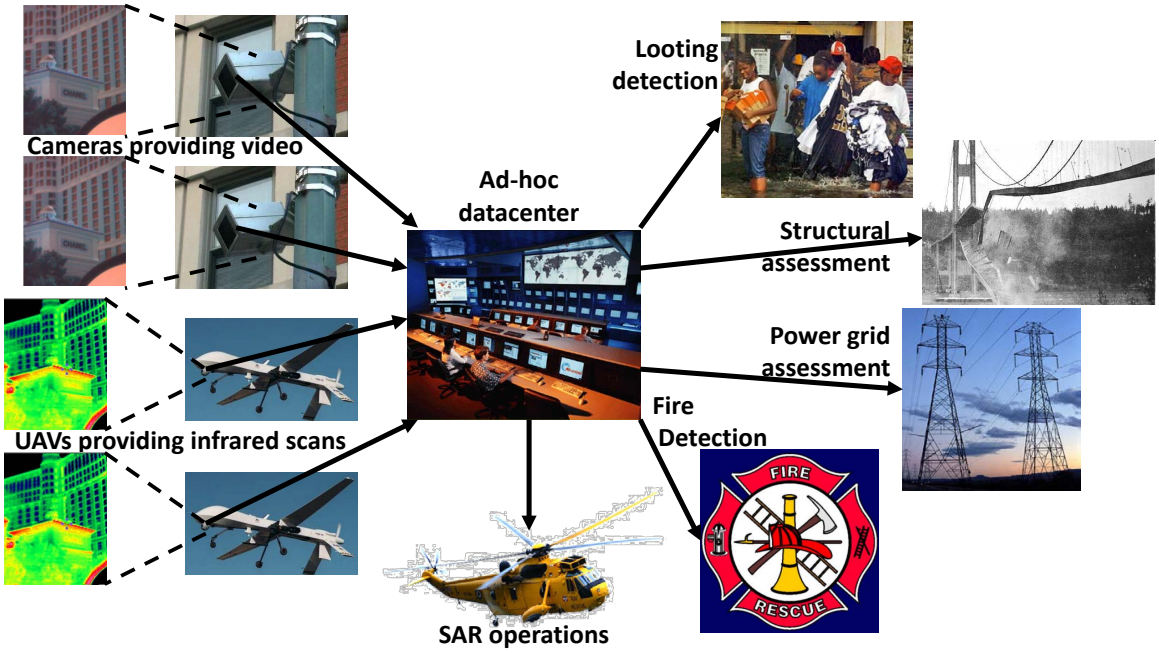

Figure 42: Uses of Infrared Scans & Video Streams during Disaster Recovery

therefore (1) use transport protocols that provide both reliability and timeliness and (2) know how these protocols behave in different computing and networking environments. Sections IV.1.5.1 and IV.1.3.1 describe how ADAMANT addresses this challenge by utilizing composite QoS metrics to measure both timeliness and reliability and incorporating transport protocols that configure the datacenter's pub/sub middleware to balance reliability and low latency.

Challenge 2: Timely configuration. Due to timeliness concerns of DRE systems such as SAR systems, the *ad hoc* datacenter used for SAR operations must be configured in a timely manner based on the computing and networking resources provided by the cloud. If the datacenter cannot be configured quickly, invaluable time will be lost leading to survivors not being saved and critical infrastructure (such as dams and power plants) not being safeguarded from further damage. During a regional or national emergency any wasted time can mean the difference between life and death for survivors and the salvaging or destruction of key regional utilities.

Moreover, applications and systems used during one disaster can be leveraged for other

disasters. Available computing and networking resources differ from one set of disaster recovery operations to another. Depending on the available cloud resources, therefore, the configuration times of *ad hoc* datacenters for SAR operations, for example, must be bounded and fast to ensure appropriate responsiveness. Determining appropriate configurations must also provide predictable response to ensure rapid and dependable response times across different computing and networking resources. Sections IV.1.4.4 and IV.1.5.4 describe how ADAMANT addresses this challenge by utilizing an artificial neural network machine learning tool to autonomically configure the datacenter's pub/sub middleware quickly and predictably.

Challenge 3: Accuracy of configurations. Since data timeliness and reliability is related to the computing resources available and the configuration of the datacenter supporting the SAR operations in a cloud as noted in Challenge 1, configuring the datacenter must be done in an accurate manner. If the datacenter is incorrectly configured then the timeliness and reliability of the data (*e.g.*, the UAV scans and camera video used to detect survivors) will not be optimal for the given computing resources. For critical operations during disasters, such as rescuing survivors, the supporting SAR system must utilize the available resources to their fullest extent. Sections IV.1.4.4 and IV.1.5.4 describe how ADAMANT addresses this challenge by using the artificial neural network machine learning tool to configure the datacenter's pub/sub middleware accurately.

Challenge 4: Reducing development complexity. Regional and local disasters occur in many places and at many different times. The functionality of applications used during one disaster may also be needed for other disasters. A system that is developed for one particular disaster in a particular operating environment, however, might not work well for a different disaster in a different operating environment. SAR operations could unexpectedly fail at a time when they are needed most due to differences in computing and networking resources available. Systems therefore must be developed and configured readily between the different operating environments presented by cloud computing to leverage the systems

across a wide range of disaster scenarios. Section IV.1.4.4 describes how ADAMANT addresses this challenge by using an artificial neural network machine learning tool to manage mapping the computing and network resources and application parameters (*e.g.*, data sending rate, number of data receivers) to the appropriate transport protocol to use.

IV.1.3 Overview of ADAptive Middleware And Network Transports (ADAMANT) for Timely Configuration

This section presents an overview of the *ADAptive Middleware And Network Transports* (ADAMANT) platform, which is QoS-enabled pub/sub middleware that integrates and enhances the *Adaptive Network Transports* (ANT) framework to support multiple transport protocols and the *Artificial Neural Network* (ANN) machine learning technology to select appropriate transport protocols in a timely and reliable manner. In the context of ADAMANT we empirically evaluate (1) the QoS delivered by DDS pub/sub middleware with respect to differences in computing and networking resources provided by cloud environments and (2) the accuracy and timeliness of ANN-based machine learning tools in determining appropriate middleware configurations.

Figure 43 shows how ADAMANT works in a cloud environment (*e.g.*, the *ad-hoc* SAR datacenter) to deploy cloud resources. Since ADAMANT configures itself based on the resources in a cloud, it must determine those resources autonomically when the cloud environment makes them available. ADAMANT queries the environment for hardware and networking resources using OS utilities.

For example, on Linux ADAMANT accesses the `/proc/cpuinfo` file to gather CPU information and executes the `ethtool` program to query network characteristics. ADAMANT combines this hardware information with other relevant application properties (*e.g.*, number of receivers and data sending rate) and sends it as input to the ANN, which determines the appropriate protocol in a timely manner and passes this information to ANT.

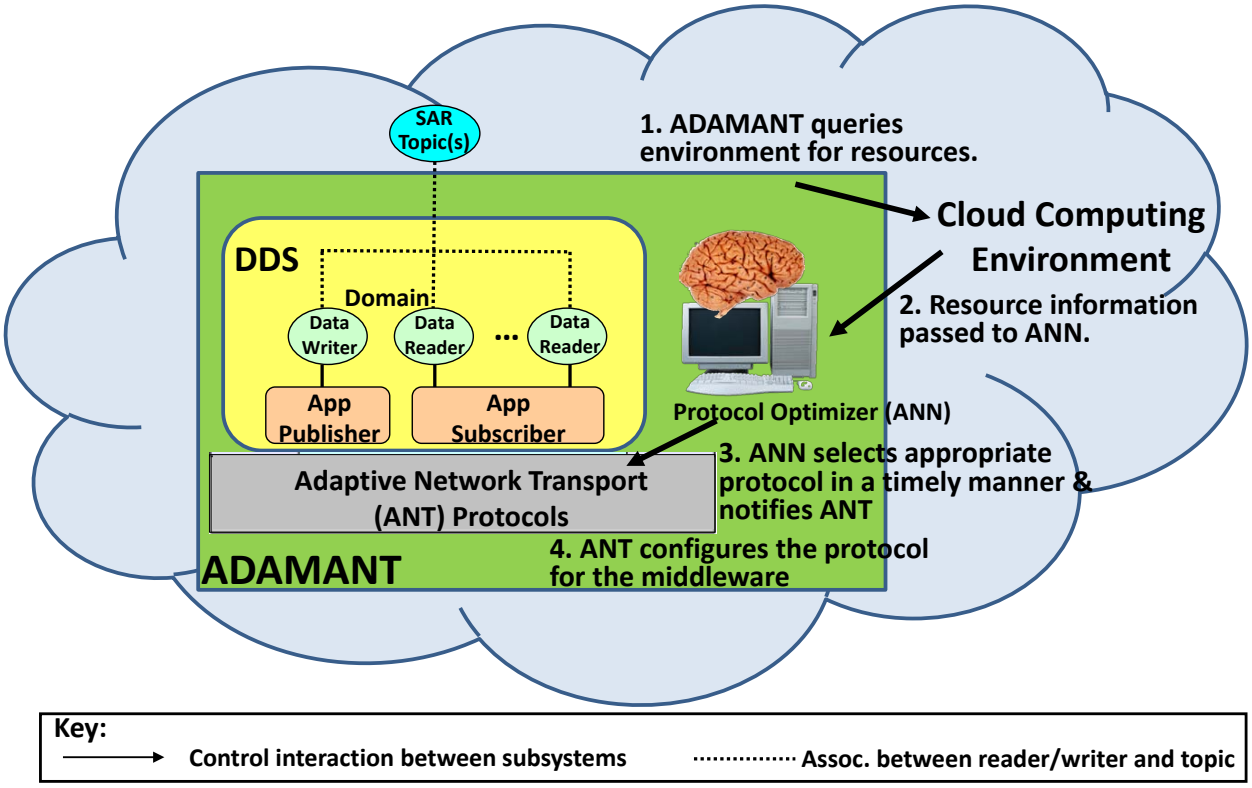

Figure 43: ADAMANT Architecture and Control Flow for Cloud Computing Environments

ANT then configures the DDS middleware to use the appropriate transport protocol. The remainder of this section describes the structure and functionality of ADAMANT.

IV.1.3.1 Adaptive Network Transports (ANT) Framework

The ANT 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 configured at startup to achieve greater flexibility and support configuration adaptation.

The ANT framework originally was derived from the Ricochet [9] transport protocol, which 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 properties. 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 protocol and ANT's NAKcast protocol, which is a NAK-based multicast protocol supporting a timeout parameter for when to send NAKs to the sender, with the evaluations done in this section. These protocols have been selected due to their support for balancing reliability and low latency [45].

The Ricochet protocol has two tunable parameters. The R parameter determines the number of packets a receiver 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. These two parameters affect the timeliness, reliability, and jitter of the data received as shown in Section IV.1.5.3. ANT helps address Challenge 1 in Section IV.1.2.2 by supporting transport protocols that balance reliability and low latency.

IV.1.4 Evaluating Adaptation Approaches for SAR Operations

Several approaches can be used to adapt transport protocols for QoS-enabled pub/-sub systems operating in flexible environments. Below we evaluate (1) policy-based approaches, (2) reinforcement learning, and (3) supervised machine learning with and without bounded search times. We focus on timely, bounded adaptation and leave other analysis aspects of the approaches (*e.g.*, memory requirements) as future work. Moreover, ADAMANT includes leveraging the QoS-enabled middleware to prioritize adjustments so that transition times are acceptable. We also present evaluation criteria relevant to developing and deploying an adaptation approach for SAR operations, including (1) boundedness in searching for a solution, (2) accuracy of solution for known environments, (3) robustness to unknown operating environments, and (4) accidental development complexity.

IV.1.4.1 Evaluating Policy-based Adaptation Approaches

Policy-based approaches provide a straightforward way to determine optimal transport protocols for a given operating environment. After certain operating conditions are checked and met, the system can be directed by the policies to alter its behavior. Figure 44 shows an example where the application checks for three environment aspects: (1) percentage loss in the network (*i.e., network_loss_percent*), (2) number of data receivers (*i.e., num_receivers*), and (3) the rate at which data is published (*i.e., sending_rate*).

Policy-based approaches can be optimized since the bounded number of (1) conditions that are checked and (2) the behaviors used to direct the system are explicitly identified. As shown in Figure 44, a `switch` statement or nested `if` statements in a programming language can be used to implement policy-based approaches. In general, policy-based approaches can provide boundedness in searching for an adaptation solution and therefore address the boundedness evaluation criterion for adaptation approaches (*e.g.*, switch statements can be optimized to constant time performance). Policy-based approaches also are highly accurate for known solutions since developers can codify the exact behavior needed

```
if (network_loss_percent == 1
    && num_receivers < 5
    && sending_rate < 0.01) {
  transport_framework->use (transport1);
} else if (network_loss_percent == 5
           && num_receivers < 5
           && sending_rate < 0.01) {
  transport_framework->use (transport2);
} else if (network_loss_percent == 10
           && num_receivers < 5
           && sending_rate < 0.01) {
  transport framework->use (transport3);
}
```

Figure 44: Policy-based Example

for a known environment, thereby addressing the evaluation criterion for accuracy in known environments.

Policy-based approaches, however, 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, so policy-based approaches do not address the robustness evaluation criterion for adaptation approaches.

Even when all relevant conditions are considered and all appropriate responses are codified, manually managing the conditions and responses for policy-based approaches increases accidental complexity. Figure 44 presents only three operating environment aspects that are checked. Since each aspect can take an infinite range of values there is an infinite number of combinations that can be checked. Even using ranges of values can lead to infinite number of combinations. Moreover, if the policies need to be modified the chance of introducing an error increases with the number of aspects considered along with the number of ranges of values for each aspect. Policy-based approaches therefore do not address the accidental development complexity criterion for adaptation approaches.

IV.1.4.2 Evaluating Reinforcement Learning

Reinforcement learning provides robustness and flexibility when not all conditions and appropriate system responses are known *a priori*. Reinforcement learning approaches leverage high-level abstract guidance for a proposed solution (*e.g.*, determining the solution to be good or bad). For example, reinforcement learning sets certain system behaviors as goals and uses positive and negative reinforcements to guide the resolution of system behavior as change in an operating environment occurs [95].

Reinforcement learning explores the possible solution space to determine generalized solutions of the negative and positive reinforcements given. Reinforcement learning is thus unbounded in its determination of an appropriate response due to online exploration of the solution space and modification of decisions while the system is running. As indicated by Bu [19], performance of reinforcement learning benefits from an additional run-time initialization period before system startup. Reinforcement learning therefore does not address the evaluation criterion for boundedness when searching for an adaptation solution.

Reinforcement learning generalizes knowledge gained from positive and negative reinforcements of multiple proposed solutions. With this generalization comes a loss of information for the specific solutions that have been tried. Reinforcement learning thus does not entirely address the criterion for accuracy in known environments.

In contrast, the generalization of knowledge for reinforcement learning does allow the approach to interpolate and extrapolate from solutions of known environments to unknown environments. Accordingly, reinforcement learning addresses the criterion for robustness to unknown operating environments.

Even when all conditions of the operating environment are known and all appropriate responses determined, reinforcement learning can manage the conditions and appropriate responses rather than forcing developers to address these areas programmatically. Knowing how to respond to various operating environments is resolved by the reinforcement

learning approach itself. Reinforcement learning thus addresses the criterion of accidental development complexity.

IV.1.4.3 Evaluating Supervised Machine Learning

Supervised machine learning techniques classify new examples while incorporating generalized knowledge from previous examples. These techniques are supervised by being provided solutions to the problem which they use to expand the generalized knowledge. Supervised machine learning techniques generally have an offline training period to build up knowledge and then are used online when a system is running. Below we evaluate two common supervised machine learning approaches of decision trees (DTs) and artificial neural networks (ANNs).

DTs build a tree structure that branches on decisions which lead down to a leaf node that can accurately classify a new example [74]. A DT generates decision branches that split the data. Decisions that split the data more evenly are placed closer to the root of the tree. In general, a DT can be unbounded in the levels of the tree that is generated.

The attributes that are used for classification to generate the tree can be combined in an exponential number of ways. These combinations are then used to determine branches in the tree. As shown in Figure 45, the attribute `network_bytes` is used multiple places in the tree to branch the tree. DTs thus do not address the boundedness criteria for adaptation.

Moreover, as with machine learning in general, the knowledge obtained is generalized to apply to a wide variety of operating environments. This generalization of knowledge implies that DTs do not perfectly address the accuracy criterion for known operating environments. They can, however, provide solutions to environments not seen previously and therefore address the robustness criterion. In addition, DTs automatically capture branching decisions to determine an appropriate transport protocol configuration, thereby addressing the criterion of accidental development complexity.

An ANN is a supervised machine learning technique that is modeled on the interaction

```
J48 pruned tree
------------------

network_bytes <= 25612604
|   percent_packet_loss <= 1
|   |   network_bytes <= 11024041
|   |   |   percent_packet_loss <= 0
|   |   |   |   network_bytes <= 3275210: NAKcast-0.05 (3.0)
|   |   |   |   network_bytes > 3275210: NAKcast-0.025 (11.0)
|   |   |   percent_packet_loss > 0
|   |   |   |   num_receivers <= 3
|   |   |   |   |   network_bytes <= 4260177: NAKcast-0.05 (6.0)
|   |   |   |   |   network_bytes > 4260177
|   |   |   |   |   |   duration <= 2127.917476: NAKcast-0.025 (2.0)
|   |   |   |   |   |   duration > 2127.917476: NAKcast-0.1 (2.0)
|   |   |   num_receivers > 3: NAKcast-0.05 (4.0)
|   |   network_bytes > 11024041: NAKcast-0.025 (17.0/1.0)
|   percent_packet_loss > 1: NAKcast-0.025 (37.0/3.0)
network_bytes > 25612604
|   network_bytes <= 25729972: Ricochet-R8C3 (2.0)
|   network_bytes > 25729972
|   |   num_receivers <= 20: Ricochet-R4C3 (62.0)
|   |   num_receivers > 20
|   |   |   std_dev <= 5439.952831: Ricochet-R4C3 (2.0)
|   |   |   std_dev > 5439.952831: Ricochet-R8C3 (2.0)
```

Figure 45: A Decision Tree For Determining Appropriate Protocol

of neurons in the human brain [85]. As shown in Figure 46, an ANN has an input layer for relevant aspects of the operating environment (*e.g.*, percent network loss, sending rate). An output layer represents the solution that is generated based on the input. Connecting the input and output layers is a hidden layer. As the ANN is trained on inputs and correspondingly correct outputs, it strengthens or weakens connections between the layers to generalize based on the inputs and outputs.

Figure 46 also shows how an ANN can be configured statically in the number of hidden layers and the number of nodes in each layer that directly affects the processing time complexity between the input of operating environment conditions and the output of an appropriate transport protocol and settings. This static configuration structure supports bounded response times. ANNs thus address the boundedness criterion for generating a solution.

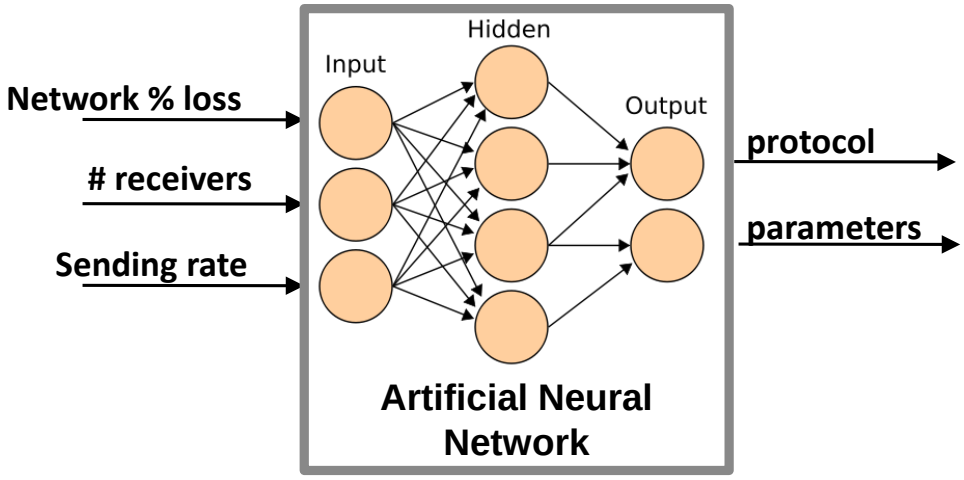

Figure 46: Artificial Neural Network For Determining Appropriate Transport Protocol

As ANNs generalize the knowledge from the supervised training, they provide interpolation and extrapolation of known training sets to handle conditions for which the techniques have not been trained. Although ANNs do not entirely address the accuracy criterion for known environments they do address the robustness criterion for unknown operating environments. Moreover, since ANNs automatically encapsulate the generalization of knowledge across supervised training data, they address the criterion of accidental development complexity since developers need not determine appropriate transport protocol configurations given operating environment features. Since ANNs address more of our objectives than the other approaches, we have chosen ANNs in the remainder of our work. We are also researching ways to increase the accuracy of ANNs for known environments.

Based on the evaluation of adaptation approaches in this section, we have determined ANNs are a promising adaptation approach for DRE systems like SAR operations. ANNs support (1) bounded time complexity for determining a solution, (2) robustness for unknown operating environments, and (3) reduction of accidental development complexity. Moreover, as shown in Section IV.2.3.1, ANNs can be configured and trained for known environments so that the accuracy is greatly increased and becomes comparable with policy-based approaches (*i.e.*, 100% accurate).

IV.1.4.4 Artificial Neural Network Tools to Determine Middleware Configurations

Several machine learning approaches can be used to configure middleware autonomically in a cloud computing environment. We selected ANN technology [48] due to its (1) fast and predictable performance, (2) accuracy for environments known *a priori* (*i.e.*, used for ANN training) *and* unknown until runtime (*i.e.*, not used for ANN training), and (3) low accidental development complexity. In particular, we chose the *Fast Artificial Neural Network* (FANN)(`leenissen.dk/fann`) implementation due to its configurability, documentation, ease of use, and open-source code. Section IV.1.5.4 shows the accuracy and timeliness of a neural network trained and tested using the data collected from the experiments described in Section IV.1.5.3. In particular, neural networks provide 100% accuracy for environments known *a priori*, high accuracy for environments unknown until runtime, and the low latency, constant time-complexity required for DRE systems such as SAR operations.

The use of an ANN helps address Challenges 2 and 3 in Section IV.1.2.2 by providing accurate, fast, and predictable guidance for determining an appropriate ADAMANT configuration for a given cloud computing environment. An ANN also helps address Challenge 4 in Section IV.1.2.2 by autonomically managing the mappings from the computing and network resources available and the application parameters (*e.g.*, data sending rate, number of data receivers) to the appropriate transport protocols. An ANN thus reduces the development complexity for configuring the pub/sub middleware appropriately as compared to manual adaptation approaches (*e.g.*, implementing switch statements), which are tedious and error-prone [57].

IV.1.4.5 DDS Implementations Used

Several DDS implementations exist on the market today. We chose OpenDDS and OpenSplice as ADAMANT's DDS implementations due to their (1) open source availability, which facilities modification and experimentation, and (2) support for a pluggable

transport framework that allows SAR application developers to create and integrate custom transport protocols for transmitting data. The pluggable protocol architectures provided by OpenDDS and OpenSplice differ in the following ways. OpenDDS's pluggable transport framework uses patterns (*e.g.*, Strategy [32] and Component Configurator [92]) to provide flexibility and delegate responsibility to the protocol only when applicable. Application developers inherit from key classes to override and define the behavior for a particular protocol. OpenSplice provides an efficient C API and an XML interface to describe the functions that support a transport protocol.

IV.1.5 Experimental Results and Analysis

The section presents the results of experiments we conducted to empirically evaluate (1) the effect of computing and networking resources on the QoS provided by ADAMANT as measured by the composite QoS metrics defined in Section IV.1.5.1 and (2) the timeliness and accuracy of an ANN in determining an appropriate ADAMANT configuration given a particular cloud computing environment. The experiments include ADAMANT with multiple aspects of the operating environment varied (*e.g.*, CPU speed, network bandwidth, DDS implementation, percent data loss in the network) along with multiple aspects of the application being varied as would be expected with SAR operations (*e.g.*, number of receivers, sending rate of the data).

IV.1.5.1 Composite QoS Metrics for Reliability and Timeliness

Our work on FLEXMAT as outlined in Chapter III indicated that some transport protocols provide better reliability (as measured by the number of network packets received divided by the number sent) and latency for certain environments while other protocols are better for other environments. We therefore developed several *composite QoS metrics* to evaluate multiple QoS aspects simultaneously, thereby providing a uniform and objective evaluation of ADAMANT in cloud computing environments.

Reliability of data in networked systems often implies discovering lost data via positive or negative acknowledgments and then retransmitting the lost data. This loss discovery and retransmission process takes time which increases the latency of the data. Composite metrics help to show how well the ADAMANT middleware addresses the multiple QoS concerns in a given operating environment via leveraging different transport protocols.

The ReLate2 metric, as described previously in Section III.5.1.3, has been expanded to a family of metrics that focus on data reliability and latency which are often in tension with each other in DRE systems. Our composite QoS metrics focus on reliability and average latency and include the QoS aspects of (1) *jitter* (*i.e.*, standard deviation of the latency of network packets), (2) *burstiness* (*i.e.*, the standard deviation of average bandwidth usage per second of time), and (3) network bandwidth usage.

We have extended the ReLate2 metric to include other QoS properties relevant to DRE systems. In particular, *jitter* (*i.e.*, standard deviation of the latency of network packets) is also an important QoS consideration for applications using multimedia data, such as SCAAL's personal surveillance video or 3-dimensional health monitoring information. For example, as outlined in Section III.5.1.3, late arriving MPEG data can be worse than not receiving the data at all. Jitter provides a way to measure the variance of data arrival times and is an important QoS consideration for data that depends on preceding or succeeding data.

Our new *ReLate2Jit* metric extends the original ReLate2 metric to include jitter. ReLate2Jit yields a numeric value that can be used to quantifiably compare the performance of transport protocols w.r.t. reliability, average latency, and jitter. ReLate2Jit values increase with an increase in jitter and low values are more desirable than high values. We calculate the standard deviation of the packet arrivals and multiply this value by the ReLate2 metric as follows:

$$ReLate2Jit_p = ReLate2_p \times \sigma_p$$

where p is the protocol being evaluated and

σ_p = standard deviation of packet latency times for protocol p.

We multiply the various QoS concerns (*e.g.* latency, reliability) to give fair weighting (*e.g.*, latency time units, percentage loss for reliability). We present experimental results using the ReLate2Jit metric. Our experimental environment is similar to the one used for our ReLate2 results [45], except that we use the OpenSplice DDS rather than OpenDDS. We send data from a publisher to subscribers while varying the sending rate, the percent loss in the network, and the number of subscribers.

We focus only on the transport protocols from previous work that balanced reliability and latency, that is, (1) the NAKcast protocol, which is a NAK-based multicast protocol, with a retransmission timeout set to 0.05 and 0.025 seconds, and (2) the Ricochet protocol, which is a lateral error correction protocol, with the R parameter set to 4 and 8 and the C parameter set to 3. Ricochet's R value determines how many packets are received before an error correction packet is sent out to the other receivers. Ricochet's C value determines how many other receivers are sent the error correction packet.

Figure 47 shows results of using the ReLate2Jit metric for an operating environment with 3 receivers, 1% network loss, and a sending rate of 100Hz. The results show that the Ricochet protocol performs well when considering reliability, average latency, and jitter compared against NAKcast. The ReLate2Jit values between Ricochet and NAKcast are not as profound for lower sending rates although Ricochet generally outperforms NAKcast. As sending rates increase Ricochet's jitter decreases proportionately accounting for greater disparity between Ricochet and NAKcast at higher rates.

Being able to predict and provision for adequate resources is an important aspect of DRE systems. If allocated resources are inadequate then DRE systems running in resource constrained environments will not perform as intended, so that related QoS (such as reliability and latency) will not be met. Network bandwidth is an important resource consideration in DRE systems since it must be provisioned and managed appropriately. Moreover, as

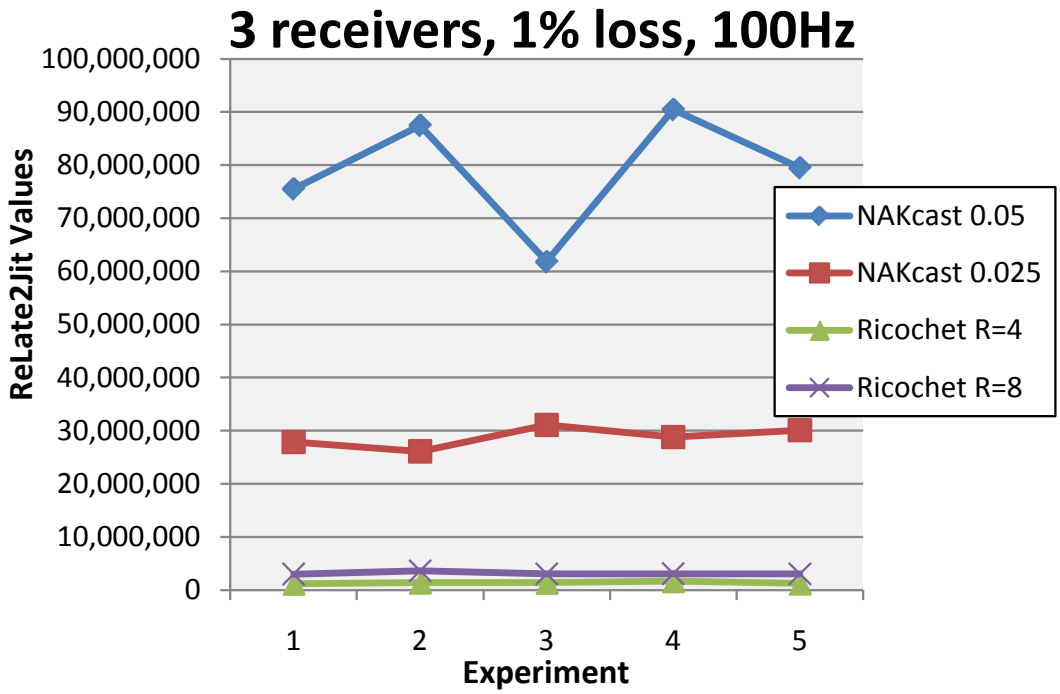

Figure 47: ReLate2Jit for 3 Receivers, 1% Network Loss, and 100Hz Sending Rate

ultra-large-scale systems [53] become more prevalent, changes in network resources (*e.g.*, bandwidth) will become more dynamic and require more online adjustments.

To evaluate the concerns of reliability, latency, and network bandwidth we also have developed the *ReLate2Net* and *ReLate2Burst* composite metrics. ReLate2Net multiples the ReLate2 metric by the average network bandwidth usage per second to determine how a transport protocol balances reliability, latency, and network bandwidth. ReLate2Burst multiples the ReLate2 metric by the network bandwidth's *burstiness* (*i.e.*, the standard deviation of average bandwidth usage per second of time) to determine how a transport protocol balances reliability, latency, and packet burstiness.

The inclusion of network bandwidth information with the ReLate2 metric provides guidance for DRE systems in evaluating transport protocols and appropriately provisioning constrained DRE systems to function as needed in dynamic environments. Moreover, for ULS systems that must manage fluctuating network bandwidth capacity, the ReLate2Net and ReLate2Burst metrics can be used to select an appropriate transport protocol.

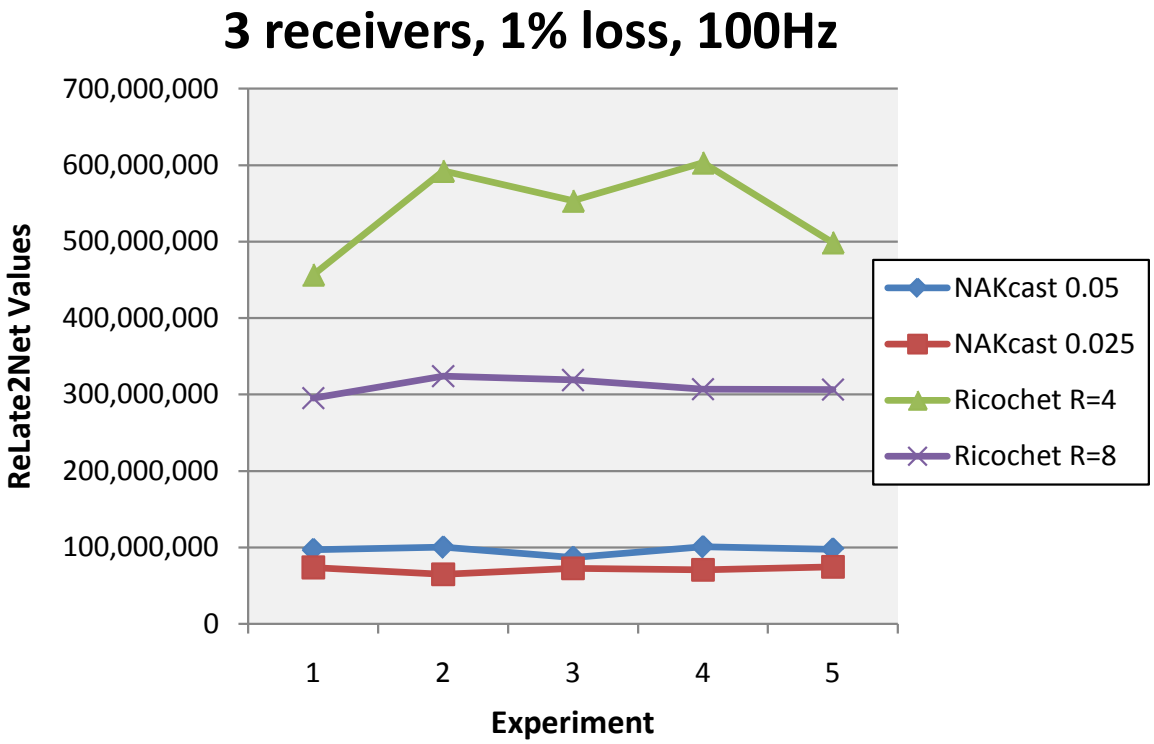

Figure 48: ReLate2Net for 3 receivers, 1% network loss, and 100Hz sending rate

Figures 48 and 49 show results of using the ReLate2Net and ReLate2Burst metrics respectively for an operating environment with 3 receivers, 1% network loss, and a sending rate of 100Hz. The ReLate2Net results highlight that the Ricochet protocol uses considerably more network bandwidth on average than NAKcast which is to be expected. Note that NAKcast with a shorter retransmission timeout (*i.e.*, 0.025 seconds) has a consistently lower ReLate2Net value than does NAKcast with a longer retransmission timeout (*i.e.*, 0.05 seconds). While *NAKcast 0.025* does use more network bandwidth on average than *NAKcast 0.05*, it also has a lower average latency since lost messages are requested sooner. The ReLate2Burst results mimic the ReLate2Net results for this environment configuration.

We apply our family of composite metrics to QoS-enabled DDS pub/sub middleware using various transport protocols supported by ANT to train the ANN. The ANN is trained with an understanding of how integration of middleware with each protocol affects the

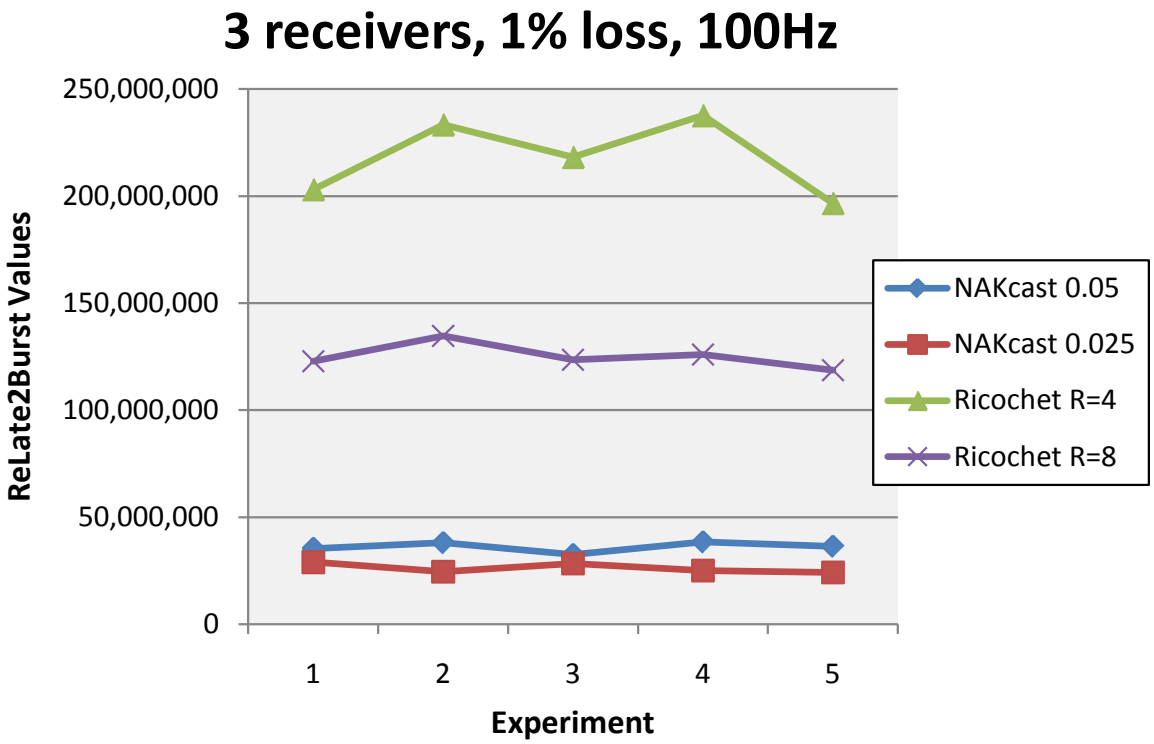

Figure 49: ReLate2Burst for 3 receivers, 1% network loss, and 100Hz sending rate

QoS properties of reliability and latency given the variability of computing and networking resources of a cloud environment.

IV.1.5.2 Experimental Setup

We conducted our experiments using the Emulab network testbed, which provides on-demand computing platforms and network resources that can be easily configured with the desired OS, network topology, and network traffic shaping. We used Emulab due to its (1) support for multiple types of computing platforms, (2) numbers of computing platforms, and (3) support for multiple network bandwidths. The flexibility of Emulab presents a representative testbed to train and test ADAMANT's configurability support for cloud computing environments.

As described in Section IV.1.2, we are concerned with the distribution of data for SAR

Table 12: Environment Variables

Point of Variability	Values
Machine type	pc850, pc3000
Network bandwidth	1Gb, 100Mb, 10Mb
DDS Implementation	OpenDDS, OpenSplice
Percent end-host network loss	1 to 5 %

Table 13: Application Variables

Point of Variability	Values
Number of receiving data readers	3 - 15
Frequency of sending data	10 Hz, 25 Hz, 50 Hz, 100 Hz

datacenters, where network packets are typically dropped at end hosts [10]. The ADAMANT software for the receiving data readers supports programmatically dropping random data packets. We modified ADAMANT to drop packets based on the loss percentage specified for the experiment.

Our experiments were configured with the following traffic generation models using version 1.2.1 of OpenDDS and version 3.4.2 of OpenSplice. One DDS data writer sent out data, a variable number of DDS data readers received the data. The data writer and each data reader ran on its own computing platform and 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.*, 3 receiving data writers, 50 Hz sending rate, 2% end host packet loss, pc3000 computing platform, and 1Gb network bandwidth).

We configured ADAMANT with Ricochet and NAKcast to determine how well it performs using these protocols. We modified NAKcast's timeout value as well as Ricochet's *R* and *C* parameters as described in Section IV.1.3.1. Table 12 outlines the points of variability provided by the cloud computing environment.

We include the DDS implementation in this table since some cloud computing environments provide hardware and software resources. We include network loss in the table

since the network characteristics in cloud computing can be specified in an end-user license agreement, which identifies the services that the cloud computing environment will provide and that consumers accept. The middleware for the SAR operations can then be configured appropriately using this information.

Table 13 outlines the points of variability due to the SAR operations. In particular, we varied the number of data receivers since only a few SAR applications might be interested in one data stream (*e.g.*, for a localized area with fine-grained searching) while many applications might be interested in a different data stream (*e.g.*, for a broader area with coarse-grained searching). Likewise, the sending rate might be high for SAR operations that need high-resolution imaging for detailed searching while a lower sending rate is sufficient for SAR operations where lower resolution imaging is sufficient for more generalized searching.

For computing resources we used Emulab's pc850 and pc3000 hardware platforms. The pc850 platform includes an 850 MHz 32-bit Pentium III processor with 256 MB of RAM. The pc3000 platform includes a 3 GHz 64-bit Xeon processor with 2 GB of RAM. We used the Fedora Core 6 operating system with real-time extensions on these hardware platforms to collect high resolution timings. The nodes were all configured in a LAN configuration indicative of a datacenter.

IV.1.5.3 Evaluating How Cloud Computing Resources Affect QoS

Below we analyze the results from experiments involving different cloud computing environments. We show experimental data where the selection of ADAMANT's transport protocol to support QoS differs based on the cloud computing environment. Information in this section addresses Challenge 1 in Section IV.1.2.2 by characterizing the performance of the transport protocols for various cloud computing environments.

Figures 50 and 51 show the results of experiments where we held constant the number of receivers (3), the percent loss (5%), and the DDS middleware (OpenSplice). We varied

the computing platform and the network bandwidth using the pc850 and pc3000 platforms, and 100Mb and 1Gb LANs, respectively. We ran the experiments using NAKcast with a NAK timeout setting of 50ms, 25ms, 10ms, and 1ms, and Ricochet with R=4, C=3 and R=8, C=3. We only include NAKcast with a timeout of 1ms and Ricochet R=4 C=3 since these were the only protocols that produced the best (*i.e.*, lowest) ReLate2 values for these operating environments. Likewise, we ran the ADAMANT experiments with sending rates of 10Hz, 25Hz, 50Hz, and 100Hz but only show results for 10Hz and 25Hz since these highlight different protocols that produce the lowest ReLate2 value.

Figure 50 shows two cases where the Ricochet protocol with R = 4 and C = 3 produces the best (*i.e.*, lowest) ReLate2 values for sending rates of both 10Hz and 25Hz when using the pc3000 computing platform and the 1Gb network. Conversely, Figure 51 shows how the NAKcast protocol with a NAK timeout set to 1 ms produces the best (*i.e.*, lowest) ReLate2 values for the same sending rates of 10Hz and 25Hz when using the pc850 computing platform and the 100Mb network. These figures show that by changing only the CPU speed, amount of RAM, and network bandwidth, different protocols produce a better ReLate2 value and therefore better support the QoS properties of reliability and average latency. The SAR datacenter pub/sub middleware should therefore be configured differently depending on the computing and networking resources that a cloud computing environment provides. No single protocol performs best in all cases based on the computing and networking resources.

We decompose the ReLate2 metric into its constituent parts of reliability and average packet latency to gain a better understanding of how changes in hardware can affect the QoS properties relevant to the ReLate2 metric. Figures 52 and 53 show the reliability of the NAKcast 0.001 and Ricochet R4 C3 protocols. The reliability of the protocols is relatively unaffected by differences in hardware and network resources as would be expected. The percent network loss is held constant for these experiments and the differences in hardware are not expected to affect how many packets are delivered reliably.

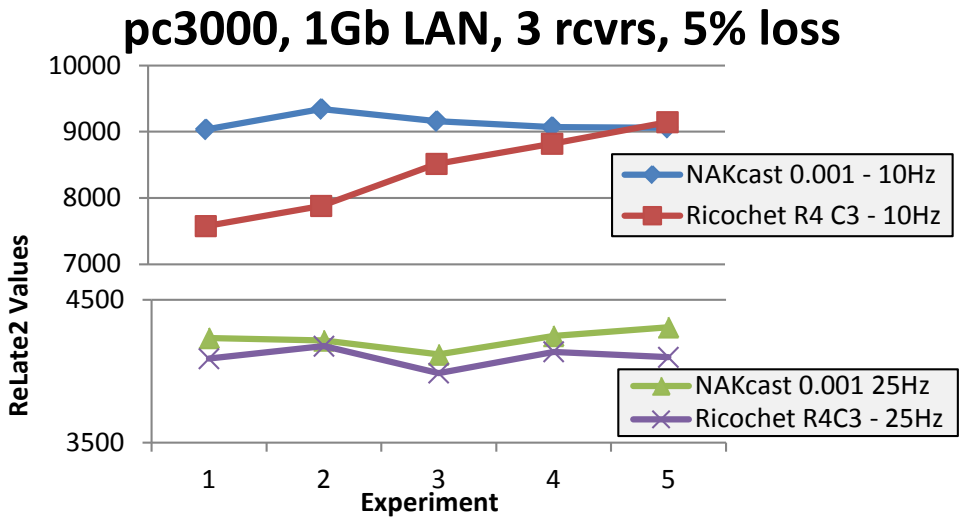

Figure 50: ReLate2: pc3000, 1Gb LAN, 3 receivers, 5% loss, 10 & 25Hz

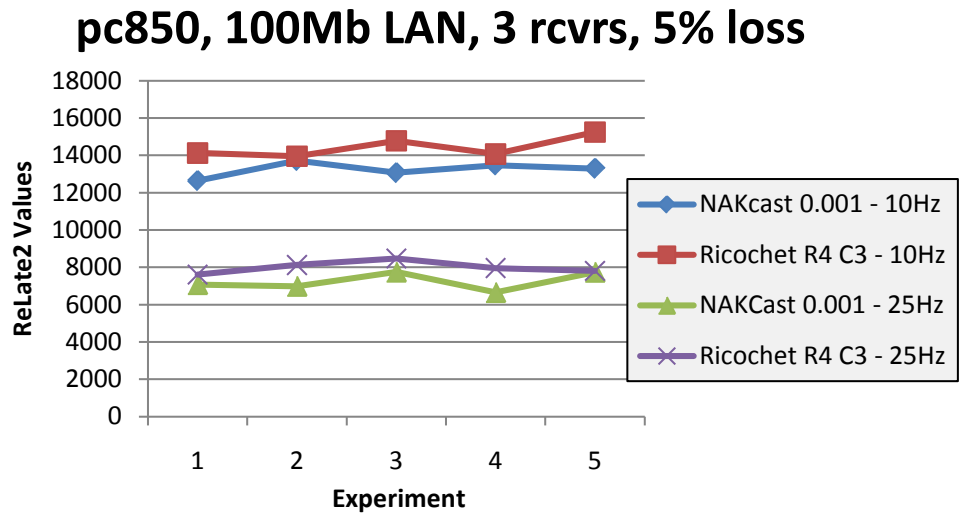

Figure 51: ReLate2: pc850, 100Mb LAN, 3 receivers, 5% loss, 10 & 25Hz

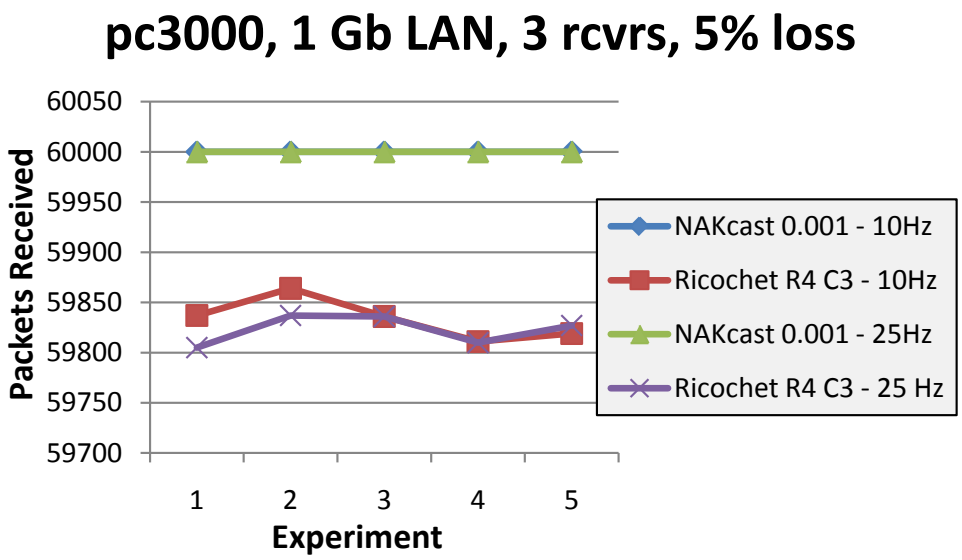

Figure 52: Reliability: pc3000, 1Gb LAN, 3 receivers, 5% loss, 10 & 25Hz

Figures 54 and 55 show that differences in computing speed and networking bandwidth have an effect on the average latency of packet arrival. In particular, there is a wider gap in the average latency times between the NAKcast and the Ricochet protocol when faster computing and networking resources are used. The faster the computing hardware and networking resources are, the faster the data packets on average should be received by the data receiver.

Since protocol reliability in these experiments is virtually constant, the difference in NAKcast performing better in one environment and Ricochet performing better in another stems from differences in average latency. With faster hardware and networks, Ricochet's average latency can overcome its lower reliability to perform better when reliability and average latency are both considered. Note that the graphs for the individual QoS property of average latency consistently show Ricochet performing better, while the graphs consistently show NAKcast performing better for reliability. Only when the QoS properties are combined in the ReLate2 metric is there a distinction between the appropriate protocol based on the hardware resources.

Figures 56 and 57 show that the differences in hardware resources affect the protocol to

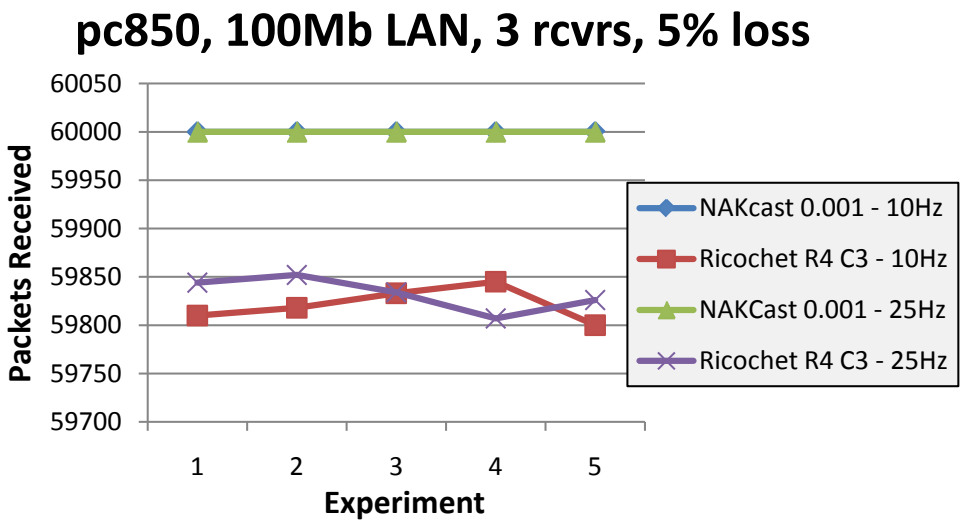

Figure 53: Reliability: pc850, 100Mb LAN, 3 receivers, 5% loss, 10 & 25Hz

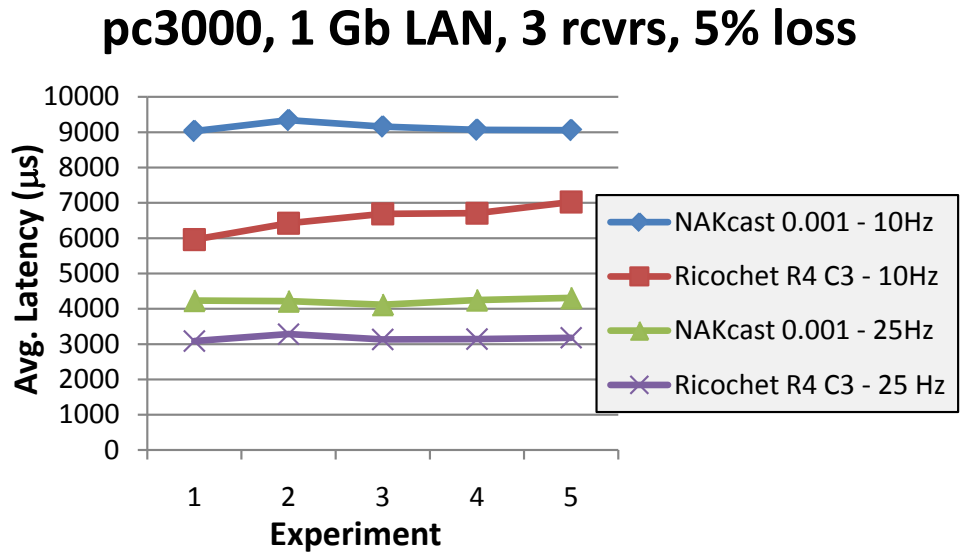

Figure 54: Latency: pc3000, 1Gb LAN, 3 receivers, 5% loss, 10 & 25Hz

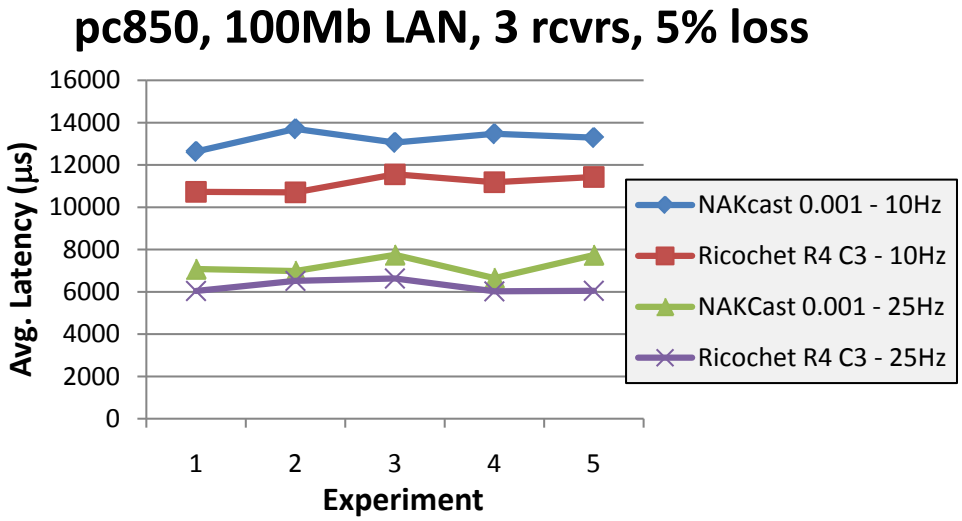

Figure 55: Latency: pc850, 100Mb LAN, 3 receivers, 5% loss, 10 & 25Hz

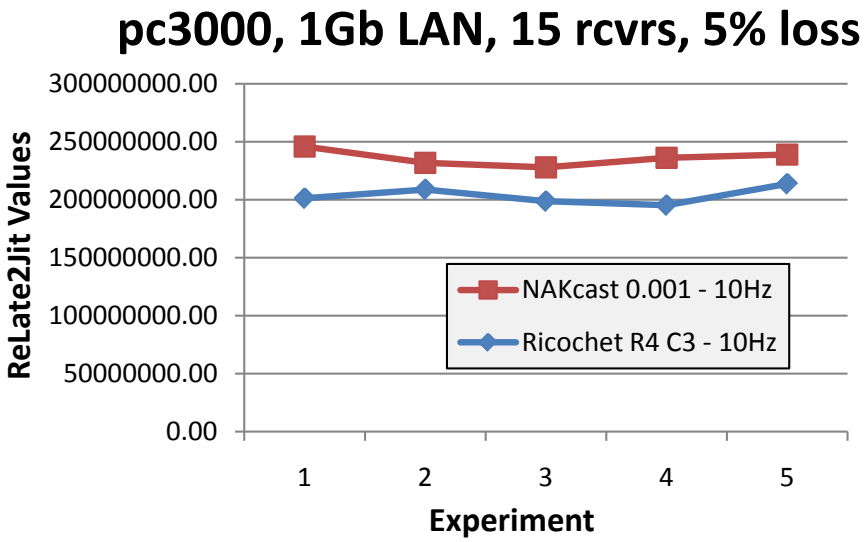

Figure 56: ReLate2Jit: pc3000, 1Gb LAN, 15 receivers, 5% loss, 10Hz

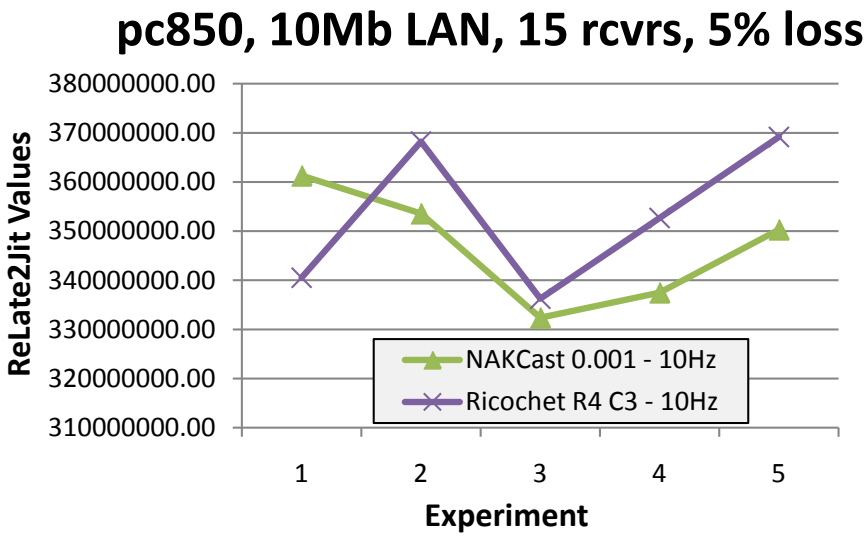

Figure 57: ReLate2Jit: pc850, 100Mb LAN, 15 receivers, 5% loss, 10Hz

choose based on the ReLate2Jit metric which measures reliability, average packet latency, and the standard deviation of packet latency (*i.e.*, jitter). The number of receivers is 15, the network percent loss is 5%, and the DDS middleware is OpenSplice. We again varied the computing platform and the network bandwidth using the pc850 and pc3000 platforms and 100Mb and 1Gb LANs, respectively. The figures only include data for NAKcast with a 1 ms timeout and Ricochet R=4 C=3 both with a 10Hz sending rate since, with this rate, the environment has triggered the selection of different protocols based on the ReLate2Jit values.

Figure 56 shows Ricochet R=4 C=3 to consistently have the best (*i.e.*, lowest) ReLate2Jit values when using pc3000 computers and a 1Gb network. Figure 57 shows NAKcast with a timeout of 1 ms as most of the time (4 out of 5 experiment runs) having the better ReLate2Jit value. We decompose the ReLate2Jit values to have a better understanding of the differences.

Figures 58 and 59 show the average latency broken out from the ReLate2Jit values above. These figures show that Ricochet R=4 C=3 consistently has the lowest average latencies regardless of the computing and network resources. Likewise, Figures 60 and 61

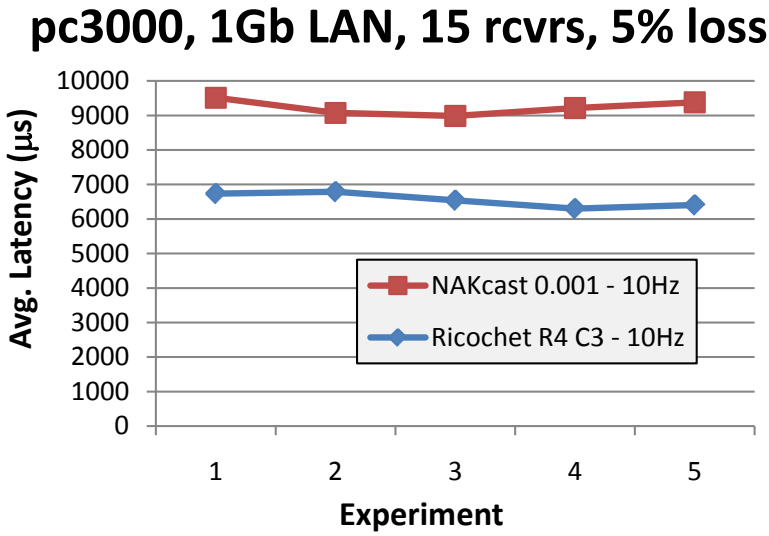

Figure 58: Latency: pc3000, 1Gb LAN, 15 receivers, 5% loss, 10Hz

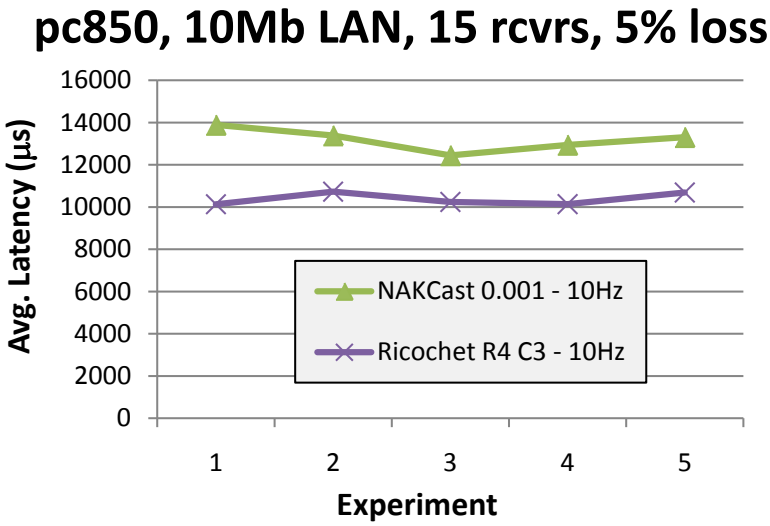

Figure 59: Latency: pc850, 100Mb LAN, 15 receivers, 5% loss, 10Hz

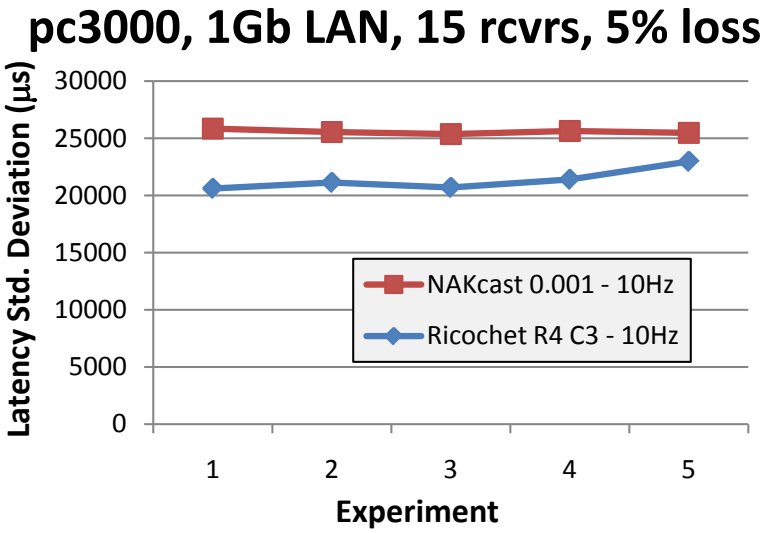

Figure 60: Jitter: pc3000, 1Gb LAN, 15 receivers, 5% loss, 10Hz

show that Ricochet R=4 C=3 consistently has lower jitter values across the different hardware. Figures 62 and 63 again show that NAKcast provides high reliability, while Ricochet provides less reliability.

All figures for individual QoS properties (*i.e.*, Figures 58 through 63) related to the ReLate2Jit measurements in Figures 56 and 57 show fairly consistent results across differing hardware. When these QoS properties are combined into a single, objective value, however, we are better able to distinguish one protocol from another thus highlighting the advantages to using composite metrics.

IV.1.5.4 Determining Appropriate Protocol with Artificial Neural Networks

As described in Section IV.1.4.4, ADAMANT uses an Artificial Neural Network (ANN) trained on experiment data (as shown in Section IV.1.5.3) to provide guidance in determining an appropriate transport protocol given (1) hardware resources provided by the cloud computing environment (*e.g.*, CPU speed, network bandwidth and loss) and (2) application properties (*e.g.*, sending rate, number of receivers). This section describes our accuracy and timeliness results using the *Fast Artificial Neural Network* (FANN) library.

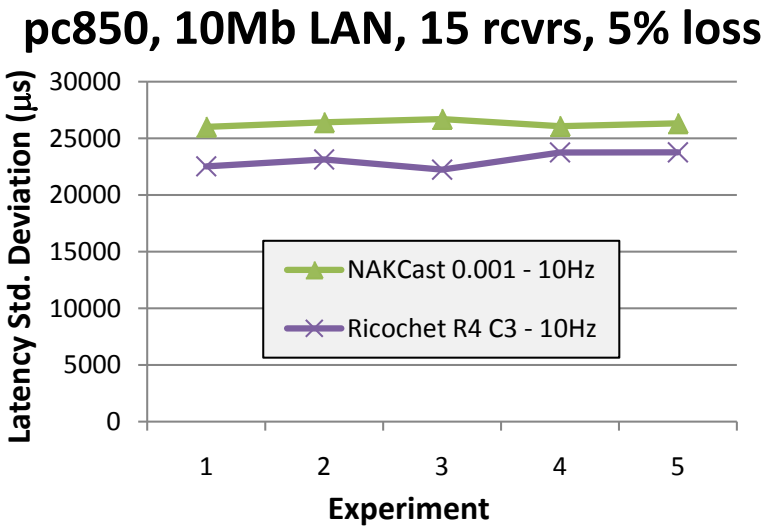

Figure 61: Jitter: pc850, 100Mb LAN, 15 receivers, 5% loss, 10Hz

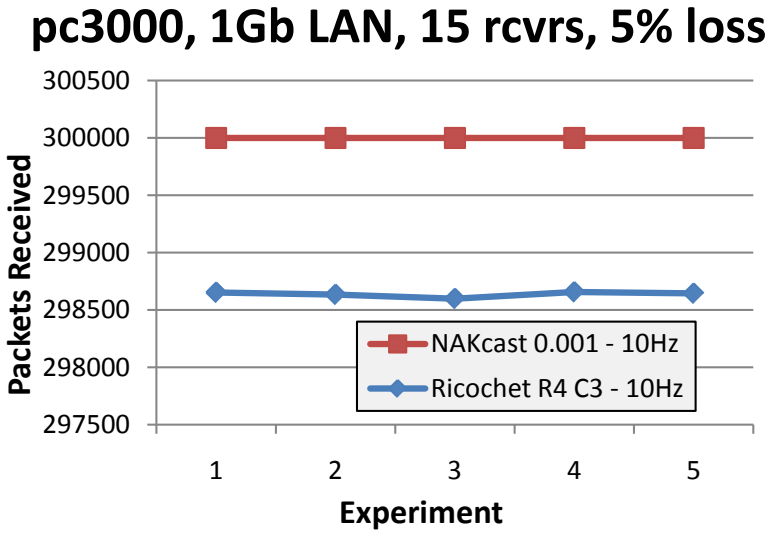

Figure 62: Reliability: pc3000, 1Gb LAN, 15 receivers, 5% loss, 10Hz

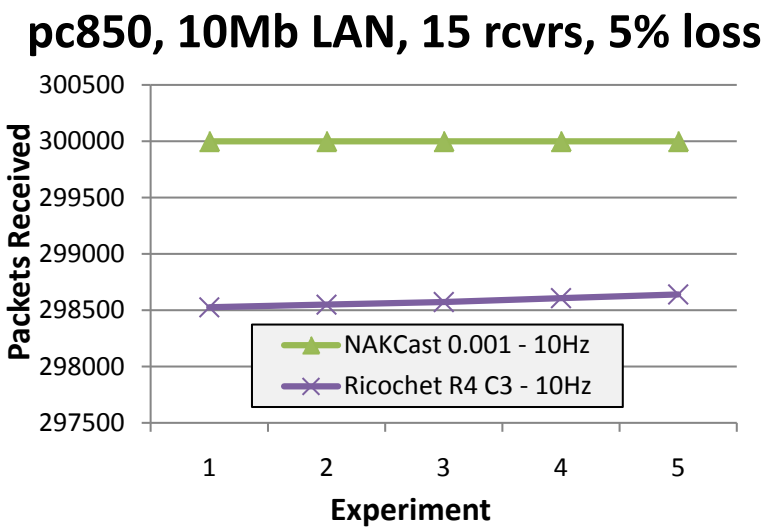

Figure 63: Reliability: pc850, 100Mb LAN, 15 receivers, 5% loss, 10Hz

Evaluating the Accuracy of Artificial Neural Networks

The first step to using an ANN is to train it on a set of data. We provided the ANN with 394 inputs where an input consists of data values outlined in Tables 12 and 13 plus the composite metric of interest (*i.e.*, ReLate2 or ReLate2Jit). We also provided the expected output (*i.e.*, the transport protocol that provided the best composite QoS value for ReLate2 or ReLate2Jit).

An example of one of the 394 inputs is the following: 3 data receivers, 1% network loss, 25Hz sending rate, pc3000 computers, 1Gb network, OpenSplice DDS implementation, and ReLate2Jit as the metric of interest. Based on our experiments, the corresponding output would be the NAKcast protocol with a NAK timeout of 1 ms. All the 394 inputs are taken from experiments that we ran as outlined in Section III.5.2.

FANN offers extensive configurability for the neural network, including the number of *hidden nodes* that connect inputs with outputs. We ran training experiments with the ANN using different numbers of hidden nodes to determine the most accurate ANN. For a given number of hidden nodes we trained the ANN five times. The weights of the ANN determine how strong connections are between nodes. The weights are initialized randomly and the initial values effect how well the ANN learns.

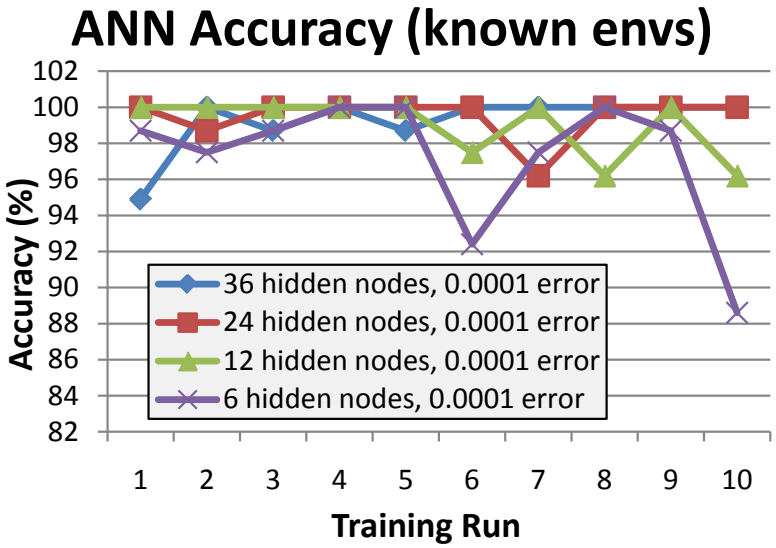

Figure 64: ANN Accuracy for environments known *a priori*

Figures 64 and 65 show the ANN accuracies for environment configurations that were known *a priori* and environments that were unknown until runtime respectively.

The ANN was configured with different numbers of hidden nodes and a stopping error of 0.0001 (*i.e.*, an indication to the ANN that it should keep iterating over the data until the error between what the ANN generates and the correct response is 0.0001). Additional experiments were conducted with higher stopping errors (*e.g.*, 0.01), but lower stopping errors consistently produced more accurate classifications as expected.

Accuracy for environments known *a priori* was determined by querying the ANN with the data on which it was trained. Since we know the answer we gave to the ANN when it was trained we check to make sure the answer matches the ANN's response. Over the 10 training runs shown in Figure 64 the highest number of 100% accurate classifications was generated using 24 hidden nodes (*i.e.*, 8).

Accuracy for environments unknown until runtime is determined by splitting out the 394 environment configurations into mutually exclusive training and testing data sets. This

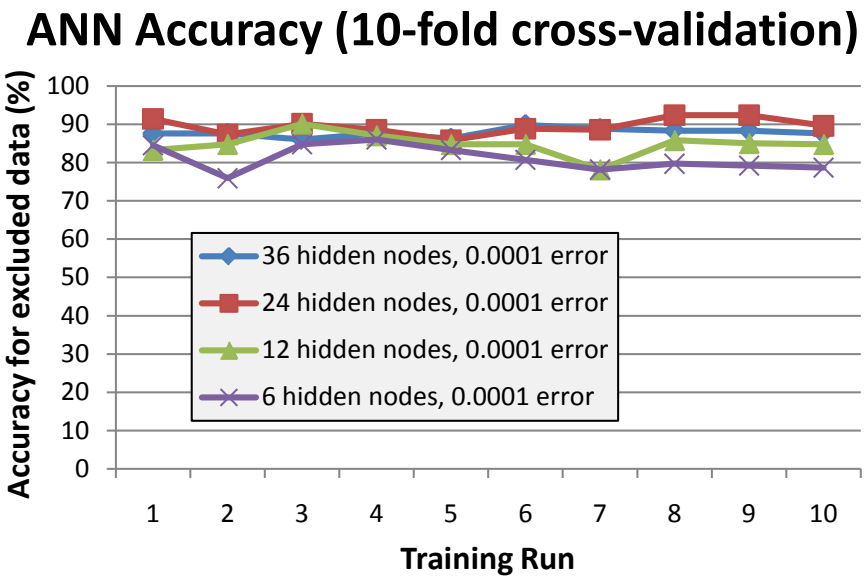

Figure 65: ANN Accuracy for environments unknown until runtime

approach is referred to as *n*-fold cross-validation where *n* is the number of mutually exclusive training and testing data sets [68]. The value of *n* also determines the amount of data excluded from training and used only for testing.

We used 10-fold cross-validation which indicates 10 sets of training and testing data where for each fold the training and testing data are mutually exclusive and the training data excludes 1/10 of the total data which is used only for testing. *N*-fold cross-validation provides insight into how well a machine learning technique generalizes for data on which it has not been trained. As shown in Figure 65 the ANN with 24 hidden nodes and a stopping error of 0.0001 produced the highest average accuracy of 89.49%. We conducted our timings tests using this ANN since it provided the highest number of 100% accurate classifications for environments known *a priori* and the highest accuracy for environments unknown until runtime.

Evaluating the Timeliness of Artificial Neural Networks

As described in Challenge 2 in Section IV.1.2.2, the datacenter for the SAR operations needs to have timely configuration adjustments. We now provide timing information based on the ANN's responsiveness when queried for an optimal transport protocol. Timeliness

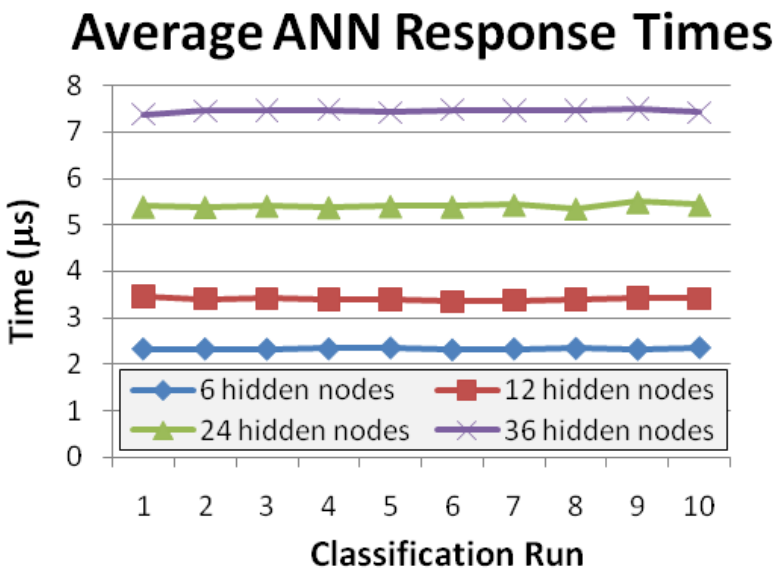

Figure 66: ANN average response times

was determined by querying the ANN with all 394 inputs on which it was trained. A high resolution timestamp was taken right before and right after each call made to the ANN.

Figures 66 and 67 show the average response times and standard deviation of the response times, respectively, for 5 separate experiments where for each experiment we query the ANN for each of the 394 inputs. The figures show that the ANN provides timely and consistent responses. As expected, the response times on the pc850 platform are slower than for the pc3000.

Inspection of the ANN source code confirmed experimental results that the ANN provides fast and predictable responses for both environments known *a priori* and unknown until runtime. When queried for a response with a given set of input values, the ANN loops through all connections between input nodes, hidden nodes, and output nodes. The number of nodes and number of connections between them were determined previously when the ANN was trained. With a high level of accuracy, predictability, and minimal development complexity, ANNs provide a suitable technique for determining ADAMANT configurations.

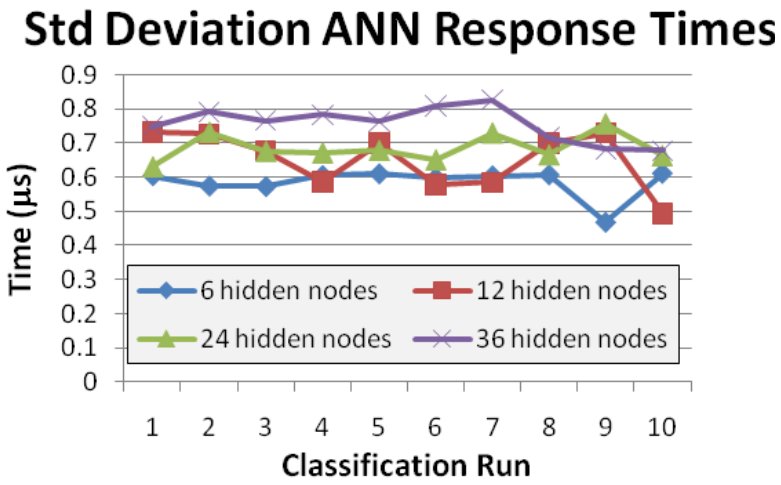

Figure 67: Standard deviation for ANN response times

IV.1.6 Lessons Learned

Developers of systems which use DRE pub/sub middleware face several configuration challenges for cloud computing environments. To address these challenges, this section presented the structure, functionality, and performance of *ADAptive Middleware And Network Transports* (ADAMANT). ADAMANT is pub/sub middleware that uses supervised machine learning to autonomously configure cloud environments with transport protocols that enhance the predictability of enterprise DRE systems.

The results in this section empirically showed how computing hardware environments affect QoS for these systems and how ADAMANT configures the system based on the computing resources provided at startup in a fast and accurate manner while reducing development complexity over manual adaptation approaches. We selected ANNs to determine appropriate configurations since they provide (1) the highest level of accuracy possible for known environments, (2) better than random or default guidance for environments not known until runtime, and (3) the timing complexity required for DRE systems. The following is a summary of lessons learned from our experience evaluating ADAMANT's configuration performance in various cloud environments:

- **Computing resources affect which QoS mechanism provides the best support.**

Differences in CPU speed and network bandwidth affect the choice of the most appropriate QoS mechanism. For certain computing environments, one transport protocol provided the best QoS; for other environments a different transport protocol was best. We leveraged this information to select the appropriate protocol for given computing resources. We are investigating other machine learning techniques that provide timeliness and high accuracy to compare with ANNs.

• **Quantifying the effect of hardware configurations on QoS is tedious and time-consuming.** While leveraging artificial neural networks with DRE pub/sub middleware provides fast, predictable configuration, neural networks need to be trained on the operating environment and the best protocol for that hardware environment. Performing empirical experiments and formatting the data take time—steps that are required regardless of the adaptation approach taken. While scripts and programs can be written to alleviate error-prone aspects of data gathering and transformation and thereby reduce accidental development complexity, running the experiments takes the greatest amount of time for preparing the neural network for training. We are researching integration of more generalized supervised machine learning techniques to provide guidance for previously unknown environments. This integration would complement neural networks so that the neural networks would provide guidance for known environments and the more generalized approach would provide even more accurate guidance for environments unknown until runtime.

• **Fast, predictable configuration for DRE pub/sub systems can support dynamic autonomic adaptation.** ADAMANT can accurately and quickly configure a DRE pub/sub system at startup in cloud environments. Some systems, however, run in operating environments that change during system operation. The ADAMANT results have motivated future work on autonomic adaptation of middleware and transport protocols to support QoS in turbulent environments. Fast, predictable configuration can be used to adapt transport protocols to support QoS while the system is monitoring the environment. When the system

detects environmental changes (*e.g.*, increase in number of receivers or increase in sending rate), supervised machine learning can provide guidance to support QoS for the new configuration.

• **Composite QoS metrics should be decomposed to better understand behavior of the system.** A change in the values from composite QoS metrics can be caused by changes in any of the individual QoS concerns or any combination of the concerns. The composite QoS metrics provide a higher level of abstraction for evaluating QoS and, as with any abstraction, details which might be important can be obfuscated. The composite QoS metrics we use are fairly easy to decompose as shown by Figures 50–55 in Section IV.1.5.3, although the more QoS properties that are composed the more decomposition is needed, which is hard, tedious, and time-consuming. However, if data for the individual QoS concerns are not available (*e.g.*, average latency, reliability) due to interpolation or extrapolation of data, it can be hard to determine why values from QoS metrics have changed.

IV.2 Autonomic Adaptation in Dynamic Environments

This section details the context, challenges, our solution approach, and results for autonomically configuring QoS-enabled pub/sub middleware for flexible computing environments.

IV.2.1 Context

Emerging trends and challenges. The use of pub/sub technologies for DRE systems has grown in recent years due to the advantages of performance, cost, and scale as compared to single computers [49, 98]. In particular, pub/sub middleware has been leveraged to ease the complexities of data dissemination for DRE systems. Examples of pub/sub middleware include the CORBA Notification Service [88], the Java Message Service (JMS) [75], Web Services Brokered Notification [79], and the Data Distribution Service (DDS) [84].

These technologies support the propagation of data and events throughout a system using an anonymous publication and subscription model that decouples event suppliers and consumers.

Pub/sub middleware is used across a wide variety of application domains, ranging from ship-board computing environments to cloud computing to stock trading. Moreover, the middleware provides policies that affect the end-to-end QoS of applications running in DRE systems. Policies that are common across various middleware technologies include grouped data transfer (*i.e.*, transmitting a group of data atomically), durability (*i.e.*, saving data for subsequent subscribers), and persistence (*i.e.*, saving data for current subscribers).

Even though tunable policies provide fine-grained control of system QoS, several challenges emerge when developing pub/sub systems deployed in dynamic environments. Middleware mechanisms used to ensure certain QoS properties for one environment configuration may be ineffective for different configurations. For example, a simple unicast protocol, such as the User Datagram Protocol (UDP), may address the specified latency QoS 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 its point-to-point property, leaving the publisher to manage the sending of data to each subscriber.

Challenges also arise when considering multiple QoS policies that interact with each other. For example, a system might need low latency QoS and high reliability QoS, which can affect latency due to data loss discovery and recovery. Certain transport protocols, such as UDP, provide low overhead but no end-to-end reliability. Other protocols, such as Transmission Control Protocol (TCP), provide reliability but unbounded latencies due to acknowledgment-based retransmissions. Still other protocols, such as lateral error correction protocols [10], manage the potentially conflicting QoS properties of reliability and low latency, but only provide benefits over other protocols in specific environment configurations.

It is hard to determine when to switch from one transport protocol to another or modify

parameters of a particular transport protocol so that desired QoS is maintained. Moreover, manual intervention is often not responsive enough for the timeliness requirements of the system. DRE systems operate within strict timing requirements that must be met for the systems to function appropriately. The problem of timely response is exacerbated as the scale of the system grows (*e.g.*, as the number of publishers or subscribers increases).

Solution approach → Integrated Supervised Machine Learning Techniques and Flexible Transport Protocol Management for Timely and Accurate Autonomic Adaptation of DRE Pub/Sub Middleware. This article describes how our work (1) monitors environment changes that affect QoS, (2) determines in a timely manner which appropriate transport protocol changes are needed in response to environment changes, (3) integrates the use of multiple supervised machine learning techniques to increase accuracy, and (4) autonomically adapts the network protocols used to support the desired QoS. We have prototyped this approach in the ADAptive Middleware And Network Transports (ADAMANT) platform [47] that supports environment monitoring and provides timely autonomic adaptation of the middleware. ADAMANT provides the following contributions to research on autonomic configuration of pub/sub middleware in dynamic environments:

• **Leveraging anonymous publish and subscribe middleware based on the DDS specification.** DDS defines topic-based high-performance pub/sub middleware to support DRE systems. ADAMANT leverages the middleware to provide environment monitoring information that is disseminated throughout the DRE system (*e.g.*, change in sending rate, change in network percentage loss) to updates to the environment occur.

• **Multiple supervised machine learning (SML) techniques as a knowledge base to provide fast and predictable adaptation guidance in dynamic environments.** ADAMANT provides timely integrated machine learning (TIML), a novel approach to provide high accuracy and timely determination of which SML technique to use for a given operating environment.

• **Configuration of DRE pub/sub middleware based on guidance from supervised**

machine learning. Our ADAMANT middleware uses the adaptive network transports (ANT) framework [45] to select the transport protocol(s) that best addresses multiple QoS concerns for given computing resources. ANT provides an infrastructure for composing and configuring transport protocols using modules that provide base functionality (*e.g.*, an IP multicast module that handles multicasting the data to the network). Supported protocols include Ricochet, which uses a variation of forward error correction called lateral error correction that exchanges error correction information among receivers [9], and NAKcast, which uses negative acknowledgments (NAKs) to provide reliability. These protocols enable trade-offs between latency and reliability to support middleware for enterprise DRE pub/sub systems.

IV.2.2 Motivating Example - Ambient Assisted Living in a Smart City Environment

This section describes Smart City Ambient Assisted Living (SCAAL) applications, which combine Ambient Assisted Living (AAL) in the context of a smart city. It also presents research challenges associated with SCAAL applications. SCAAL applications help motivate the need for managing QoS interactions and providing timely adjustments of transport protocols for QoS-enabled pub/sub middleware deployed in dynamic environments. The objective for smart cities is to meld computational infrastructure into the surrounding environment and establish ubiquitous, context-aware services in a metropolitan area [25]. The purpose of AAL is to increase the independence and quality of life for elderly people, while decreasing the need for direct interaction of healthcare workers so they are freed up for other concerns.

As an example SCAAL scenario depicted in Figure 68, imagine an elderly person is navigating a large metropolitan area equipped with multiple technological devices. These devices aid in various aspects of the person's ability to be aware of her environment (*e.g.*, mobility, sensory enhancement, communication, and monitoring devices). In particular, the elderly person has a history of heart disease and 3-dimensional high-resolution heart

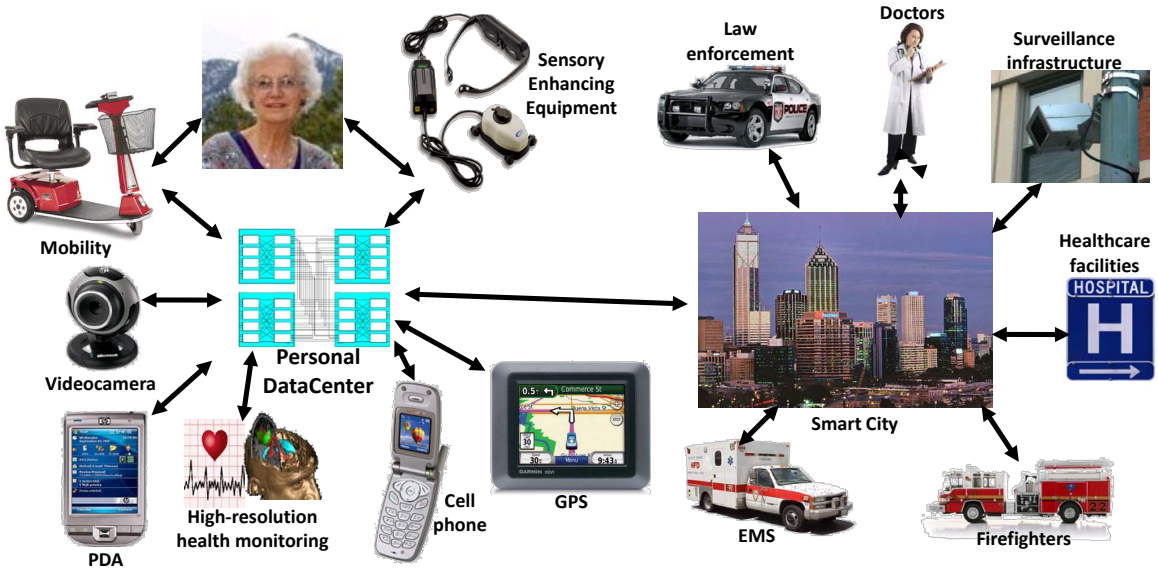

Figure 68: Smart City Ambient Assisted Living (SCAAL) Example

monitoring equipment is periodically transmitting data. A personal datacenter publishes and subscribes to the data being managed by the personal devices including the heart monitoring data, and interfaces with the smart city by publishing and subscribing to data from the ambient environment. More specifically, health care workers, hospitals, and emergency medical services specialists are subscribing to the heart monitoring information that is being published.

The personal datacenter operates in a dynamic environment since (1) the elderly person moves through space in the smart city and updates personal information in time and (2) the smart city enhances and updates the amount and kind of data that it provides as it moves through time. Our research focuses on (1) composite metrics to evaluate transport protocols in support of multiple QoS concerns (such as reliability and low latency for high-resolution 3D heart monitoring information), (2) evaluations of multiple transport protocols in different operating environments using the composite metrics, (3) support for monitoring the environment, (4) supervised machine learning techniques to determine transport protocols that best support the QoS that a personal datacenter device must manage in a SCAAL application, and (5) autonomically adapting the transport protocols to provide the best QoS

given the changes in the environment. Supporting autonomic adaptation of the personal datacenter presents the following challenges:

Challenge 1: Managing interacting QoS requirements. The personal datacenter must manage multiple interacting QoS requirements (*e.g.*, data reliability so enough data is received and low latency and jitter for soft realtime data so that detailed 3-dimensional heart monitoring data arrive before they are needed). For example, the streamed data must be received soon enough so that successive dependent data can also be used, such as dependent MPEG B and P frame data being received before the next I frame makes them obsolete. Moreover, the personal datacenter must balance the interacting QoS requirements with an environment that varies dynamically (*e.g.*, number of data senders and receivers, network bandwidth, network packet loss). Section IV.2.4.3: *Addressing Challenge 1: Managing Interacting QoS Requirements* describes how we address this challenge by supporting runtime migration and reconfiguration in bounded time of transport protocols used as the QoS mechanisms to provide needed QoS.

Challenge 2: Accurate Adaptation. The personal datacenter must be able to adjust to changes in the environment accurately. As changes in environment occur (*e.g.*, increases in heart data updates, decreases in networking capability, requests for data from additional senders and receivers), the personal datacenter must accommodate data needs for data producers and consumers, take advantage of additional resources, or provide access to additional data producers and consumers while maintaining QoS. For a given environment configuration, the SCAAL application must accurately implement adjustments that are appropriate to the operating environment. If the personal datacenter cannot make accurate adjustments as the environment changes then situation awareness and critical health information could be lost or delayed causing loss of orientation or injury to the elderly person. Section IV.2.4.3: *Addressing Challenge 2: Accurate Adaptation* describes how we address this challenge by leveraging DDS to disseminate the environment monitoring information

needed to determine an accurate adaptation and TIML to accurately determine the appropriate transport protocol.

Challenge 3: Timely Adaptation. Due to timeliness concerns of DRE systems such as SCAAL applications, the personal datacenter must adjust in a timely manner as the environment changes. If the personal datacenter cannot adjust quickly enough it will fail to perform adequately and critical data such as 3-D heart information will not be received in time. As the amount of data relevant to the SCAAL application fluctuates and the demand for information varies with a corresponding change in the data update rate, the personal datacenter must be configured to accommodate these changes with appropriate responsiveness to maintain the specified quality of service. Configuration changes must not only be timely in general but they must also be bounded—and ideally constant time—so that critical information updates (such as health monitoring) are not lost or received too late to be of use. Section IV.2.4.3: *Addressing Challenge 3: Timely Adaptation* describes how we address this challenge by using constant-time complexity machine learning techniques, constant-time integration of these techniques, and constant-time migration of transport protocols.

Challenge 4: Reducing development complexity. Many elderly people can use a personal datacenter to improve their independence. Likewise, the health care industry can benefit from the decreased workload for health care providers. A personal datacenter that is developed for one particular elderly individual in a particular operating environment, however, might not work well for a different elderly individual in a different operating environment with different personal equipment. Personal datacenters should therefore be developed and configured readily between the different operating environments presented by different metropolitan areas, differences in personal equipment, and differences in the data needs of various individuals to leverage the personal datacenters across a wide range of individuals and locales. Section IV.2.4.3: *Addressing Challenge 4: Reducing Development Complexity* describes how we address this challenge by leveraging DDS to disseminate

environment updates, and using machine learning to map environment configurations to the appropriate transport protocols.

IV.2.3 Evaluating Supervised Machine Learning Techniques for DRE Systems

In this section we present the context for evaluating supervised machine learning techniques for use in DRE systems. We also present the empirical results of our evaluations. We use these results to determine the appropriate techniques for different situations (*i.e.*, environment configurations known *a priori* or unknown until runtime) so that the challenges in Section IV.2.2 are addressed.

IV.2.3.1 Context

An autonomic system operates by managing itself without external intervention [60]. Many enterprise DRE pub/sub systems autonomically (1) monitor their environment and (2) adjust their operational behavior as the environment changes since manual adjustment is tedious, slow, and error prone. For example, a shift in network reliability can prompt QoS-enabled middleware, such as DDS, to change mechanisms (such as the transport protocol used to deliver data) since some mechanisms provide better reliability than others in certain environments. Likewise, applications leveraging cloud computing environments where elastically allocated resources (*e.g.*, CPU speeds and memory) cannot be characterized accurately *a priori* may need to adjust to available resources (such as using compression algorithms optimized for the available CPU power and memory) at system startup. If adjustments take too long the mission(s) the system implements could be jeopardized.

One way to autonomically adapt enterprise DRE pub/sub systems involves policy-based approaches [5, 27, 67] that externalize and codify logic to manage the behavior of the systems. Policy-based approaches provide deterministic response times to guide appropriate adjustments given changes in the environment and can be optimized to ensure low latency performance. The complexity of developing and maintaining policy-based approaches for

enterprise DRE systems can be unacceptably high, however, since developers must determine and implement the policies which are applicable for certain environmental configurations. Moreover, developers must manage how the policies interact to provide needed adjustments.

Machine learning techniques support algorithms that allow systems to adjust behavior based on empirical data (*e.g.*, inputs from the environment. These techniques can be used to support autonomic adaptation by learning appropriate adjustments to various operating environments. Unlike policy-based approaches, however, machine learning techniques automatically recognize complex sets of environment properties, provide highly accurate support for environment properties not previously known or encountered, and make appropriate decisions accordingly.

Conventional machine learning techniques, such as decision trees [16] and reinforcement learning [20], have been used to address autonomic adaptation for non-DRE systems [43]. As shown in Section IV.1.4.3, these techniques are not well suited for enterprise DRE pub/sub systems, however, since they do not provide bounded times when determining adjustments [48]. Some techniques, such as reinforcement learning [19], explore the solution space until an appropriate solution is found, regardless of the elapsed time. Other techniques, such as decision trees, have time complexities that are dependent upon the specific data and cannot be determined *a priori*. Moreover, decision trees may contain decision branches that are much longer than others, thereby making the determination of appropriate adaptations unpredictable, which is undesirable for DRE pub/sub systems.

Supervised machine learning techniques with bounded times provide a promising way to addressing the accuracy, timeliness, and development complexity of DRE pub/sub systems. Some techniques, however, provide higher accuracy for environment configurations known *a priori*, whereas other techniques provide higher accuracy for environment configurations unknown until runtime. Since known and unknown environment configurations

are relevant in dynamic operating environments, both types of techniques should be leveraged when they are most appropriate.

In general, machine learning uses guidance from past known environments to handle new and unknown environments. This generality sacrifices some accuracy, however, that would otherwise be provided for known environments. Machine learning techniques that are specialized for the environments they have seen—and on which they have been trained—are said to be overfitted [29], which makes the accuracy comparable to policy-based approaches (*i.e.*, 100% accurate). Overfitted techniques are particularly suited for environments known *a priori* where the technique can be specialized for the known cases. Generalized machine learning techniques, however, are more appropriate for handling environments unknown until runtime.

This section describes how we evaluated both overfitted machine learning and generalized machine learning to (1) reduce the complexity of autonomic adaptive enterprise DRE pub/sub systems, (2) provide the constant time complexity required of DRE pub/sub systems, and (3) increase the adaptation accuracy over any one single machine learning technique. In particular, our approach tunes an artificial neural network (ANN) [85] (which is a technique modeled on the interaction of neurons in the human brain) to retain as much information about environment configurations and adjustments known *a priori* as possible (*e.g.*, greatly increasing the number of connections between input environment characteristics and output adjustments typically used in an ANN). Our approach also tunes a support vector machine (SVM) to provide highly accurate adaptations for environments unknown until runtime.

IV.2.3.2 Experimental Results and Analysis

This section presents the results of experiments that use ANNs and SVMs to determine development complexity, timeliness, and accuracy in selecting an appropriate transport

protocol for ADAMANT given a particular operating environment. We conducted experiments for both environments known *a priori* (*i.e.*, using the same data for training and testing of the machine learning technique) and environments unknown until runtime (*i.e.*, using mutually exclusive data for training and testing). The experimental input data used to train the ANNs and SVMs include ADAMANT with multiple properties of the operating environment varied (*e.g.*, CPU speed, network bandwidth, DDS implementation, percent data loss in the network), along with multiple properties of the application being varied (*e.g.*, number of receivers, sending rate of the data) as would be expected for an ad hoc datacenter used for SAR operations.

From our previous experiments that empirically evaluate how transport protocols perform in different operating environments [45] we gathered 394 inputs where an input consists of data values for features that determine a particular operating environment (*e.g.*, CPU speed, network bandwidth, number of data receivers, sending rate). Table 14 delineates the inputs for both of the machine learning techniques that ADAMANT utilizes (*i.e.*, ANNs and SVMs). The composite QoS metrics combine multiple QoS concerns into a single value. ReLate2 combines reliability and average network packet latency while ReLate2Jit extends the ReLate2 metric to also include standard deviation of packet arrival times (*i.e.*, packet jitter).

We also provided the expected output to the machine learning techniques, that is, the transport protocol that provided the best QoS with respect to data reliability, average latency, and jitter (*i.e.*, standard deviation of the latency of network packets) depending on which combination of these QoS properties were required. Both machine learning techniques used output a transport protocol and protocol settings given the operating environment inputs. An example of one of the 394 inputs is the following: 15 data receivers, 5% network loss, 100Hz data sending rate, 3GHz CPU, 1Gb network, using the OpenSplice DDS implementation, and specifying the combination of reliability and average latency as

Machine Learning Inputs	Values
Number of data receivers	3 to 25
Network bandwidth	1 Gb/s, 100 Mb/s, 10 Mb/s
DDS implementation	OpenDDS, OpenSplice
Percent end-host network loss	3 to 10%
CPU type	850 MHz, 3 GHz
Data sending rate	10 Hz, 25 Hz, 50 Hz, 100 Hz
Available RAM	500 MB, 2 GB
Composite QoS metric	ReLate2, ReLate2Jit

Table 14: Machine Learning Inputs

the QoS properties of interest. Based on our experiments, the corresponding output would be the Ricochet transport protocol with the *R* and *C* parameters set to 4 and 6, respectively.

We next empirically evaluated the accuracy of ANNs and SVMs for both environments known *a priori* and unknown until runtime. We ran experiments with various numbers of hidden nodes and stopping errors. We used the Fast Artificial Neural Network (FANN) library (leenissen.dk/fann) as our ANN implementation due to its configurability, documentation, ease of use, and open-source availability. FANN offers extensive configurability for the neural network including the number of hidden nodes that connect the inputs with the output. We used the libSVM library (www.csie.ntu.edu.tw/∼cjlin/libsvm) for our SVM implementation due to its configurability, documentation, tutorials, and open-source availability.

Evaluating the Accuracy of ANNs and SVMs for Environments Known *A Priori*. Below we evaluate the accuracy of ANNs and SVMs for environments known *a priori*. We first focus on ANNs and then SVMs. Our first step to measuring the accuracy of ANNs for environments known *a priori* was to train ANNs on the 394 inputs described in Section IV.1.5.4. We ran training experiments with the ANNs using different numbers of hidden nodes to determine the most accurate ANN. For a given number of hidden nodes we trained the ANN 10 different times. The weights of the ANNs determine how strong

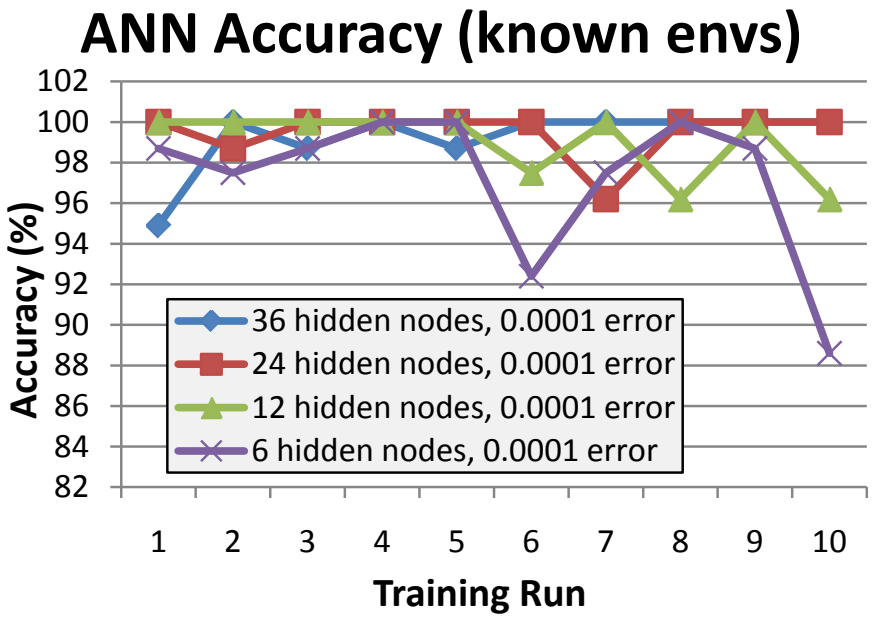

Figure 69: Accuracy of ANN with 6, 12, 24, & 36 Hidden Nodes & 0.0001 Stopping Error

connections are between nodes. The weights are randomly initialized and these initial values have an effect on how well and how quickly the ANN learns.

Figure 69 shows the accuracies for the ANNs configured with 6, 12, 24, and 36 hidden nodes and a 0.0001 stopping error over 10 training runs. Additional experiments were conducted with higher stopping errors (*e.g.*, 0.01) but lower stopping errors consistently produced more accurate classifications as expected. Figure 69 also shows the effect of random initial weights on the accuracy of the ANN since the accuracy can vary across training runs. Accuracy was determined by querying the ANN with the 394 inputs on which it was trained.

A 100% accurate classification was generated at least once with all ANN configurations. The ANN with 24 hidden nodes provided the best accuracy (as measured by the number of 100% accurate classifications) across all the training runs even compared to using 36 hidden nodes—100% accuracy all but 2 times out of 10. The ANNs with 36 and 12 hidden nodes both provided 100% accuracy 7 out of 10 times.

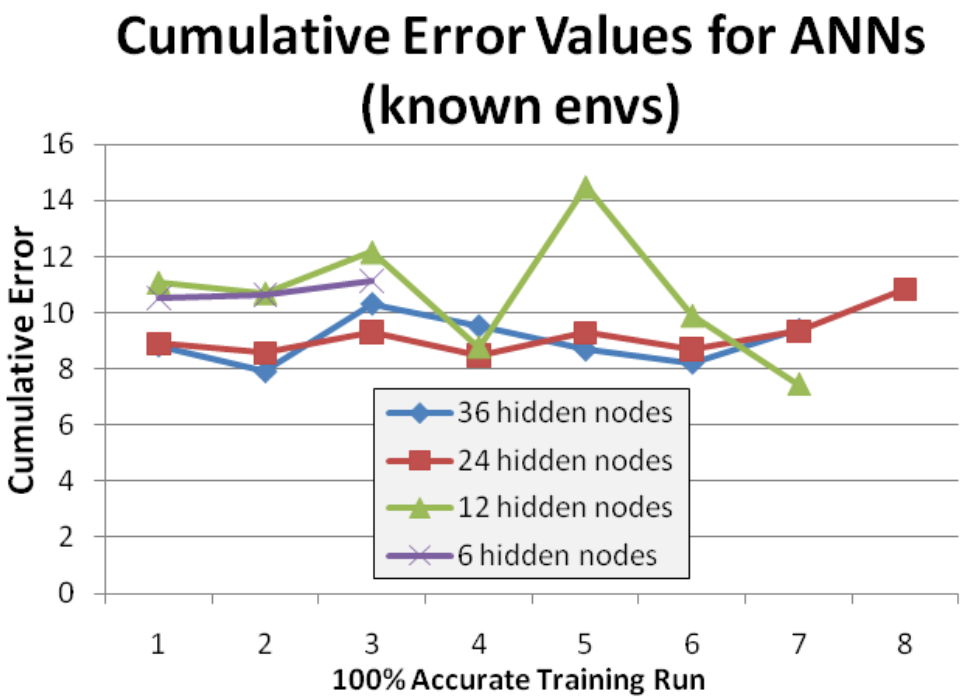

Figure 70: Cumulative Errors for ANNs with 100% Accuracy

The amount of error between values generated by the ANN and the known correct values is another measure of accuracy. The transport protocol output produced by the ANN is considered accurate if it is closer numerically to the correct protocol than to any other protocol. The ANN produces numerical values that are compared to the numerical representation of the correct protocol. The differences between these values can be accumulated across the 394 inputs and compared between different ANN configurations (*e.g.*, 36 hidden nodes, 24 hidden nodes).

Figure 70 shows the errors accumulated for all the data of a single training & test run (*i.e.*, using the 394 inputs). The training runs here are only included if the ANN had 100% accuracy (as shown by Figure 69) in that run since we want 100% accuracy for environments known *a priori*. For example, the ANN configured with 6 hidden nodes had only 3 training runs where the ANN was accurate for 100% of the 394 inputs and thus only 3 data points exist for that ANN configuration.

Across all the runs where the ANNs produced 100% accuracy, the ANN with 36 hidden

nodes produced the lowest average error (*i.e.*, 8.99). The ANN with 24 hidden nodes, however, produced the second lowest average error (*i.e.*, 9.18) with only a 2.1% increase over the lowest error value. In this regard there is not a significant difference in the cumulative errors between the ANNs configured with 36 and 24 hidden nodes. By contrast, the 3rd lowest average error (*i.e.*, 10.65 using 12 hidden nodes) represents a 19% increase.

As with the ANNs, we trained the SVMs on the 394 inputs to compare accuracy for environments known *a priori*. SVMs are a specialized kind of linear classifier that utilize kernels, which create additional features from the inputs that can be used for training and classification [99]. A variety of kernels can be used, such as radial basis function (RBF) and polynomial kernels, which produce different levels of accuracy.

We ran training experiments with the SVMs using different kernels and scaling the data in different ways. In addition to RBF and polynomial kernels, we included a simple linear classifier as a baseline to compare to the SVMs. The libSVM library provides the flexibility to utilize the SVMs with RBF and polynomial kernels as well as a simple linear classifier.

In addition, we scaled the data in 4 different ways: (1) no scaling (*i.e.*, using the original environment configuration values), (2) scaling the input values (*i.e.*, environment configurations) to be between -1 and 1, (3) scaling the input to be between -1 and 1 and scaling the output (*i.e.*, the transport protocol specified) to be between 0 and 1, and (4) scaling both input and output to be between -1 and 1. Since there is no nondeterminism with training SVMs there is no need to have multiple training runs for a single SVM configuration (*e.g.*, SVM with polynomial kernels and no scaling of data).

Figure 71 shows the accuracies for the linear classifier and the SVMs using the RBF and polynomial kernels. The SVM with the RBF kernel (SVM-RBF) was able to correctly predict the appropriate transport protocol for 100% of the environment configurations. This result is somewhat surprising since SVMs are designed to generalize their learning and to handle environment configurations unknown until runtime. The SVM using the polynomial kernel (SVM-Polynomial) and the linear classifier produced their highest accuracy when

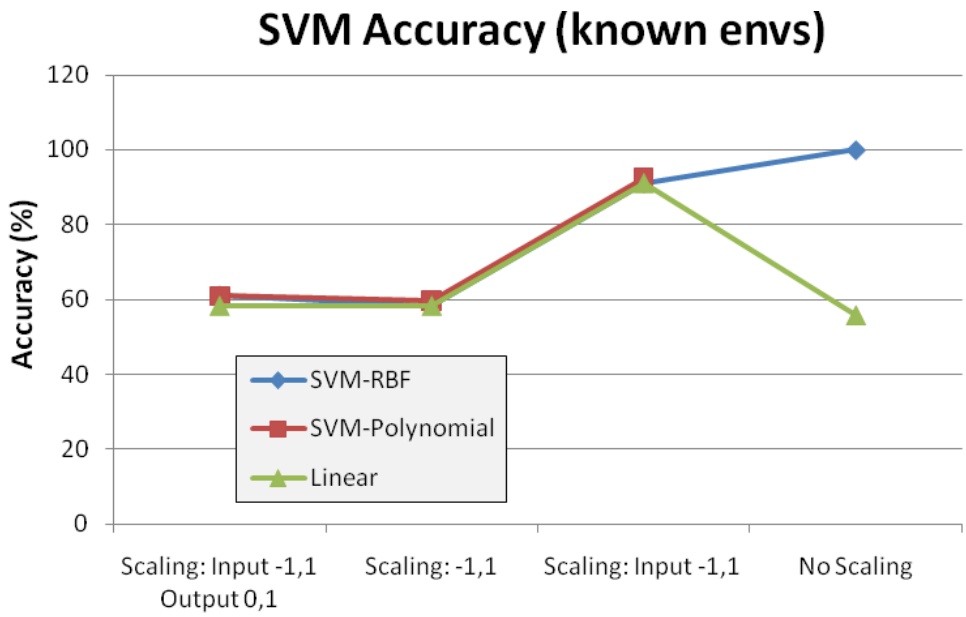

Figure 71: Accuracy of SVMs with RBF, Polynomial, and Linear Kernels

the input data was scaled between -1 and 1 (92.39% and 91.17% accuracy respectively). Figure 71 also shows the affect that data scaling has on accuracy.

SVM-RBF had its highest accuracy (100%) when the data was not scaled while the linear classifier and SVM-Polynomial had their highest accuracy when only the input data was scaled. SVM-Polynomial was not able to complete training when the data was not scaled and therefore has no accuracy for that case. SVM-RBF, SVM-Polynomial, and the linear classifier all had their lowest accuracies when the output data (*i.e.*, the selected transport protocol) was scaled. Figures 69, 70, and 71 show that ANNs with 24 and 36 hidden nodes and an SVM with RBF kernels and unscaled data produce the highest accuracies when the environment configurations are known *a priori*.

Evaluating the Accuracy of ANNs and SVMs for Environments Unknown until Runtime. Below we measure the accuracy of ANNs and SVMs for environment configurations that are unknown until runtime. This scenario splits out the 394 environment configurations into mutually exclusive training and testing data sets, which is referred to as n-fold cross-validation where n is the number of mutually exclusive training and testing

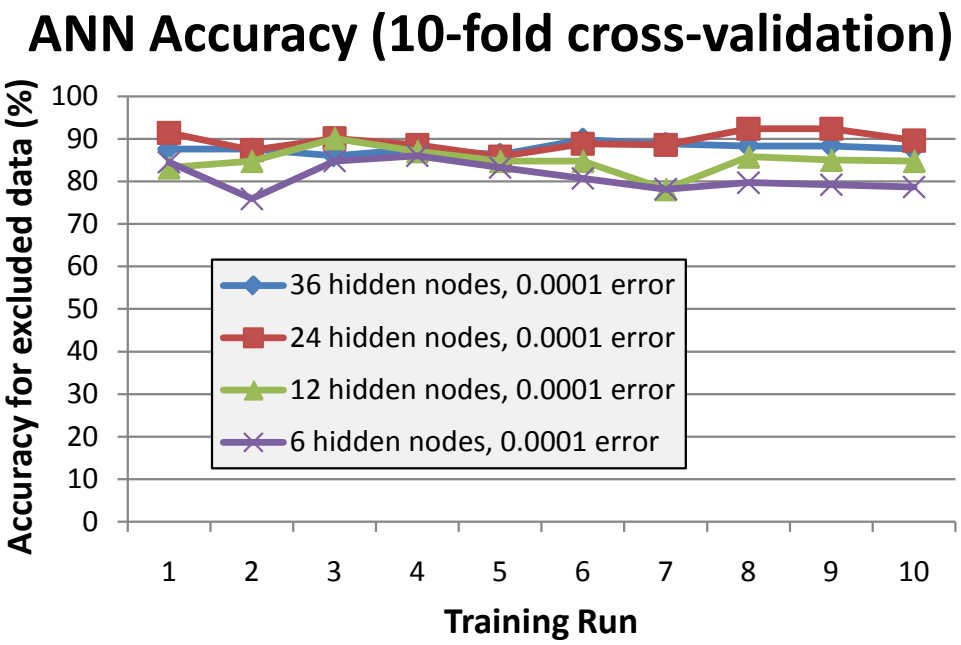

Figure 72: ANN Accuracies for 10-fold Cross-validation (0.0001 Stopping Error)

data sets [68]. The value of n also determines the amount of data excluded from training and used only for testing. A 10-fold cross-validation indicates 10 sets of training and testing data where for each fold the training and testing data are mutually exclusive and the training data excludes 1/10 of the total data, which is used only for testing. N-fold cross-validation provides insight into how well a machine learning technique generalizes for data on which it has not been trained.

We started by examining the accuracy of ANNs for environments unknown until runtime using 10-fold cross-validation. Since we are focusing on generalizing the machine learning for environment configurations not previously encountered, we added ANN configurations to those listed in Section 4.2.1 to include a larger stopping error to see the effect on accuracy. Figure 72 shows the accuracy for the excluded data across the 10-folds for 10 different training runs. The ANNs in this figure all use the stopping error of 0.0001. Figure 73 shows the accuracy for the excluded data across the 10-folds using a stopping error of 0.01. We split out these data into 2 separate figures for clarity.

As Figures 72 and 73 show, the accuracy for determining the correct transport protocol

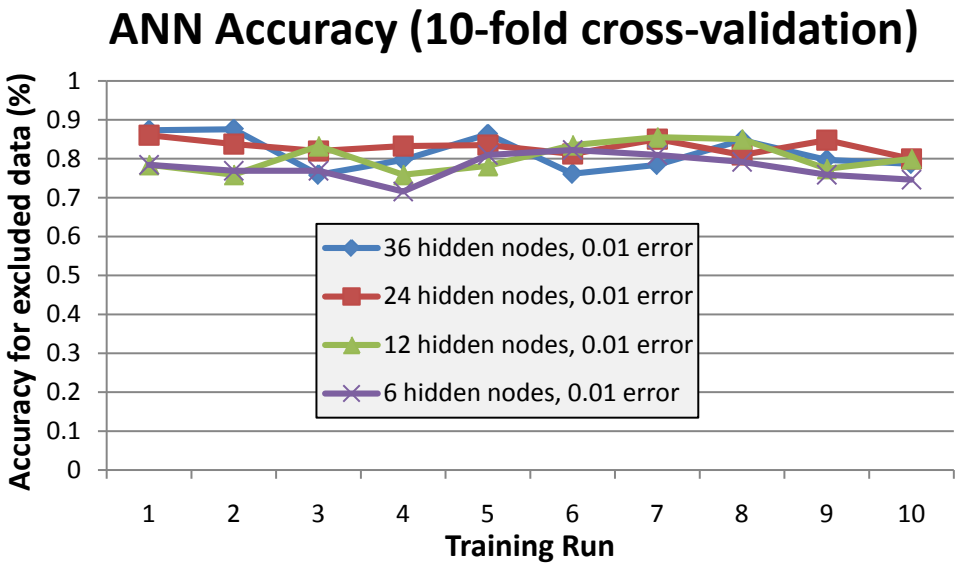

Figure 73: ANN Accuracies for 10-fold Cross-validation (0.01 Stopping Error)

given the unknown environment configurations was highest with the lower stopping error of 0.0001. The lowest accuracy for any run with a stopping error of 0.0001 was 75.89% (for 6 hidden nodes) while the lowest accuracy for any run with a stopping error of 0.01 was 71.57% (again for 6 hidden nodes). The highest accuracy was obtained by 24 hidden nodes and stopping error of 0.0001 for training run 9 (*i.e.*, 92.39%). This configuration also produced the highest average accuracy across all the training runs (*i.e.*, 89.49%) with the ANN configuration of 36 hidden nodes and a stopping error of 0.0001 producing the second highest average accuracy across all the training runs (*i.e.*, 87.79

To evaluate the linear classifier and SVMs, we used the same 10-fold cross-validation data described earlier in this section. Figure 74 shows accuracies for the excluded data for the same linear classifier and the SVMs described in Section 4.2.1. As in Figure 71, SVM-RBF again produces the highest accuracy for all the SVMs and the linear classifier (*i.e.*, 87.56%). This accuracy was produced, however, when the input data was scaled to values between -1 and 1. When no scaling of the data was performed, the accuracy of SVM-RBF decreased by 14.49% (*i.e.*, 74.87% accuracy).

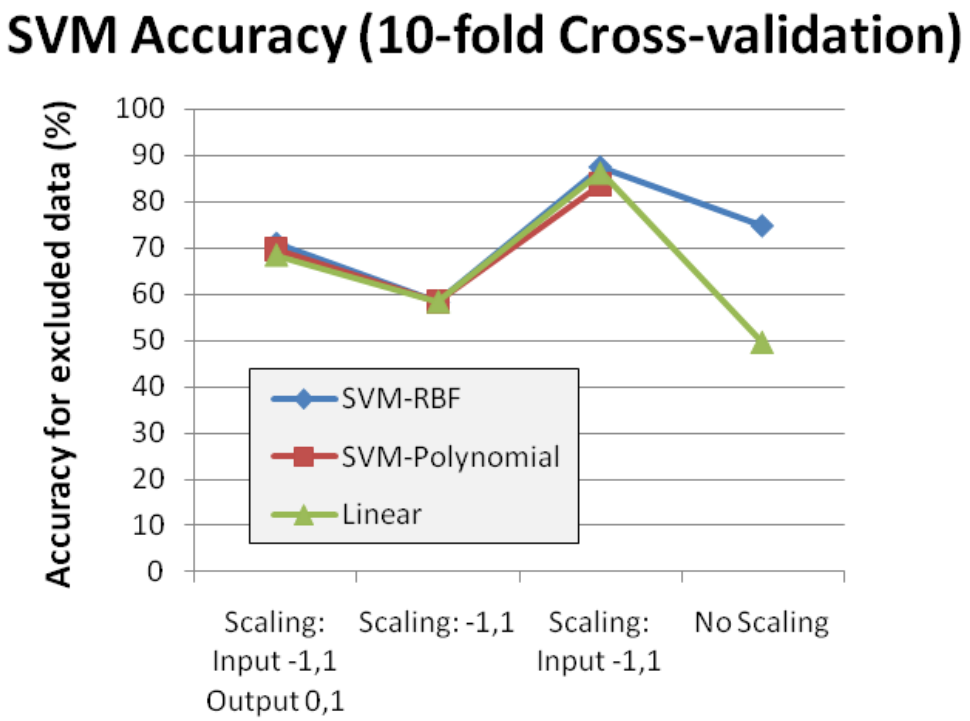

Figure 74: SVM Accuracies for 10-fold Cross-validation (0.01 Stopping Error)

SVM-RBF produced the highest accuracies across all data scalings compared to SVM-Polynomial and the linear classifier. As was the case with known environments, SVM-Polynomial was unable to complete training when the data was unscaled. When the input data was scaled between -1 and 1, SVM-RBF, SVM-Polynomial, and the linear classifier all produced their highest accuracies.

With the current experiments involving environments unknown until runtime, the ANN with 24 hidden nodes and a stopping error of 0.0001 produced the highest accuracy. To see if these results would hold with an increased percentage of unknown data, we created 2-fold cross-validation data. For this set of data, half of the environment configurations would be used to train the ANNs and SVMs and the other half would be used to test the ANNs and SVMs for accuracy.

Figures 75 and 75 show the accuracy for 2-fold cross-validation of ANNs configured with 6, 12, 24, and 36 nodes and 0.0001 and 0.01 stopping errors respectively. The highest average accuracy across the 10 trainings run is produced by the ANN with 36 hidden nodes

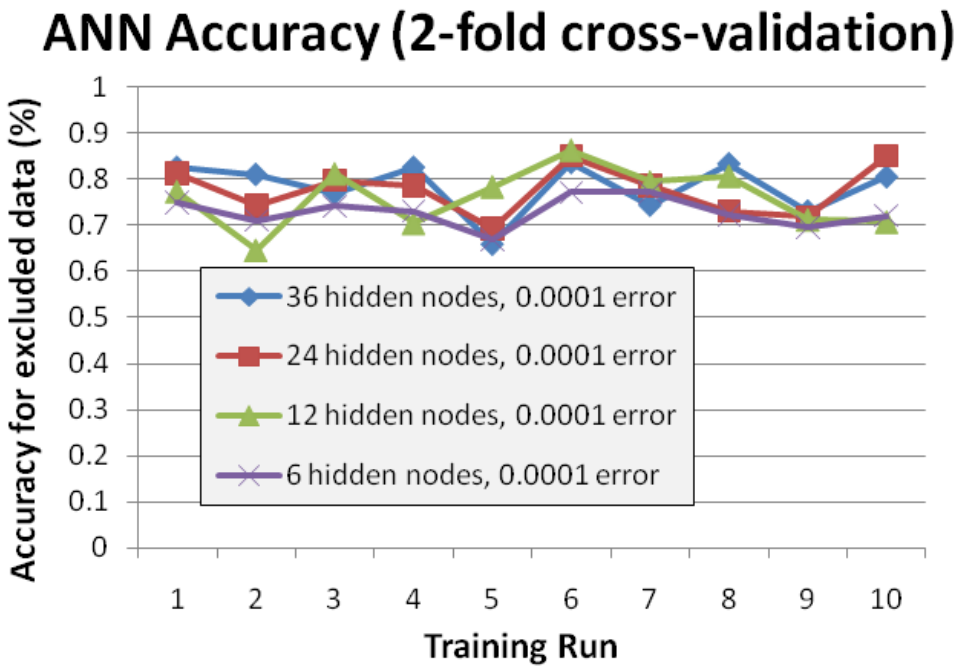

Figure 75: ANN Accuracies for 2-fold Cross-validation (0.0001 Stopping Error)

and a stopping error of 0.0001 (*i.e.*, 78.35% average accuracy). The second highest average accuracy is now produced by the ANN with 24 hidden nodes and a stopping error of 0.0001 (*i.e.*, 77.69% average accuracy). It is also interesting to note that the ANN with 6 hidden nodes and a stopping error of 0.01 (*i.e.*, 74.19% average accuracy) produced higher average accuracies than with a stopping error of 0.0001. This result indicates that the ANN with 6 nodes generalizes its learning better with a higher stopping error.

Figure 77 shows the accuracy of SVM-RBF, SVM-polynomial, and the linear classifier for 2-fold cross-validation. In this case, both SVM-RBF and SVM-Polynomial produce the highest accuracy (*i.e.*, 86.29%) when the input data is scaled from -1 to 1. Scaling the input data this way again produces the highest accuracies for SVM-RBF, SVM-Polynomial and the linear classifier as it did for 10-fold cross-validation. SVM-RBF also produced the worst 2-fold cross-validation accuracy when the input data was scaled from -1 to 1 and the output data was scaled from 0 to 1 (*i.e.*, 49.49% accuracy). Overall, SVM-RBF and

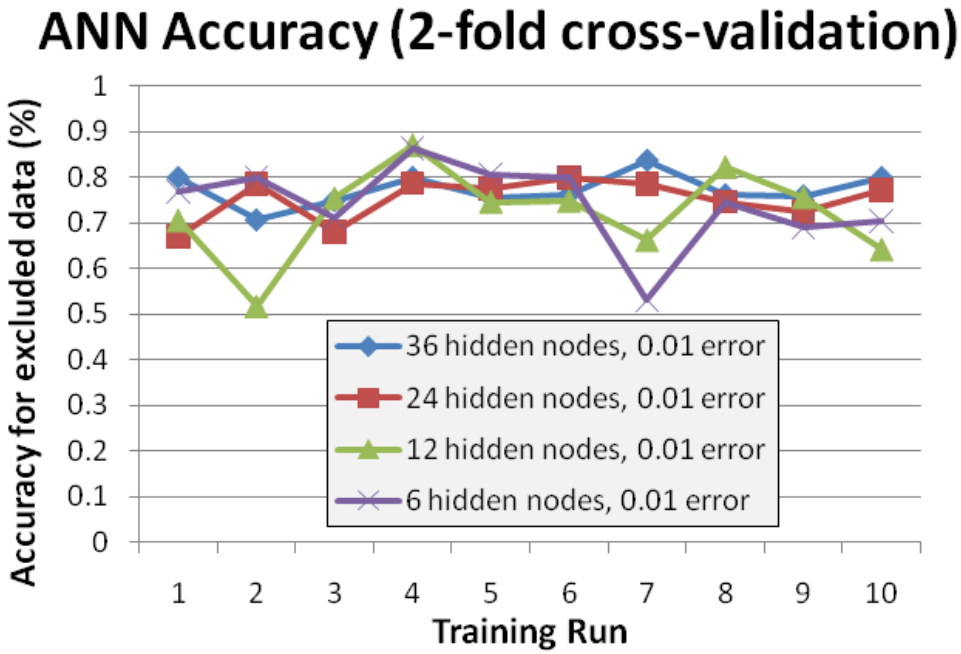

Figure 76: ANN Accuracies for 2-fold Cross-validation (0.01 Stopping Error)

SVM-Polynomial produced the most accurate results for 2-fold cross-validation even when including the accuracy results for the ANNs.

Evaluating the Timeliness of ANNs and SVMs. For the SAR motivating example, we are not only interested in accurate transport protocol guidance. We also need low latency, constant time-complexity to meet the needs of DRE systems as outlined in Section 2.1.2. We now empirically evaluate the runtime timeliness of ANNs and SVMs.

To gather timing data, we used a 3 GHz CPU with 2GB of RAM running the Fedora Core 6 operating system with realtime extensions. Timeliness was determined by querying the ANNs and SVMs with all 394 inputs on which it was trained (*i.e.*, timing was done for the case of environments known *a priori*). A high resolution timestamp was taken before and after each call made to the ANN, the SVM, or the linear classifier and the times were calculated by subtracting the timestamp taken before the call from the timestamp taken after the call.

Figures 78 and 79 show the average response times and standard deviation of the

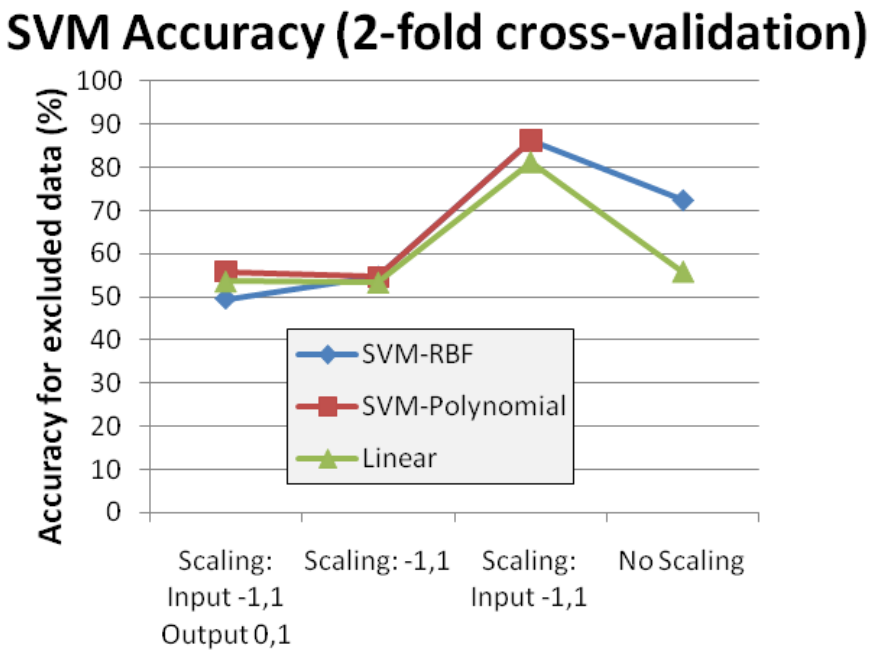

Figure 77: SVM Accuracies for 2-fold Cross-validation

response times for ANNs, respectively, for 10 separate experiments where for each experiment we query the ANN for each of the 394 inputs. The figures show that the ANN provides timely and consistent responses. The standard deviations for all the ANNs and all the classification runs are below 1 μs. As expected, the response times using more hidden nodes are slower than response times with fewer hidden nodes. The increase in latency is less than linear, however (*e.g.*, response times using 12 hidden nodes are less than twice that using 6 hidden nodes).

Figures 80 and 81 show the average response times and standard deviations of the response times, respectively, for SVM-RBF, SVM-Polynomial, and the linear classifier where the data is scaled as described in Section 4.2.1. When trained on data that has been scaled differently, there is a difference in the average response time. The more data is scaled, the lower the average response time is (*e.g.*, scaling all the data values to be between -1 and 1 vs. no scaling). SVM-RBF has the highest average response times. This result is not surprising since the RBF kernels create more complicated calculations to determine the appropriate transport protocol. The standard deviations for the response times are fairly

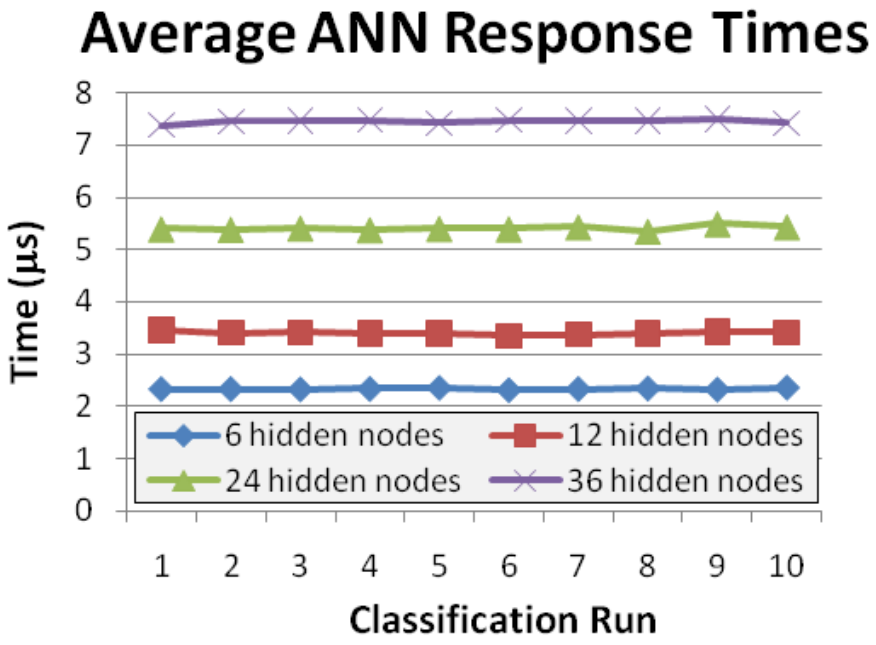

Figure 78: ANN Average Response Times (μs)

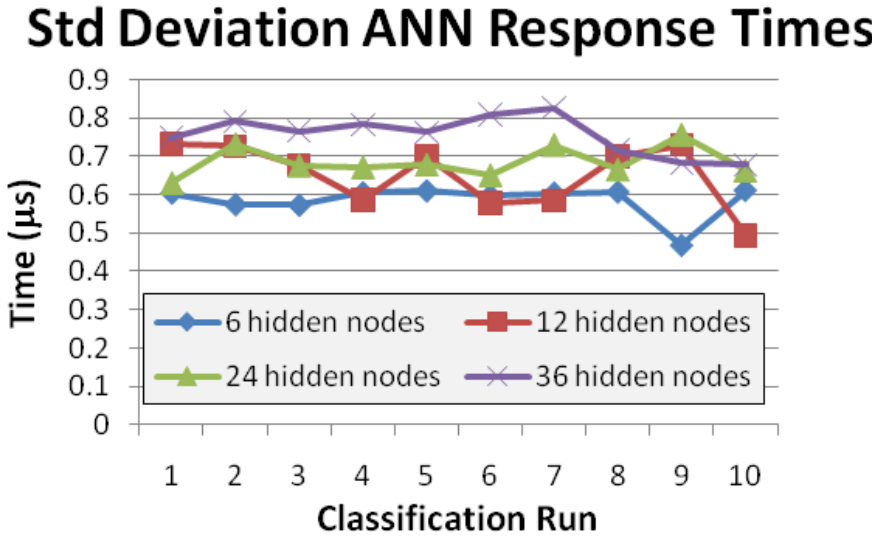

Figure 79: Standard Deviation for ANN Response Times (μs)

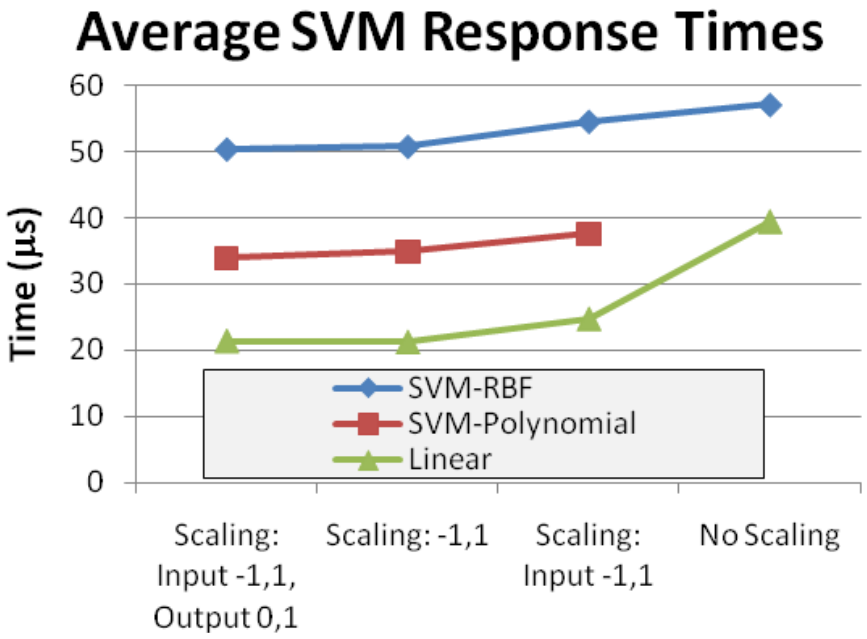

Figure 80: SVM Average Response Times (μs)

consistent regardless of scaling. SVM-RBF exhibits an unusually high standard deviation when the data is not scaled. This data might show an anomaly with the libSVM library with respect to timeliness since we also needed to modify the implementation to remove sources of unbounded time complexity for testing timeliness (*e.g.*, replace calls to *malloc()* with statically allocated memory since *malloc()* provides no bounded time guarantees).

Analyzing the Experimental Results. The ANN with 24 hidden nodes and a stopping error of 0.0001 produced the most accurate result for experiments dealing with environments known *a priori*. The ANN with 36 hidden nodes was comparable in its accuracy for environments known *a priori*. When considering the timeliness advantage of 24 hidden nodes compared to 36 hidden nodes, however, the ANN with 24 hidden nodes seems the most appropriate for environments known *a priori*.

With the 10-fold cross-validation experiments, the ANN with 24 hidden nodes and a stopping error of 0.0001 produced the highest accuracy over the other ANN configurations, the linear classifier, and the SVMs. This result is somewhat surprising since SVMs are designed to be more general and handle unknown input well whereas over-fitted ANNs

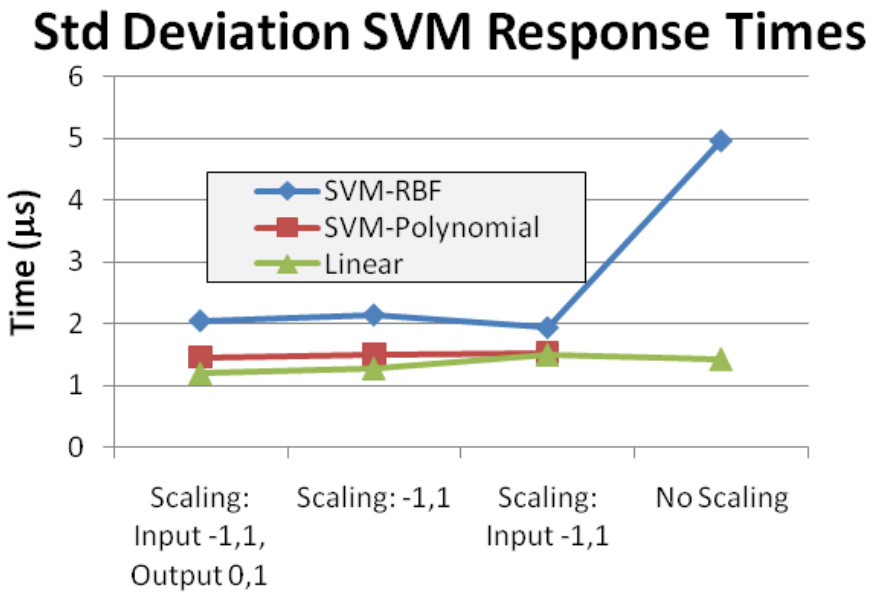

Figure 81: Standard Deviation for SVM Response Times (μs)

are not expected to do as well. However, with the 2-fold cross-validation experiments, the ANN with 36 hidden nodes and a stopping error of 0.0001 produced more accurate results averaged over 10 training runs than the ANN with 24 hidden nodes and the same stopping error. These differing results indicate that if only ANNs are used for machine learning, we should use one type of ANN for environments known *a priori* (*i.e.*, 24 hidden nodes and stopping error of 0.0001) and one type for environments unknown until runtime (*i.e.*, 36 hidden nodes and a stopping error of 0.0001). We speculate that the excluded environment configurations for the 10-fold cross-validation was a small enough percentage of the overall data that it was not different enough to cause a different ANN configuration or an SVM to be more accurate. The results of the 2-fold cross-validation support this conclusion.

The experimental results shown in this section illustrate that different supervised machine learning techniques provide different levels of accuracy depending on whether the environment configuration is known *a priori* or unknown until runtime. Based on these experimental data, we have chosen to use an ANN with 24 hidden nodes and a stopping error of 0.0001 for environments known *a priori* due to its (1) perfect accuracy and (2)

low latency constant classification time. We have also chosen to use an SVM with a polynomial kernel for environments unknown until runtime since this technique (1) provided the highest accuracy for all the configurations of ANNs and SVMs tested with the most generalized data for validation (*i.e.*, 2-fold cross validation data) and (2) provided lower response latencies than the SVM with the RBF kernel. We combine both of these machine learning techniques in our TIML approach to increase the overall accuracy of ADAMANT and leverage constant-time perfect hashing to determine if the environment configuration is known *a priori* or unknown until runtime and use the machine learning that will provide the highest accuracy.

IV.2.4 Structure and Functionality of ADAMANT

This section presents the structure and functionality of the ADAMANT middleware platform, focusing on its software architecture and control flow. It also describes how ADAMANT addresses the challenges of SCAAL applications presented in Section IV.2.2.

IV.2.4.1 Architecture of ADAMANT

Figure 82 shows ADAMANT's control flow and logical architecture. This section details the architecture of ADAMANT while the following Section IV.2.4.2 describes how autonomic adaptation is manifested in ADAMANT in each one of the steps illustrated in Figure 82.

ADAMANT integrates and enhances the following technologies and innovative techniques to provide autonomic adaptation of DRE pub/sub middleware in dynamic environments and address the challenges listed in the motivating example section:

• The OMG Data Distribution Service (DDS) is standards-based QoS-enabled pub/sub middleware for exchanging data in event-based DRE systems. It provides a global data store in which publishers and subscribers write and read data, respectively. ADAMANT

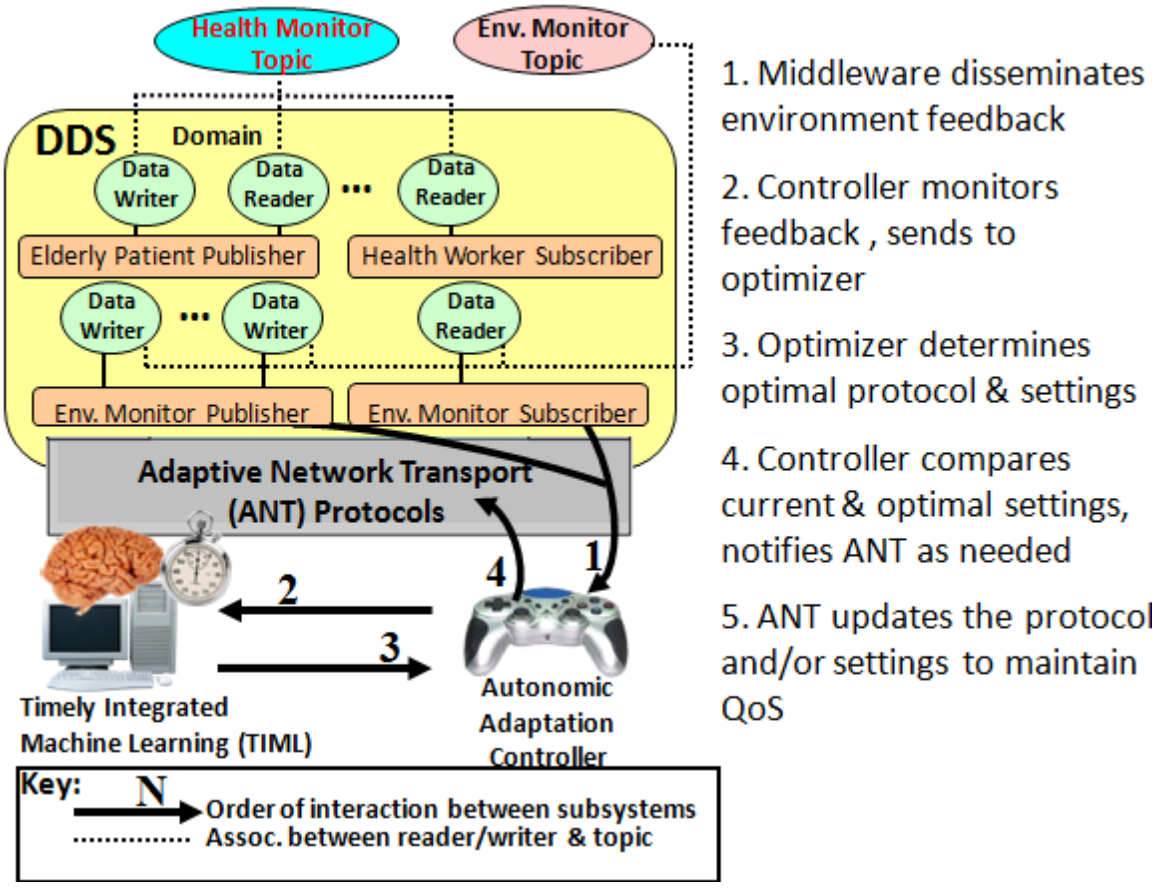

Figure 82: ADAMANT Architecture and Control Flow for SCAAL Applications

uses DDS to provide the infrastructure for disseminating environment monitoring information needed to determine accurate adaptations, as well as normal application data, such as the health monitoring information in SCAAL applications. DDS enables applications to communicate by publishing information they have and subscribing to information they need in real time.

DDS enables flexibility and modular structure by decoupling; location, via anonymous publish/subscribe; redundancy, by allowing any numbers of readers and writers; time, by providing asynchronous, time-independent data distribution; and platform, by supporting a platform-independent model that can be mapped to different languages (*e.g.*, Java and C++).

The DDS architecture consists of two layers: (1) the datacentric pub/sub (DCPS) layer that provides APIs to exchange topic data based on chosen QoS policies and (2) the data local reconstruction layer (DLRL) that makes topic data appear local. Our work focuses on DCPS since it is more broadly supported than the DLRL. Moreover, DCPS provides finer grained control of QoS.

The DCPS entities in DDS include topics, which describe the type of data to write 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. Moreover, publishers manage groups of data writers and subscribers manage groups of data readers. Various properties of these entities can be configured using combinations of the 22 DDS QoS policies shown in Table 2. DDS' rich support for QoS can be applied for application data and for the environment monitoring topic that ADAMANT provides (*e.g.*, prioritization for transporting and managing the operating environment updates as well as the application data).

• TIML provides a novel integration of multiple supervised machine learning techniques as a knowledge base. This knowledge base, in turn, provides fast and predictable adaptation guidance in dynamic environments. TIML uses machine learning techniques

to manage the inherent complexity of providing the appropriate transport protocol recommendation for a given operating environment. TIML utilizes perfect hashing [17] on the mapping of environment configurations to transport protocols to provide constant-time determination of which supervised machine learning technique to use for a given environment configuration. In particular, TIML utilizes the GPERF [70] open-source implementation of perfect hashing.

For our ADAMANT prototype TIML uses several supervised machine learning techniques, including Artificial Neural Networks (ANNs) [85] to determine in a timely manner the appropriate transport protocol for the QoS-enabled pub/sub middleware platform given an environment configuration that is known *a priori* (*i.e.*, used for training). It also uses Support Vector Machines (SVMs) [73] to determine in a timely manner the appropriate transport protocol for an environment configuration unknown until runtime (*i.e.*, not used for training).

An ANN is a supervised machine learning technique modeled on neuron interactions in the human brain. As shown by Figure 46 in Section IV.1.4.3, an ANN has an input layer for aspects of the operating environment (*e.g.*, percent network loss and sending rate). An output layer represents the solution generated based on the input. A hidden layer connects the input and output layers. As the ANN is trained on inputs and correspondingly correct outputs, it strengthens or weakens connections between layers to generalize based on inputs and outputs.

Figure 46 also shows how an ANN can be configured statically in the number of hidden layers and the number of nodes in each layer that directly affects the processing time complexity between the input of operating environment conditions and the output of an appropriate transport protocol and settings. This static configuration structure supports bounded response times.

SVMs are supervised learning methods used for classification and prediction. Given a set of training examples where each example is denoted as belonging to a particular class

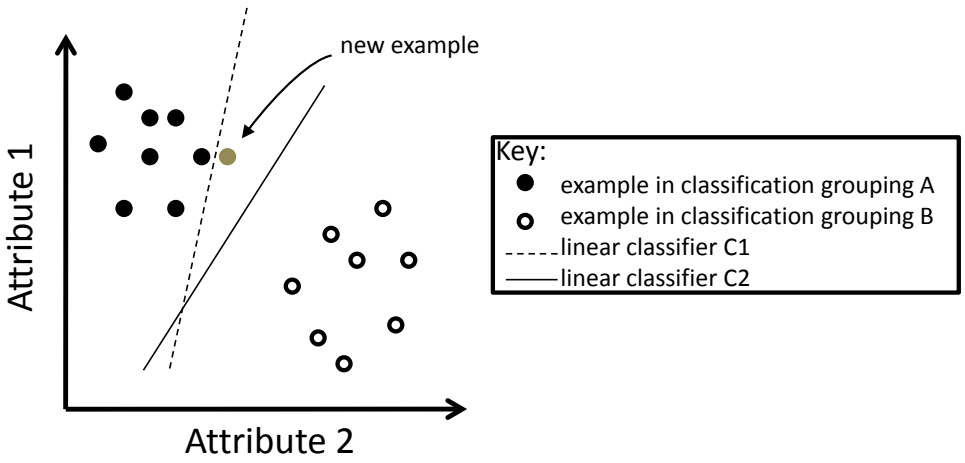

Figure 83: Maximizing Grouping Differences in a Support Vector Machine

or grouping, an SVM builds a model that predicts into which grouping a new example should be categorized. The SVM generates the boundaries between the different groupings to maximize the differences between the groupings. This maximization helps to correctly classify new examples that have not been used in training the SVM model using the heuristic of locality (*i.e.*, examples that belong in the same group should be fairly close to each other in the classification space).

Figure 83 illustrates conceptually how an SVM makes its determination for classification boundaries. The examples in grouping A are represented by solid circles while the examples in grouping B are represented by hollow circles. For simplicity and clarity, the examples are classified using two attributes. The dashed line C1 and the solid line C2 represent two different classifiers. An SVM produces a classifier similar to C2 since its margin between the two classification groupings is larger than with C1. A new example (*i.e.*, a solid grey circle) that needs to be classified using the SVM belongs in classification grouping A. The example is close to the line C1 but on the opposite side of the rest of the examples for that grouping and is therefore incorrectly classified whereas with C2 the new example would be correctly classified. An SVM maximizes the margin of differences between classification groupings.

- ADAMANT uses the ANT framework to select the transport protocol(s) that best

address multiple QoS concerns for a given operating environment. ANT provides infrastructure for composing and configuring transport protocols via base modules, such as the IPMulticastModule that supports sending out and receiving data using IP Multicast. These modules can flexibly and dynamically be connected together by publishing and subscribing to event types (*e.g.*, SEND_PACKET_EVENT, GOT_PACKET_EVENT, SEND_NAK_EVENT, and GOT_NAK_EVENT).

ANT supports transport protocols that balance the need for reliability and low latency. For example, Ricochet enables trade-offs between latency and reliability to support middleware for DRE pub/sub systems involved with dissemination of multimedia data. The ANT framework allows ADAMANT to change and reconfigure transport protocols (including protocol parameters) while an application is running. The time complexity for ANT to reconfigure and transition between protocols is bounded as needed for DRE systems.

• A QoS monitoring topic defines the data for environment information relevant to adapting transport protocols. ADAMANT leverages DDS to provide this topic dedicated to describing the operating environment of an application. This environment information is used to determine appropriate adaptation of the QoS mechanisms in ADAMANT, namely, the transport protocols. Moreover, since ADAMANT leverages DDS to create the environment monitoring topic, DDS QoS policies can also be applied to the dissemination of this topic data providing fine-grained control as to when and how environment configuration updates are propagated in the SCAAL application (*e.g.*, applying DDS' transport priority QoS policy to health monitoring data to ensure the data has priority over other data on the network).

• Autonomic control manages the adaptation process. ADAMANT provides an autonomic controller that responds to changes in the operating environment. Whenever environment changes are communicated via ADAMANT's environment monitoring topic, the

controller passes the changes to TIML to determine the appropriate response. The controller then passes TIML's recommended adaptations to the ANT framework to change the transport protocols while the system is running.

IV.2.4.2 Control Flow of ADAMANT

ADAMANT supports the Monitor, Analyze, Plan, Execute - Knowledge approach [50], which abstracts the management architecture into the four needed functions of collecting data, analyzing the data, creating a plan of action based on the updated data and corresponding analysis, and executing the plan. ADAMANT components are physically distributed across the computing platforms in the system (*e.g.*, each computing platform has its own identical instantiation of TIML and the autonomic adaptation controller). Since environment configuration changes are published to all subscribers via DDS, all local ADAMANT components receive the same updates. Since components are deterministic, they generate the same transport protocol to use and initiate the same protocol modifications. This distributed architecture enables scalability in the number of publishers, subscribers, and computing platforms.

The first step in ADAMANT's control flow (shown as Step 1 in Figure 82) is receiving changes to the environment configuration. ADAMANT creates and supports an environment monitoring topic to which various application data senders and receivers can publish and subscribe, respectively. For example, the heart monitoring portion of the SCAAL application can publish changes to the environment monitor topic when it adjusts its data sending rate based on requests from health workers subscribing to the data. Likewise, data subscribers can query the environment monitor topic for the periodic sending rate of the data and then calculate the percent loss in the network by dividing the expected number of data updates for a given period with the actual number of updates received.

Figure 84 shows the data described in environment monitor topic. The data is described in the platform-independent interface definition language (IDL) as defined by the

```
struct QosMonitoring {
  long receiver_count;
  long percent_network_loss;
  long send_rate_in_Hz;
  string cpu_speed;
  string ram;
  string network_speed;
  string dds_impl;
  string composite_metric;
};
```

Figure 84: Environment Monitor Topic

OMG. Our prototype is interested in the following aspects of the environment information since the data values for these aspects are used to determine the most appropriate transport protocol:

• receiver_count: the number of receivers currently receiving application data (*e.g.*, 5 => 5 receivers receiving application data).

• percent_network_loss: the percent packet loss in the network (*e.g.*, 3 => 3% loss of packets in the network).

• send_rate_in_Hz: the data sending rate for the heart monitoring data in Hz (*e.g.*, 50 => sending rate of 50 Hz).

• cpu_speed: the speed of the CPU being used in MHz (*e.g.*, "2992.883" => CPU speed of 2.992883 GHz). For clarity and simplicity, the ADAMANT prototype assumes common CPU speeds for all machines used.

• RAM: the amount of random access memory available on the machines being used in bytes (*e.g.*, "2062172" => 2 GB of RAM). Again, for clarity and simplicity, the ADAMANT prototype assumes common amount of RAM for all machines used.

• network_speed: the speed of the network being used in Mb/sec (*e.g.*, "1000" => 1 Gb/sec network).

• dds_impl: the DDS implementation being used (*e.g.*, "OpenSplice" indicates the use of PrismTech's OpenSplice DDS implementation). For simplicity as a proof of concept,

the ADAMANT prototype only currently supports the OpenSplice DDS implementation, though support for the OpenDDS or RTI DDS implementations can easily be added.

• composite_metric: the composite metric that is of interest to the application (*e.g.*, "Re-Late2"). The ReLate2 family of composite metrics quantitatively evaluates multiple QoS properties. For example, the ReLate2 metric combines data reliability and latency to produce a single value used for objective comparison. Other composite metrics include ReLate2Jit that quantitatively evaluates data reliability, latency, and jitter; ReLate2Net that evaluates reliability, latency, and network bandwidth usage; and ReLate2Burst that evaluates reliability, latency, and network data burstiness [48]. ADAMANT supports all these composite metrics while easing the incorporation of additional composite metrics.

Once updates have been made to the environment monitor topic, the autonomic controller receives the updated environment configuration (outlined as Step 2 in Figure 82). The autonomic controller then compares the new and previous environment configurations. If the configurations are different the controller invokes TIML to determine which transport protocol and parameter values best support the desired QoS. If the configurations do not differ the autonomic controller simply returns since no adaptation is needed.

Step 3 in Figure 82 shows how TIML receives the new environment configuration and determines if the configuration is one on which the machine learning techniques have been trained. If the machine learning techniques have previously been trained offline using the configuration TIML uses an ANN to determine the appropriate transport protocol and parameter settings. Since we overfitted the ANN to the training data, the ANN will produce 100% accurate determinations for these known environment configurations.

If machine learning techniques have not previously been trained on an environment configuration, however, TIML uses an SVM to determine the appropriate transport protocol and parameter settings. Our work on evaluating machine learning techniques in Section IV.2.3.2 shows how an SVM will provide higher accuracy for determining the appropriate protocol and parameters than an ANN when the input environment configuration

was not used during offline training (*i.e.*, unknown until runtime). The overall accuracy of ADAMANT is enhanced by combining the 100% accuracy of an overfitted ANN for environment configurations known *a priori* with the higher accuracy of an SVM for environment configurations unknown until runtime. In particular, we see an increase in accuracy of 8.6% combining both an ANN and an SVM, compared to only using an ANN (*i.e.*, 77.69% average ANN accuracy for environments unknown until runtime compared to 86.29% for the SVM = 8.6% increase).

Both ANNs and SVM provide constant-time complexity for determining protocols and parameters. The mechanism used to determine if the environment configuration have been known *a priori* must therefore also provide constant-time complexity to maintain this time complexity for the entire protocol optimization process. TIML utilizes perfect hashing for the environment configurations to determine in constant time whether or not an environment configuration is known *a priori* (*i.e.*, used for training) or unknown until runtime. TIML provides the environment configurations on which the ANN has been trained as keys to the perfect hashing to map to the corresponding scaled environment configuration data. If the key is found via the perfect hash then TIML knows that the environment configuration has been seen before in offline training and uses the ANN since it will provide perfect accuracy. If the key is not mapped, then TIML will use the SVM since it provides the highest accuracy for environment configurations that are unknown until runtime.

Once the appropriate transport protocol has been decided, TIML returns this result to the autonomic controller (Step 4 in Figure 82). The controller then compares the recommended transport protocol and protocol parameters with the current transport protocol and protocol parameters. If there is no difference, the controller need not take any further action. If there are differences between the current protocol and the recommended protocol, the controller passes the new protocol settings to ANT to make the needed adaptation.

Our ADAMANT prototype uses the OpenSplice DDS implementation, which uses a networking daemon on each machine to send and receive data across machine boundaries.

The ANT framework resides in the networking daemon since the ANT protocols are used to disseminate the application data across the network. The autonomic controller resides in the application executable since it needs to respond to updates in the environment as facilitated by the environment monitor topic. For a single computer platform, OpenSplice uses shared memory to communicate between the SCAAL application executable and the OpenSplice daemon. Since the daemon runs as a separate process from the application executable, some form of interprocess communication (IPC) is needed to have the controller inform ANT of the needed protocol changes.

The form of IPC used when communicating between the autonomic controller and ANT can vary depending upon the needs of the application and the IPC mechanisms supported by the operating system. In our ADAMANT prototype the autonomic controller residing in the application executable sends a signal to ANT residing in the networking daemon. The OpenSplice networking daemon is enhanced to include a signal handler. In particular, when the controller determines the transport protocol must be modified it sends a SIGHUP signal to the networking daemon. When the networking daemon processes the SIGHUP signal, the daemon invokes ANT to reconfigure. ADAMANT utilizes the Component Configurator pattern [92] for ANT to reconfigure itself by constructing the appropriate configuration file and then signaling ANT to reconfigure.

The need for IPC depends upon the DDS implementation. For example, rather than using a network daemon, the OpenDDS DDS implementation supports direct point-to-point network connectivity between application executables residing on different machines. For ADAMANT using OpenDDS, intraprocess communication would be needed rather than IPC. ADAMANT would set a variable accessible across threads using appropriate locking mechanisms. ANT would then wait until the variable was set (*e.g.*, using a condition variable) and reconfigure the transport protocol as needed.

After ANT receives the signal to reconfigure (Step 5 in Figure 82) it determines whether to modify an existing transport protocol or switch to a new protocol. ANT keeps track of

the current transport protocol being used for comparison. If the current protocol must be modified then ANT invokes the appropriate methods on the relevant protocol modules to change the protocol parameters. If a new protocol must be used ANT first disables the existing protocol and enables the new protocol.

The modules in the ANT framework use pub/sub communication to consume and supply events of interest. This approach allows for flexibility in the way modules are connected together to create the functionality needed for a particular transport protocol. This approach also allows the enabling/disabling of transport protocols simply by registering and unregistering for particular events. ANT thus unregisters event interest for the modules involved with the old protocol to disable the old protocol and registers event interest for the modules involved with the new protocol to enable the new protocol.

IV.2.4.3 Addressing Challenges of SCAAL Applications

This section describes how ADAMANT addresses the challenges of SCAAL applications presented in Section IV.2.2.

Addressing Challenge 1: Managing interacting QoS requirements. ADAMANT addresses the challenge of managing interacting QoS requirements by using the transport protocols provided by the ANT framework. ANT supports transport protocols that address interacting QoS requirements. In particular, it provides the NAKcast and Ricochet transport protocols that balance the contentious QoS requirements of data reliability and low latency. As shown in previous work [45], these protocols ameliorate the loss of network data packets while imposing low latency overhead. In particular, the NAKcast protocol uses negative acknowledgments (a.k.a. NAKs) that the receiver sends to the sender for notification of lost data packets. NAKcast provides a tunable timeout parameter to determine when NAKs should be sent. The Ricochet protocol supports error correction information that the receivers send to each other to recover from lost data packets. Ricochet provides a tunable parameter to determine how many data packets need to be received before error

correction is sent out. Ricochet also provides a tunable parameter to determine how many other receivers receive the error correction information from a single receiver.

Addressing Challenge 2: Accurate adaptation. ADAMANT addresses the challenge of accurate adaptation in several ways. First, it leverages the use of DDS to provide the infrastructure to disseminate the environment monitoring information needed to determine an accurate adaptation. Second, it uses TIML to provide an integration of multiple supervised machine learning techniques to provide high accuracy for both operating environments known *a priori* and operating environments unknown until runtime. TIML supports accurate adaptation guidance in dynamic environments by using the most accurate machine learning technique for operating environments known *a priori* (*i.e.*, ANNs) integrated with the most accurate technique for operating environments unknown until runtime (*i.e.*, SVMs). Third, ADAMANT's autonomic controller ensures accuracy by managing the adaptation process of receiving environment updates, delegating this information to TIML to provide guidance, and passing the recommended transitions to ANT.

Addressing Challenge 3: Timely adaptation. ADAMANT addresses the challenge of timely adaptation in several ways. First, it uses DDS to disseminate the environment monitoring information needed to determine an accurate adaptation. Second, since the monitoring information is realized as a DDS topic, the DDS QoS policies can be applied to the topic and the applicable entities involved with the topic (*e.g.*, data readers, data writers). For example, the transport priority QoS policy can be applied to the environment monitoring data to ensure the environment updates have priority over other data on the network.

ADAMANT supports constant-time runtime transition and reconfiguration of transport protocols used as the QoS mechanisms to provide needed QoS, as discussed in Section IV.2.5. In particular, TIML utilizes an ANN to provide adaptation guidance in constant time for operating environments known *a priori*. TIML uses an SVM to guide adaptation in constant time for operating environments unknown until runtime. Moreover, TIML uses

constant-time perfect hashing to integrate the machine learning techniques and determine the appropriate technique to use.

Addressing Challenge 4: Reducing development complexity. ADAMANT addresses the challenge of reducing development complexity by using machine learning techniques that manage the inherent complexity of providing the appropriate transport protocol recommendation for a given operating environment. The machine learning techniques can also be used directly in the ADAMANT implementation. These techniques thus reduce development complexity by eliminating the accidental complexity of transforming the mapping of environments to protocols from design to implementation [48]. Moreover, ADAMANT provides an environment monitoring topic that disseminates and handles the environment information updates relevant to adapting the QoS mechanisms of transport protocols.

IV.2.5 Experimental Results and Analysis

This section describes the setup, design, and analysis of results from experiments we conducted to identify the need for autonomic adaptation of transport protocols and evaluate the timeliness of the adaptations in dynamic environments representative of the SCAAL applications presented in Section IV.2.2. These results quantify (1) the effect of changes in the operating environment on the QoS provided by ADAMANT as measured by the composite QoS metrics defined below, (2) the timeliness of TIML's determination of an appropriate transport protocol, and (3) the timeliness of ADAMANT's adaptation of transport protocols via the ANT framework.

IV.2.5.1 Experimental Setup

We conducted our experiments using the Emulab testbed (www.emulab.net) at the University of Utah. Emulab allows the configuration of various types of computing and networking platforms. For our experiments highlighting the need for adaptation, we held the

computing and networking platform constant (*i.e.*, 3 GHz CPU, 1 Gbps LAN). We used the Redhat Fedora Core release 6 OS with realtime patches across all the computing nodes.

The points of variability for the experiments were indicative of dynamic environments. In particular, we varied the number of data receivers, the percent loss in the network, and the data sending rate as outlined in Section IV.2.4.2. By adjusting these variables we were able to highlight scenarios where changes in the environment mandated changes to the transport protocols being used to provide the highest level of QoS for the multiple QoS properties involved.

IV.2.5.2 Composite QoS Metrics for Reliability and Timeliness

Our previous work on QoS-enabled pub/sub middleware performance [45, 48] showed that some transport protocols provide better reliability (as measured by the number of network packets received divided by the number sent) and latency for particular environments while other protocols are better for other environments. We therefore developed several composite QoS metrics to quantitatively evaluate multiple QoS aspects simultaneously. These composite metrics provide a uniform and objective evaluation of ADAMANT in dynamic environments. Our family of composite metrics are based on the QoS concerns of reliability and average latency and optionally include the QoS aspects of (1) jitter (*i.e.*, standard deviation of the latency of network packets), (2) network bandwidth usage, and (3) burstiness (*i.e.*, the standard deviation of average bandwidth usage per second of time).

In particular, we defined the ReLate2 family of composite QoS metrics. The ReLate2 metric is defined by the product of the average data packet latency and the percent loss that the transport protocol provides + 1 (to account for 0% loss) which implies an order of magnitude increase for 9% loss. Based on previous research [7, 8, 78], this adjustment is relevant for multimedia data such as the high-resolution 3-D health data in our SCAAL example. For example, if for a given protocol the average packet latency is 1,000 μs and the percent loss is 0 (*i.e.*, no packets lost) then the ReLate2 value is 1,000. Having 9% and

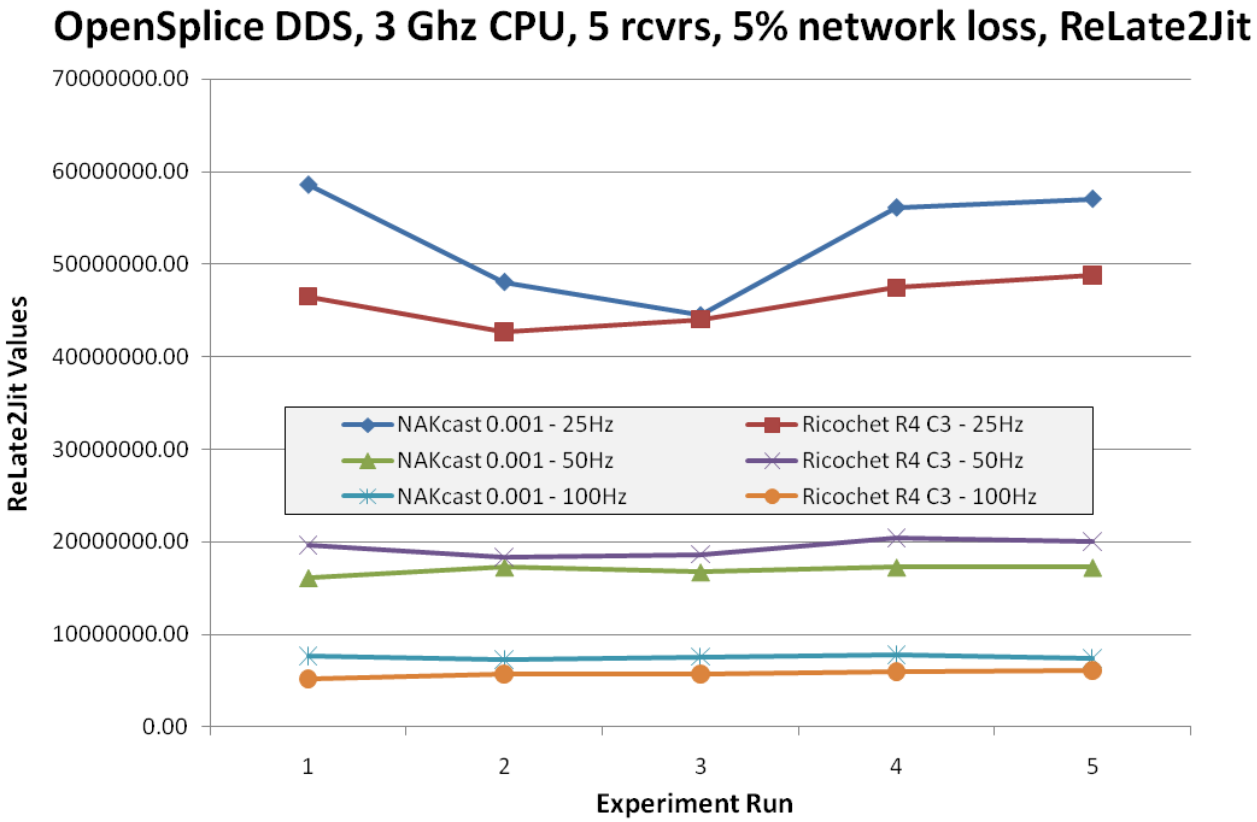

Figure 85: Effect of Changing Data Sending Rate

19% loss with the same average latency produces the ReLate2 values of 10,000 and 20,000, respectively. ReLate2Jit is a product of the ReLate2 value and the jitter of the data packets to quantify the multiple QoS concerns of jitter, reliability, and average latency.

IV.2.5.3 Experiments Highlighting Need for Autonomic Adaptation

We now present the results of experiments for autonomic adaptation of the QoS mechanisms of transport protocols. We apply the composite metrics defined in the previous section to several different operating environments to highlight how differences in the environment trigger differences in the transport protocols used to support QoS. Figure 85 shows a change in the sending rate corresponds to a change in the protocol that provides the best QoS.

In particular, for an operating environment using the OpenSplice DDS implementation,

machines with 3 GHz CPUs, 5 data receivers, and 5% network packet loss, we see that for a data sending rate of 25Hz, the NAKcast protocol (with a timeout parameter to determine NAK transmissions of 0.001 seconds) performs better (*i.e.*, has lower ReLate2Jit values) than Ricochet (with an *R* value of 4 and a *C* value of 3).

Ricochet's R value determines how many data packets are received before error correction data is sent (*e.g.*, 4 packets received before one error correction packet is sent) and Ricochet's C value determines how many other receivers this receiver sends error correct data (*e.g.*, 3 receivers receive error correction data from any one receiver). When the sending rate is changed to 50Hz, however, Ricochet performs better. Finally, when the sending rate is further increased to 100Hz NAKcast again performs better (*i.e.*, has lower ReLate2Jit values).

IV.2.5.4 Timeliness of TIML

We next describe the timeliness of TIML as it decides the most appropriate transport protocol for a given environment configuration. As described in Challenge 2 (timely adaptation) in Section IV.2.2, the personal datacenter for the SCAAL application needs to have timely adaptations. We now provide timing information based on the responsiveness of TIML when queried for an optimal transport protocol. We used the Emulab configuration as described in Section IV.2.5.1. A high resolution timestamp was taken right before and right after each call was made to TIML.

TIML combines and integrates the use of ANN and SVM machine learning techniques. These techniques present different response times (although the times for each technique remain constant). We therefore conducted experiments with operating environment configurations that would use the ANN (*i.e.*, the configurations that were known *a priori*) and configurations that would use the SVM (*i.e.*, the configurations that were unknown until runtime). Since these techniques provide constant-time performance, their compute times

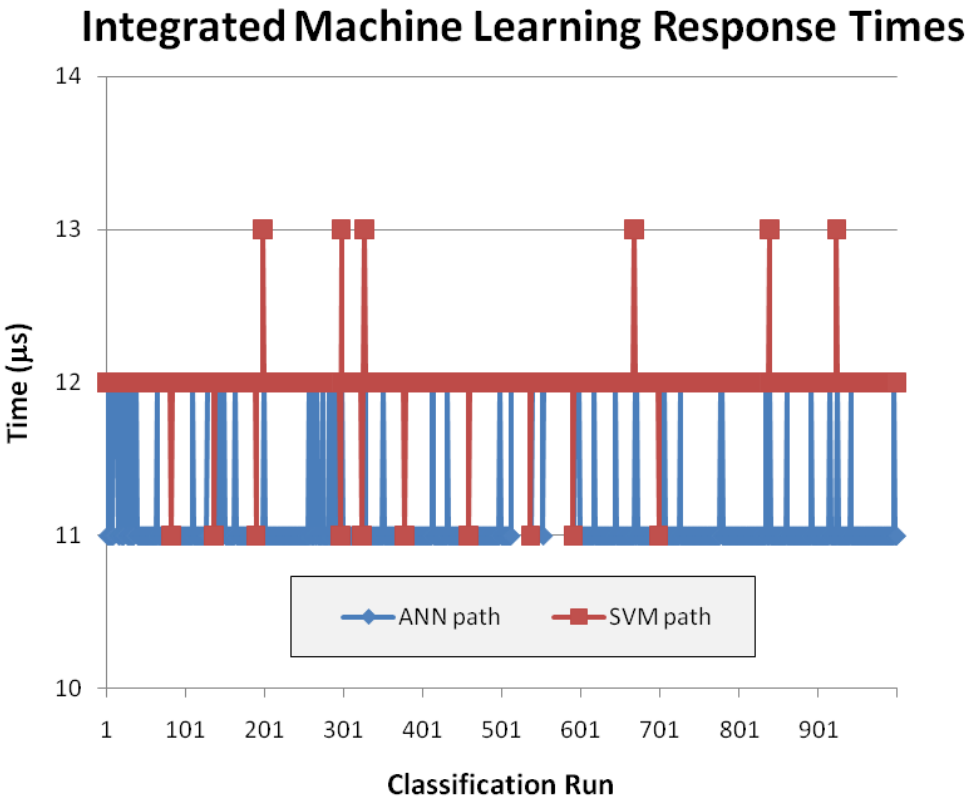

Figure 86: Integrated Supervised Machine Learning Response Times

are invariant to the specific environment configuration, so we did not run timing test for all different environment configurations.

Figure 86 presents the response times for TIML in ADAMANT for 1,000 iterations when TIML selects and uses either an ANN or the SVM. The figure highlights the times used within the integrated machine learning techniques when the environment configuration is (1) known *a priori* and thus triggers the use of an ANN and (2) unknown until runtime triggering the use of an SVM. On average, TIML, when using the ANN, presents the lower response time of 11.161 μs while TIML using the SVM presents an average response time of 11.996 μs. The bound on TIML is then the maximum between the two (*i.e.*, 11.996 μs). The figure also appears to show that TIML using the ANN has more jitter than TIML using the SVM. The jitter is within the resolution of the timers (*i.e.*, 1 μs) used for

collecting the times, however, since the times only vary by +/- 1 μs from the median values (*i.e.*, 11 μs for the TIML when the ANN is used and 12 μs when the SVM is used).

IV.2.5.5 Timeliness of ANT Reconfiguration

We now describe the experiments we conducted to show the timeliness of the ANT framework as it transitions from one transport protocol to another. As described in Challenge 2 (timely adaptation)in Section IV.2.2, the personal datacenter for the SCAAL application needs to have timely adaptations. In the previous section we presented timing results for determining the appropriate transport protocol. In this section we provide timing information on the reconfiguration of transport protocols supported in the ANT framework portion of ADAMANT. We used the same experimental environment as described in Section IV.2.5.1. A high resolution timestamp was taken right before and right after each call made to ANT to reconfigure transport protocols.

Figure 87 shows the times taken for transport protocol reconfiguration across 1,000 iterations. The figure includes times for three different scenarios. Two of the scenarios are most relevant for the transport protocols that best handle reliability and latency (*i.e.*, the NAKcast and Ricochet protocols). The third scenario presents a baseline when checks are performed to determine if a protocol transition is needed but no transition is needed.

The baseline times for no reconfiguration shows 0 μs taken to determine that no protocol reconfiguration is needed. Obviously, some time is taken to make the determination that no reconfiguration is needed but this time is smaller than the resolution of the timestamps (*i.e.*, < 1 μs). These times provide an idea of the overhead required in making any protocol reconfiguration.

The remaining two scenarios are when (1) the NAKcast protocol is running and a transition is made to the Ricochet protocol and (2) when the Ricochet protocol is running and a transition is made to the NAKcast protocol. The times for these transitions should be constant since, when reconfiguring, ANT registers a constant number of events and event

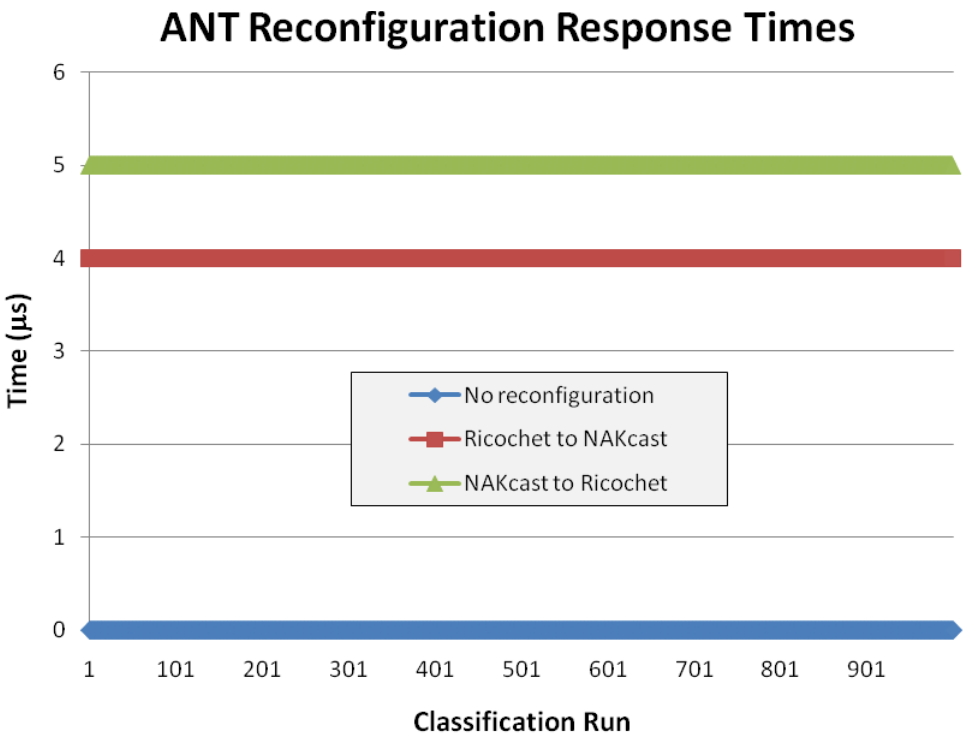

Figure 87: Transport Protocol Reconfiguration Times within ANT

handlers for the new protocol and unregisters a constant number of events and event handlers for the old protocol. The number of event and event handlers is known *a priori* at development time. Registering and unregistering events and event handlers correspond to inserting and removing items from a queue which are constant time operations.

In particular, for the NAKcast and Ricochet protocols, we know *a priori* the number and kinds of events and event handlers that each protocol uses. ADAMANT first unregisters all the relevant events and event handlers for an old protocol and then registers all the relevant events and event handlers for the new protocol. Since ADAMANT controls how and in what order events and event handlers are registered and unregistered in ANT, managing the lists for these events and event handlers can be done in constant time. The Ricochet to NAKcast transition consistently takes 4 μs while the NAKcast to Ricochet transition consistently takes 5 μs. For the ADAMANT prototype using the OpenSplice DDS implementation, these transitions are happening within the single network daemon

per computing platform. As noted in Section IV.2.4.2, ANT's transitions are deterministic with the DDS middleware ensuring that all the computing platform see the same updates and therefore make the same transitions. These empirical transition times verify that ANT protocol transitions are made in a constant amount of time.

IV.2.5.6 Summary of Results

The results of experiments presented in this section show that there are scenarios where a change in the operating environment requires a change in the QoS mechanisms (*e.g.*, transport protocols) that ADAMANT is utilizing. Based on this information, the experiments show that ADAMANT delivers constant-time decision making regarding the appropriate the transport protocol to use as well as constant-time transitioning from one transport protocol to another. For QoS-enabled DRE pub/sub applications ADAMANT provides the constant-time complexity needed for detecting environment changes, determining the appropriate course of action, and executing that plan.

IV.2.6 Lessons Learned

Developers of systems that utilize DRE pub/sub middleware face a number of challenges when developing and deploying their systems in dynamic environments. To address these challenges, we have developed ADAMANT to integrate and enhance (1) QoS-enabled pub/sub middleware, (2) an environment monitoring topic, (3) a flexible transport protocols, (4) a novel integration of supervised machine learning techniques, and (5) an autonomic controller to provide fast and predictable reconfiguration of middleware and transport protocols for enterprise DRE pub/sub systems. This section empirically shows how ADAMANT can autonomically adapt to changing conditions in operating environments to support QoS in a fast, constant-time, and accurate manner.

The following is a summary of lessons learned from our experience evaluating ADAMANT's autonomic adaptation performance in various operating environments:

• **Several trade-offs exist when using machine learning in dynamic environments.** There are several trade-offs between having machine learning that (1) is completely accurate for environments known at training time, (2) highly accurate for environments unknown until runtime, (3) can accommodate new data on which to train as the system is running, and (4) can expend the appropriate amount of time interactively training machine learning tools while the system is running. Since overfitting an ANN to environment configurations known *a priori* provides perfect accuracy and low response times, it is preferable to incorporate new operating environment configurations unknown until runtime into the ANN training set while the system is running. A low-priority thread could be used to constantly retrain the ANN and swap in the updated ANN at appropriate times. While this approach would incorporate new environment configurations, our future work will address trade-offs between when to migrate to using the updated ANN and how to determine the priority of the low-priority training thread so that it will not be starved.

• **Preparing environment information for use in machine learning tools is time consuming and tedious.** Data should be scaled since scaling the data typically produces the best results. The scaling factors used on the data for training the machine learning tools should be applied to the data collected from the environment during runtime.

• **To increase accuracy of determining appropriate responses to changes in the operating environment, multiple machine learning approaches can be integrated to handle configurations known *a priori* and environment configurations not known until runtime.** Some machine learning techniques provide higher accuracy than others for operating environments known *a priori*. In particular, ANNs can be overfitted to the data to provide 100% accuracy for these kinds of environments. Other techniques provide higher accuracy for environments unknown until runtime. An integration of multiple machine learning techniques can provide higher overall accuracy than can be provided by any single machine learning technique.

If timeliness is a concern, then when integrating multiple techniques, care must be

taken to ensure that the integration itself does not change the time complexity characteristics. ADAMANT incorporates TIML to increase its overall accuracy for both operating environments known *a priori* and environments unknown until runtime while also ensuring that the integration itself maintains the constant-time complexity needed by DRE systems.

• **Transport protocols need to be selectively used based on the QoS specified.** While several DDS implementations provide pluggable transport frameworks to leverage standard and custom transport protocols (*e.g.*, OpenDDS and OpenSplice), the properties of the transport protocols need to be dictated by the QoS specified by the application. For example, in our work we would like to specify that the environment monitoring topic information be sent and received reliably. The DDS implementations, however, provide no infrastructure for mapping between the transport protocols (*e.g.*, Ricochet, NAKcast) used and the QoS properties specified (*e.g.*, reliable data communication, best-effort).

We are researching the development of a transport protocol taxonomy that a QoS-enabled middleware would be able to leverage to determine which protocol to use based on QoS specified at the application level using the DDS QoS policies. The properties that transport protocols provide can be used to classify the protocols with respect to QoS. The middleware can then select the most appropriate transport protocol based on the QoS properties needed. The middleware can also select different transport protocols for different QoS properties.

• **QoS-enabled middleware provides a fairly coarse-grained approach to reliability.** Utilizing transport protocols such as Ricochet and NAKcast allows QoS-enabled middleware to provide finer-grained reliability as well as considering latency. However, reliability is typically only supported as the dichotomy of best-effort or reliable with no consideration of highly probabilistic reliability. Moreover, the semantics of combining multiple QoS aspects (*e.g.*, reliability and latency) are not clearly defined at the middleware level. Transport protocols such as Ricochet and NAKcast capture the finer-grained reliability property of high probability of reliability but not perfect reliability

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

IV.3 Related Research

This section compares our work on autonomic adaptation of QoS-enabled DRE pub/sub middleware in flexible and dynamic environments with related R&D efforts.

Specialized embedded middleware. Bellavista *et al.* [13] present their work on embedded middleware 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. In this sense, MUM presents adaptation functionality to an application as it moves through its environments. In contrast, our work on ADAMANT focuses on adaptively configuring embedded middleware based on the QoS of the application and the resources presented in the environment.

Boonma *et al.* [15] have developed a DDS implementation called TinyDDS which is 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, TinyDDS provides no support for adaptively configuring itself given different resources provided by the environment as is the case with ADAMANT.

Support for adaptive middleware. The Mobility Support Service (MSS) [22] 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. Unlike ADAMANT, however, MSS does not configure itself with respect to the available computing resources.

Ostermann *et al.* [83] present the ASKALON middleware for cloud environments that is based on middleware for grid workflow application development but enhanced to leverage clouds. ASKALON provides an infrastructure that allows the execution of workflows on conventional grid resources but that can adapt on-demand to supplement these resources with additional cloud resources as needed. Whenever the ASKALON task scheduler determines that existing grid computing resources are fully utilized, the scheduler can optionally include cloud resources to enhance the computing capability to execute a workflow more quickly. In contrast to ADAMANT, however, ASKALON does not address the adaptive configurability needs of enterprise DRE systems in elastic clouds.

Gridkit [37] 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). In contrast to ADAMANT, however, Gridkit does not address timely adaptation, nor does it focus on discovering and leveraging the elastic provisioning of cloud resources.

David and Ledoux have developed SAFRAN [28] 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 point-cuts, and (4) adaptation which uses adaptation policies to link join points to advices.

The SAFRAN component framework, however, provides only development support for maintaining specified QoS. The adaptive policies and component implementation are the responsibility of the application developer. Moreover, SAFRAN does not address timely configuration of components across the elastic resources of cloud computing. In contrast, ADAMANT provides a middleware implementation that adapts to the cloud resources presented to it.

Machine learning in support of autonomic adaptation. Vienne and Sourrouille [103] 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.

In contrast to ADAMANT, however, DCBL focuses only on a single computer, rather than scalable DRE pub/sub systems. Moreover, reinforcement learning used by DCBL can have non-constant and even unbounded time complexities unlike ADAMANT which provides fast and predictable decision making. DCBL also requires developers to specify in an XML file the selection of operating modes given a QoS level along with execution paths, which increases accidental development complexity whereas ADAMANT ameliorates this

complexity by having an artificial neural network manage the appropriate operating modes for a given environment.

RAC [19] uses reinforcement learning for the configuration of Web services. RAC autonomically configures services via performance parameter settings to change the services' workload and also to change the virtual machine configurations. The reinforcement learning component of RAC is enhanced with an additional runtime initialization period at system startup.

Reinforcement learning explores the possible solution space to determine generalized solutions of the negative and positive reinforcements given. Due to RAC's use of reinforcement learning, its determination of an appropriate response is unbounded due to online exploration of the solution space and modification of decisions while the system is running. In contrast, ADAMANT uses multiple supervised machine learning techniques to provide fast, predictable complexity decision making and high accuracy.

Tock *et al* [100] utilize machine learning for data dissemination in their work on Multicast Mapping (MCM). MCM hierarchically clusters data flows so that multiple topics map to a single session and multiple sessions are mapped to a single reliable multicast group. MCM manages the scarce availability of multicast addresses in large-scale systems and uses machine learning for adaptation as user interest and message rates change during the day. MCM is designed only to address the scarce resource of IP multicast addresses in large-scale systems, however, rather than timely adaptation based on available resources as done with ADAMANT.

Infrastructure for autonomic computing. Grace *et al.* [38] 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 provide an adaptation implementation as ADAMANT does.

Valetto *et al.* [102] have 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 provide an implementation unlike ADAMANT.

Autonomic adaption of service level agreements. Herssens *et al.* [42] present work centering around autonomically adapting service level agreements (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. However, this work does not address the timeliness concerns that are addressed in ADAMANT.

Autonomic adaption of networks. The Autonomic Real-time Multicast Distribution System (ARMDS) [18] 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 that form the multicast graph to enhance the consistency of service delivery. 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. While ARMDS provides timely adaptation its focus is at the level of the network itself rather than on the higher level abstractions presented by QoS-enabled pub/sub middleware.

CHAPTER V

CONCLUDING REMARKS

In this thesis we initially presented the *Distributed QoS Modeling Language (DQML)*, which is a 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.

We 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 utilizes the *ReLate2* family of composite QoS metrics we developed to evaluate multiple QoS concerns in providing empirical evaluations and analysis.

Finally, we presented ADAPtive Middleware And Network Transports (ADAMANT), which is a software platform to support QoS of pub/sub DRE systems in flexible and 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 DRE systems operating in flexible and dynamic environments.

Table 15 presents the summary of research contributions and is followed by a list related research publications.

Table 15: Summary of Research Contributions

Category	**Contributions**
Correct QoS Design	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.
Quantitative Evaluation of Multiple QoS Concerns	ReLate2 Metrics: a family of metrics that combines multiple QoS concerns to provide quantitative evaluation of (1) reliability, (2) average latency, (3) jitter, (4) network bandwidth usage, and (5) network packet burstiness.
Evaluation of QoS Mechanisms	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.
Autonomic Adaptation for Qos	ADAMANT: autonomic configuration of QoS-enabled pub/sub middleware to support QoS in flexible computing environments (*e.g.*, cloud computing infrastructure) and autonomic adaptation to support QoS in dynamic operating environments. ADAMANT provides this support 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.

Summary of Publications and Presentations

Research on DQML, FLEXMAT, ReLate2 metrics, and ADAMANT has led to the following referred journal, conference, book chapter, and workshop publications.

Journal Publications

1. Joe Hoffert, Daniel Mack, & Douglas Schmidt, *Integrating Machine Learning Techniques to Adapt Protocols for QoS-enabled Distributed Real-time and Embedded Publish/Subscribe Middleware*, International Journal of Network Protocols and Algorithms (NPA): Special Issue on Data Dissemination for Large-scale Complex Critical Infrastructures, Vol. 2, No. 3 (2010)

Conference Publications

1. Joe Hoffert & Douglas Schmidt, *Adapting Distributed Real-time and Embedded Publish/Subscribe Middleware for Cloud-Computing Environments*, ACM/IFIP/USENIX 11th International Middleware Conference (Middleware 2010), Bangalore, India, November 2010

2. Joe Hoffert & Douglas Schmidt, *Evaluating Supervised Machine Learning for Adapting Enterprise DRE Systems*, 2010 International Symposium on Intelligence Information Processing and Trusted Computing (IPTC 2010), Huanggang, China, October 2010

3. Joseph W. Hoffert, Douglas Schmidt, & Aniruddha Gokhale, *Evaluating Transport Protocols for Real-time Event Stream Processing Middleware and Applications*, The 11th International Symposium on Distributed Objects, Middleware, and Applications (DOA '09), Algarve, Portugal, November 2009

4. Joseph W. Hoffert & Douglas Schmidt, *Maintaining QoS for Publish/Subscribe Middleware in Dynamic Environments*, The 3rd ACM International Conference on Distributed Event-Based Systems (DEBS '09), Nashville, TN, USA, July 2009

5. Joseph W. Hoffert, Douglas Schmidt, & Aniruddha Gokhale, *DQML: A Modeling Language for Configuring Distributed Publish/Subscribe Quality of Service Policies*, The 10th International Symposium on Distributed Objects, Middleware, and Applications (DOA '08), Monterrey, Mexico, November 2008

6. Joseph W. Hoffert, Douglas Schmidt, & Aniruddha Gokhale, *A QoS Policy Configuration Modeling Language for Publish/Subscribe Middleware Platforms*, The Inaugural International Conference on Distributed Event-Based Systems, Toronto, Ontario, Canada, June 20-22, 2007.

7. Joseph W. Hoffert, Shanshan Jiang, & Douglas C. Schmidt, *A Taxonomy of Discovery Services and Gap Analysis for Ultra-Large Scale Systems*, 2007 ACM Southeast Conference, Winston-Salem, North Carolina, USA, March 23 - 24, 2007.

Book Chapters

1. Joe Hoffert, Douglas Schmidt, & Aniruddha Gokhale, *Productivity Analysis of the Distributed QoS Modeling Language*, in Model-Driven Domain Analysis & Software Development: Architectures & Functions. Ed. Dr. Janis Osis & Dr. Erika Asnina, Riga Technical University, Latvia.

Workshop Publications

1. Joe Hoffert, Douglas Schmidt, & Annirudha Gokhale, *Adapting and Evaluating Distributed Real-time and Embedded Systems in Dynamic Environments*, The 1st International Workshop on Data Dissemination for Large scale Complex Critical Infrastructures (DD4LCCI 2010), Valencia, Spain, April 2010

2. Joseph W. Hoffert, Daniel Mack, & Douglas Schmidt, *Using Machine Learning to Maintain Pub/Sub System QoS in Dynamic Environments*, The 8th Workshop on Adaptive and Reflective Middleware (ARM) 2009, Urbana Champaign, IL, December 2009

3. Joseph W. Hoffert & Douglas Schmidt, *FLEXible Middleware And Transports (FLEXMAT) for Real-time Event Stream Processing (RT-ESP) Applications*, Workshop on Distributed Object Computing for Real-time and Embedded Systems (OMG RTWS '09), Washington, D.C., USA, July 13-15, 2009.

4. Joe Hoffert, Mahesh Balakrishnan, Douglas Schmidt, & Ken Birman, *Supporting Large-scale Continuous Stream Datacenters via Pub/Sub Middleware and Adaptive Transport Protocols*, The 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS 2008), Yorktown, New York, USA, September 15-17, 2008.

5. Mahesh Balakrishnan, Joe Hoffert, Ken Birman, & Douglas Schmidt, *Rethinking Reliable Transport for the Datacenter*, The 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS 2008), Yorktown, New York, USA, September 15-17, 2008.

6. Joe Hoffert, Mahesh Balakrishnan, Doug Schmidt, & Ken Birman, *Supporting Scalability and Adaptability via ADAptive Middleware And Network Transports (ADAMANT)*, Workshop on Distributed Object Computing for Real-time and Embedded Systems (OMG RTWS '08), Washington, D.C., USA, July 14-16, 2008.

Submitted Papers

1. Joe Hoffert, Douglas Schmidt, & Aniruddha Gokhale, *Evaluating Timeliness and Accuracy Trade-offs of Supervised Machine Learning for Adapting Enterprise DRE Systems in Dynamic Environments*, International Journal of Computational Intelligence Systems.

2. Joe Hoffert, Aniruddha Gokhale, & Douglas Schmidt, *Autonomic Adaptation of Publish/Subscribe Middleware in Dynamic Environments*, International Journal of Adaptive, Resilient and Autonomic Systems.

REFERENCES

[1] Global Information Grid. The National Security Agency, www.nsa.gov/ia/industry/gig.cfm?MenuID=10.3.2.2.

[2] Net-Centric Enterprise Services. Defense Information Systems Agency, http://www.disa.mil/nces/.

[3] Silvia Abrahão and Geert Poels. Experimental evaluation of an object-oriented function point measurement procedure. *Inf. Softw. Technol.*, 49:366–380, April 2007.

[4] Silvia Abrahão and Geert Poels. A family of experiments to evaluate a functional size measurement procedure for web applications. *J. Syst. Softw.*, 82:253–269, February 2009.

[5] D. Agrawal, Kang-Won Lee, and J. Lobo. Policy-based management of networked computing systems. *Communications Magazine, IEEE*, 43(10):69 – 75, October 2005.

[6] Colin Atkinson and Thomas Kuhne. Model-driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[7] Yan Bai and Mabo R. Ito. A study for providing better quality of service to voip users. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, pages 799–804, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Yan Bai and M.R. Ito. A new technique for minimizing network loss from users' perspective. *Journal of Network Computing Applications*, 30(2):637–649, 2007.

[9] Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleisch. Ricochet: Lateral Error Correction for Time-Critical Multicast. In *NSDI 2007: Fourth Usenix Symposium on Networked Systems Design and Implementation*, Boston, MA, 2007.

[10] Mahesh Balakrishnan, Stefan Pleisch, and Ken Birman. Slingshot: Time-Critical Multicast for Clustered Applications. In *The IEEE Conference on Network Computing and Applications*, 2005.

[11] Krishnakumar Balasubramanian, Douglas C. Schmidt, Zoltan Molnar, and Akos Ledeczi. Component-based system integration via (meta)model composition. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 93–102, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Martin Bateman, Saleem Bhatti, Greg Bigwood, Devan Rehunathan, Colin Allison,

Tristan Henderson, and Dimitrios Miras. A comparison of tcp behaviour at high speeds using ns-2 and linux. In *CNS '08: Proceedings of the 11th communications and networking simulation symposium*, pages 30–37, New York, NY, USA, 2008. ACM.

[13] Paolo Bellavista, Antonio Corradi, and Luca Foschini. Context-aware handoff middleware for transparent service continuity in wireless networks. *Pervasive and Mobile Computing*, 3(4):439 – 466, 2007. Middleware for Pervasive Computing.

[14] B.W. Boehm. Improving software productivity. *Computer*, 20(9):43–57, Sept. 1987.

[15] Pruet Boonma and Junichi Suzuki. Middleware support for pluggable non-functional properties in wireless sensor networks. *SERVICES '08: Proceedings of the 2008 IEEE Congress on Services - Part I*, pages 360–367, July 2008.

[16] L. Breiman, J. Freidman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth, Monterey, CA, 1984.

[17] Andrej Brodnik and J. Ian Munro. *Algorithms - ESA '94*, chapter Membership in constant time and minimum space, pages 72–81. Springer LNCS, Berlin / Heidelberg, 1994.

[18] Bjorn Brynjulfsson, Gisli Hjalmtysson, Kostas Katrinis, and Bernhard Plattner. Autonomic network-layer multicast service towards consistent service quality. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, pages 494–498, Washington, DC, USA, April 2006. IEEE Computer Society.

[19] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. A Reinforcement Learning Approach to Online Web Systems Auto-configuration. In *The 29th IEEE International Conference on Distributed Computing Systems*, pages 2–11, Washington, DC, USA, 2009. IEEE Computer Society.

[20] Lucian Busoniu, Robert Babuska, Bart De Schutter, and Damien Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010.

[21] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud Computing and Emerging IT platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[22] M. Caporuscio, A. Carzaniga, and A.L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *Software Engineering, IEEE Transactions on*, 29(12):1059–1071, Dec. 2003.

[23] Antonio Carzaniga, Matthew J. Rutherford, and Alexander L. Wolf. A routing scheme for content-based networking. In *INFOCOM*, 2004.

[24] Kenneth Chan and Iman Poernomo. Qos-aware model driven architecture through the uml and cim. In *Enterprise Distributed Object Computing Conference, 2006. EDOC '06. 10th IEEE International*, pages 345 –354, October 2006.

[25] Mani Chandy, Opher Etzion, Rainer von Ammon, and Peter Niblett. 07191 summary – event processing. In Mani Chandy, Opher Etzion, and Rainer von Ammon, editors, *Event Processing*, number 07191 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[26] Xiaolin Cheng, Prasant Mohapatra, Sung-Ju Lee, and Sujata Banerjee. Performance evaluation of video streaming in multihop wireless mesh networks. In *NOSSDAV '08: Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 57–62, New York, NY, USA, 2008. ACM.

[27] Abdur Rahim Choudhary. Policy based management in the global information grid. *International Journal of Internet Protocol Technology*, 3(1):72–80, 2008.

[28] Pierre-Charles David and Thomas Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97. Springer Berlin / Heidelberg, 2006.

[29] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM Comput. Surv.*, 27(3):326–327, 1995.

[30] G. Eisenhauer, K. Schwan, and F.E. Bustamante. Publish-subscribe for high-performance computing. *Internet Computing, IEEE*, 10(1):40–47, Jan.-Feb. 2006.

[31] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 5(6):784–803, 1997.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[33] Miller George. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review*, 63:81–97, 1956.

[34] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[35] Aniruddha Gokhale, Balachandran Natarajan, Douglas C. Schmidt, Andrey Nechypurenko, Jeff Gray, Nanbor Wang, Sandeep Neema, Ted Bapty, and Jeff Parsons. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications. In *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, November 2002. ACM.

[36] B.T. Gold, J. Kim, J.C. Smolens, E.S. Chung, V. Liaskovitis, E. Nurvitadhi, B. Falsafi, J.C. Hoe, and A.G. Nowatzyk. TRUSS: A Reliable, Scalable Server Architecture. *Micro, IEEE*, 25(6):51–59, Nov.-Dec. 2005.

[37] Paul Grace, Geoff Coulson, Gordon S. Blair, and Barry Porter. Deep Middleware for the Divergent Grid. In *Middleware '05: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 334–353, New York, NY, USA, 2005. Springer-Verlag New York, Inc.

[38] Paul Grace, Geoff Coulson, Gordon S. Blair, and Barry Porter. A Distributed Architecture Meta-model for Self-managed Middleware. In *Proceedings of the 5th Workshop on Adaptive and Reflective Middleware (ARM '06)*, page 3, New York, NY, USA, 2006. ACM.

[39] Jeff Gray, Yuehua Lin, Jing Zhang, Steve Nordstrom, Aniruddha Gokhale, Sandeep Neema, and Swapna Gokhale. Replicators: Transformations to Address Model Scalability. In *Lecture Notes in Computer Science: Proceedings of 8th International Conference Model Driven Engineering Languages and Systems (MoDELS 2005)*, pages 295–308, Montego Bay, Jamaica, November 2005. Springer Verlag.

[40] Brent Hailpern and Peri Tarr. Model-Driven Development: The Good, the Bad, and the Ugly. *IBM Systems Journal*, 45(3):451–461, July 2006.

[41] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, USA, 1989.

[42] Caroline Herssens, Stéphane Faulkner, and Ivan J. Jureta. Context-driven autonomic adaptation of sla. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 362–377, Berlin, Heidelberg, 2008. Springer-Verlag.

[43] Andrea Hess, Michael Nussbaumer, Helmut Hlavacs, and Karin Anna Hummel. *Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks*, chapter Automatic Adaptation and Analysis of SIP Headers Using Decision Trees, pages 69–89. Springer Berlin / Heidelberg, 2008.

[44] James H. Hill, John Slaby, Steve Baker, and Douglas C. Schmidt. Applying System

Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.

[45] Joe Hoffert, Aniruddha Gokhale, and Douglas Schmidt. Evaluating Transport Protocols for Real-time Event Stream Processing Middleware and Applications. In *Proceedings of the 11th International Symposium on Distributed Objects, Middleware, and Applications (DOA '09)*, Vilamoura, Algarve-Portugal, November 2009.

[46] Joe Hoffert, Douglas Schmidt, and Aniruddha Gokhale. A QoS Policy Configuration Modeling Language for Publish/Subscribe Middleware Platforms. In *Proceedings of International Conference on Distributed Event-Based Systems (DEBS)*, pages 140–145, Toronto, Canada, June 2007.

[47] Joe Hoffert and Douglas C. Schmidt. Maintaining qos for publish/subscribe middleware in dynamic environments. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems (DEBS 09)*, Nashville, TN, USA, July 2009.

[48] Joe Hoffert, Douglas C. Schmidt, and Aniruddha Gokhale. Adapting and Evaluating Distributed Real-time and Embedded Systems in Dynamic Environments. In *Proceedings of the 1st International Workshop on Data Dissemination for Large scale Complex Critical Infrastructures (DD4LCCI 2010)*, Valencia, Spain, April 2010.

[49] Yi Huang and Dennis Gannon. A comparative study of web services-based event notification specifications. *Proceedings of the International Conference on Parallel Processing Workshops*, 0:7–14, 2006.

[50] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing-degrees, models, and applications. *ACM Comput. Surv.*, 40:7:1–7:28, August 2008.

[51] Gordon Hunt. DDS Use Cases: Effective Application of DDS Patterns and QoS. In *OMG's Workshop on Distributed Object Computing for Real-time and Embedded Systems*, Washington, D.C., July 2006. Object Management Group.

[52] M. Ibnkahla, Q.M. Rahman, A.I. Sulyman, H.A. Al-Asady, Jun Yuan, and A. Safwat. High-speed Satellite Mobile Communications: Technologies and Challenges. *Proceedings of the IEEE*, 92(2):312 – 339, February 2004.

[53] Software Engineering Institute. Ultra-Large-Scale Systems: Software Challenge of the Future. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, June 2006.

[54] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[55] Yi Jin, Xu Liu, Jianfeng Zhan, and Shuang Gao. A Dynamic Provisioning Framework for Multi-tier Internet Applications in Virtualized Data Center. *Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. Ninth International Conference on*, pages 329–332, Dec. 2008.

[56] Amogh Kavimandan and Aniruddha Gokhale. Automated Middleware QoS Configuration Techniques using Model Transformations. In *Proceedings of the* 14^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 93–102, St. Louis, MO, USA, April 2008.

[57] Amogh Kavimandan, Anantha Narayanan, Aniruddha Gokhale, and Gabor Karsai. Evaluating the Correctness and Effectiveness of a Middleware QoS Configuration Process in Distributed Real-time and Embedded Systems. In *Proceedings of the* 11^{th} *IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2008)*, pages 100–107, Orlando, FL, USA, May 2008.

[58] Stuart Kent. *The unified modeling language*, pages 126–152. Cambridge University Press, New York, NY, USA, 2001.

[59] Stuart Kent. Model Driven Engineering. In *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM 02)*, pages 286–298, Turku, Finland, May 2002. Springer-Verlag LNCS 2335.

[60] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[61] J.M. Kirschberg, M.S. Delgado, and S.S. Ribes. Rccmp: reliable congestion controlled multicast protocol. In *1st EuroNGI COnference on Next Generation Internet Networks Traffic Engineering*, April 2005.

[62] Arvind S. Krishna, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt. Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems. In *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, pages 180–189, San Francisco, CA, March 2005. IEEE.

[63] Vibhore Kumar, Brian F. Cooper, and Karsten Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 3 –14, June 2005.

[64] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.

[65] Guoli Li and Hans-Arno Jacobsen. Composite Subscriptions in Content-based Publish/Subscribe Systems. In *Proceedings of the 6th International Middleware*

Conference, Grenoble, France, 2005.

[66] Qingping Lin, Hoon Kang Neo, Liang Zhang, Guangbin Huang, and Robert Gay. Grid-based Large-scale Web3D Collaborative Virtual Environment. In *Web3D '07: Proceedings of the Twelfth International Conference on 3D Web Technology*, pages 123–132, New York, NY, USA, 2007. ACM.

[67] Suping Liu and Yongsheng Ding. An adaptive network policy management framework based on classical conditioning. In *Proceedings of the 7th World Congress on Intelligent Control and Automation, (WCICA 2008)*, pages 3336 –3340, June 2008.

[68] Yong Liu. Create Stable Neural Networks by Cross-Validation. In *IJCNN '06: Proceedings of the International Joint Conference on Neural Networks*, pages 3925–3928, 2006.

[69] Joseph Loyall, Jianming Ye, Richard Shapiro, Sandeep Neema, Nagabhushan Mahadevan, Sherif Abdelwahed, Michael Koets, and Denise Varner. A Case Study in Applying QoS Adaptation and Model-Based Design to the Design-Time Optimization of Signal Analyzer Applications. In *Military Communications Conference (MILCOM)*, Monterey, California, November 2004.

[70] Robert C. Martin, editor. *More C++ gems*, chapter GPERF: a perfect hash function generator, pages 461–491. Cambridge University Press, New York, NY, USA, 2000.

[71] David McKinnon, Kevin E. Dorow, Tarana R. Damania, Olav Haugan, Wesley E. Lawrence, David E. Bakken, and John C. Shovic. A Configurable Middleware Framework with Multiple Quality of Service Properties for Small Embedded Systems. In *2nd IEEE International Symposium on Network Computing and Applications*, pages 197–204. IEEE, April 2003.

[72] M. Menth and R. Henjes. Analysis of the Message Waiting Time for the FioranoMQ JMS Server. *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 1–1, 2006.

[73] David Meyer, Friedrich Leisch, and Kurt Hornik. The support vector machine under test. *Neurocomputing*, 55(1-2):169 – 186, 2003. Support Vector Machines.

[74] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[75] Richard Monson-Haefel and David A. Chappell. *Java Message Service*. O'Reilly, 1st edition, December 2000.

[76] Ripal Nathuji, Canturk Isci, Eugene Gorbatov, and Karsten Schwan. Providing Platform Heterogeneity-awareness for Data Center Power Management. *Cluster Computing*, 11(3):259–271, 2008.

[77] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *Proceedings of EuroSys 2010*, pages 237–250, Paris, France, April 2010.

[78] Mohd Farhan Ngatman, Md Asri Ngadi, and Johan M. Sharif. Comprehensive study of transmission techniques for reducing packet loss and delay in multimedia over ip. *International Journal of Computer Science and Network Security*, 8(3):292–299, 2008.

[79] P. Niblett and S. Graham. Events and service-oriented architecture: the oasis web services notification specifications. *IBM Syst. J.*, 44:869–886, October 2005.

[80] Object Computing Incorporated. OpenDDS. `http://www.opendds.org`, 2007.

[81] Object Management Group. *Data Distribution Service for Real-time Systems Specification*, 1.2 edition, January 2007.

[82] Carlos O'Ryan, Douglas C. Schmidt, David Levine, and Russell Noseworthy. Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations. In *Proceedings of the 5^{th} Workshop on Object-oriented Real-time Dependable Systems*, Montery, CA, November 1999. IEEE.

[83] S. Ostermann, R. Prodan, and T. Fahringer. Extending Grids with Cloud Resource Management for Scientific Computing. In *10th IEEE/ACM International Conference on Grid Computing, 2009*, pages 42 –49, 13-15 2009.

[84] Gerardo Pardo-Castellote. OMG Data-Distribution Service: Architectural Overview. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCSW '03, pages 200–206, Washington, DC, USA, 2003. IEEE Computer Society.

[85] Dan W. Patterson. *Artificial Neural Networks: Theory and Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[86] Beth Plale, Dennis Gannon, Jerry Brotzge, Kelvin Droegemeier, Jim Kurose, David McLaughlin, Robert Wilhelmson, Sara Graves, Mohan Ramamurthy, Richard D. Clark, Sepi Yalda, Daniel A. Reed, Everette Joseph, and V. Chandrasekar. CASA and LEAD: Adaptive Cyberinfrastructure for Real-Time Multiscale Weather Forecasting. *Computer*, 39(11):56–64, 2006.

[87] R. Premraj, M. Shepperd, B. Kitchenham, and P. Forselius. An empirical analysis of software productivity over time. *Software Metrics, 2005. 11th IEEE International Symposium*, Sept. 2005.

[88] S. Ramani, K.S. Trivedi, and B. Dasarathy. Performance analysis of the corba notification service. In *Reliable Distributed Systems, 2001. Proceedings. 20th IEEE Symposium on*, pages 227 –236, October 2001.

[89] Kai Sachs, Samuel Kounev, Jean Bacon, and Alejandro Buchmann. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Perform. Eval.*, 66:410–434, August 2009.

[90] D.C. Schmidt and H. van't Hag. Addressing the challenges of mission-critical information management in next-generation net-centric pub/sub systems with opensplice dds. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –8, April 2008.

[91] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[92] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[93] Nishanth Shankaran, Xenofon Koutsoukos, Chenyang Lu, Douglas C. Schmidt, and Yuan Xue. Hierarchical Control of Multiple Resources in Distributed Real-time and Embedded Systems. *Real-Time Systems*, 1(3):237–282, April 2008.

[94] Surjalal Sharma and Steven Curtis. *Magnetospheric Multiscale Mission*. Springer Verlag, 2005.

[95] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. The MIT Press, March 1998.

[96] Sumant Tambe, Akshay Dabholkar, and Aniruddha Gokhale. Generative Techniques to Specialize Middleware for Fault Tolerance. In *Proceedings of the* 12^{th} *IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

[97] Yoshio Tanaka, Hiroshi Takemiya, Hidemoto Nakada, and Satoshi Sekiguchi. Design, Implementation and Performance Evaluation of GridRPC Programming Middleware for a Large-Scale Computational Grid. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 298–305, Washington, DC, USA, 2004. IEEE Computer Society.

[98] Sasu Tarkoma and Kimmo Raatikainen. State of the Art Review of Distributed Event Systems. Technical Report C0-04, University of Helsinki, 2006.

[99] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition, Third Edition*. Academic Press, Inc., Orlando, FL, USA, 2006.

[100] Yoav Tock, Nir Naaman, Avi Harpaz, and Gidon Gershinsky. Hierarchical Clustering of Message Flows in a Multicast Data Dissemination System. In *Proceedings of Parallel and Distributed Computing and Systems (PDCS 2005)*, pages 320–326, November 2005.

[101] Christos Tselikis, Sarandis Mitropoulos, and Christos Douligeris. An evaluation of the middleware's impact on the performance of object oriented distributed systems. *Journal of Systems and Software*, 80(7):1169 – 1181, 2007. Dynamic Resource Management in Distributed Real-Time Systems.

[102] Giuseppe Valetto, Laurent Walter Goix, and Guillaume Delaire. Towards Service Awareness and Autonomic Features in a SIP-Enabled Network. In *Autonomic Communication*, pages 202–213, Berlin, Heidelberg, 2006. Springer-Verlag.

[103] Patrice Vienne and Jean-Louis Sourrouille. A Middleware for Autonomic QoS Management Based on Learning. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, pages 1–8, New York, NY, USA, 2005. ACM.

[104] Jens von Pilgrim. Measuring the level of abstraction and detail of models in the context of mdd. In *Second International Workshop on Model Size Metrics*, pages 10–17, October 2007.

[105] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[106] Ming Xiong, Jeff Parsons, James Edmondson, Hieu Nguyen, and Douglas C. Schmidt. Evaluating Technologies for Tactical Information Management in Net-Centric Systems. In *Proceedings of the Defense Transformation and Net-Centric Systems conference*, Orlando, Florida, April 2007.

[107] Jianming Ye, Joseph Loyall, Richard Shapiro, Richard Schantz, Sandeep Neema, Sherif Abdelwahed, Nagabhushan Mahadevan, Michael Koets, and Denise Varner. A Model-Based Approach to Designing QoS Adaptive Applications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 221–230, Washington, DC, USA, 2004. IEEE Computer Society.