ALGORITHMS AND TECHNIQUES FOR SCALABLE, RELIABLE EDGE-TO-CLOUD INDUSTRIAL INTERNET OF THINGS

By

Kyoungho An

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, 2015

Nashville, Tennessee

Approved:

Dr. Aniruddha S. Gokhale

Dr. Douglas C. Schmidt

Dr. Janos Sztipanovits

Dr. Christopher J. White

Dr. Sumant Tambe

To my respected parents, Soocheon An and Heaja Kim,

and

To my beloved fiancée, Jihye Bae

ACKNOWLEDGMENTS

This work would not have been possible without the financial support of the National Science Foundation (NSF), the Air Force Research Labs (AFRL), and the Department of Energy (DoE).

I would like to express my sincere gratitude to my advisor Dr. Aniruddha Gokhale for his continuous support during my Ph.D. studies and research, for his guidance, patience, and encouragement. Whenever I needed a discussion about research ideas, he was always willing to converse and provided me with insightful feedbacks. I also would like to thank Dr. Douglas Schmidt, Dr. Jules White, and Dr. Janos Sztipanovits for serving as members of my dissertation committee and for providing insights on my research.

The internship work that I have done at Real-Time Innovations (RTI) allowed me to find direction for my research and complete the dissertation. I would like to appreciate Dr. Sumant Tambe, Dr. Paul Pazandak, and Dr. Gerardo Pardo-Castellote for the internship opportunity and their advice to develop my research idea and its prototype. I am especially thankful for Dr. Sumant Tambe who served as my dissertation committee member as well. I am also grateful for RTI engineers who helped implement and experiment my work.

This work would have been incomplete if it were not the feedbacks and support given by the DOG group members: Subhav Pradhan, Faruk Caglar, Prithviraj Patil, Shashank Shekhar, Shweta Khare, Yogesh Barve, Shunxing Bao, and Anirban Bhattacharjee. I also like to express my thanks to our system administrator, Eric Hall, for setting testbeds used for experiments.

And of course, my family, fiancée, and members of Bridge Community Church at Nashville. I could not have made this achievement without the support and prayers from them. Last but not least, I praise my Lord, the mighty Savior, for providing me this opportunity and granting me the capability to finish this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

I.1 Emerging Trends

Significant advances in server and network technologies, and an explosive growth in the number of sensors and mobile devices have given rise to a number of intelligent services, which are grouped under a new paradigm called the Internet of Things (IoT). IoT involves machine-to-machine communication technologies that enable embedded processors, mobile devices, and sensors to communicate with one another with limited human intervention or none at all [90]. A special case of the IoT paradigm that focuses primarily on domains for mission-critical applications is called the *Industrial Internet of Things (IIoT)* [79].

IIoT systems are poised to deliver substantial societal, economical and environmental benefits across multiple sectors by realizing smart transportation, smart healthcare, agile and smart manufacturing and smart energy [3, 5, 10, 25, 29]. One trait of IIoT includes the need for big data analytics due to the sheer volume, variety and velocity of the data that is generated from many different sources [48]. Big data analytics for IIoT will require cloud computing [2] capabilities so that the data is processed and analyzed for the different intelligent services in an economic, elastic, and scalable manner. It will also require a scalable messaging middleware for communication between the numerous machines [26] involved in the system. Therefore, scientific advances in the systems software for the discovery and data dissemination between machines at the edge as well as the cloud and also timely and reliable analytics conducted in the cloud are significant to fulfill the performance and reliability demands of IIoT services.

The current vision of IIoT has gradually evolved from existing concepts such as distributed real-time embedded (DRE) systems [77] and cyber physical systems (CPS) [44],

where quality of service (QoS) properties, such as timeliness, reliability and security are important. The second trait of IIoT thus involves the need for stringent QoS assurance for data discovery, dissemination and data processing, along the dimensions of timeliness, reliability, and security. The presence of a very large number of connected edge devices, cloud infrastructures for real-time big data analytics, and QoS requirements at all these levels makes it challenging to realize IIoT and differentiates it from prior concepts. The IIoT paradigm also requires autonomic computing capabilities [40] because the big data analytics used in operational reliability and economical efficiency is also used to build predictive models for self-diagnosis, self-correction and self-optimization.

Although the need for cloud computing to perform big data analytics in IIoT was highlighted above, it may not always be feasible to transfer very large amounts of data from deeply embedded sensor networks to a cloud. IIoT systems such as wind farms and mining operations require data analytics at different levels including the machine, plant, and enterprise-levels [22, 82]. This three-tier architectural pattern is gaining acceptance as evidenced by an ongoing work on a reference architecture for IIoT by the Industrial Internet Consortium (IIC) [16]. Each level of analytics corresponding to this architecture and shown in Figure 1 illustrates different requirements including the need to handle different data types, volume of data, frequency, and response time expected. Figure 1 also presents representative numbers for the data quantity and data frequency numbers for each analytics level for a wind farm IIoT system provided GE [82] that indicates the different scales of the overall system. At the machine level, the data quantity is low while data frequency is high. In contrast to the machine level, the enterprise level requires computation on very large quantities of data and but has a low data frequency.

This classification indicates that each level will likely adopt divergent approaches to meet their functional and non-functional requirements. The key to getting accurate outcomes from the analytics at each level stems from the ability of the systems software to deliver and collect the right data at the right time and in the right quantities. To realize

	Machine (Turbine)	Plant (Wind Farm)	Enterprise (Power Producer)
Data Quantity	> 100 tags	> 6,000 tags	> 1,000,000 tags
Data Frequency	40 milliseconds	1 second	1 second - 10 minutes

Figure 1: GE's Three-level Model for Big Data Analytics in an IIoT System using Wind Farm Example [81]

these essential traits, the system software, which often is in the form of distributed communications middleware that supports data sharing used in machine and plant levels, and also cloud infrastructures used in the enterprise level, must provide correct and desired QoS according to requirements of each level.

I.2 Challenges for IIoT Middleware

Using the 3-level analytics model presented by GE, we have formulated a corresponding 3-level IIoT systems problem formulation, which is subsequently used to situate the research challenges addressed by this doctoral research in realizing the systems software for IIoT. Figure 2 depicts such an architecture comprising a cloud (representing the enterprise level) and the edge (representing both the plant and machine level). In this architecture, the devices and sensors in a machine (*e.g.*, wind turbine in the wind farm example) share data via a machine data bus for optimizing operations of the machines. The research issues at these level stem primarily from having to meet the hard real-time, safety, reliability and security challenges.

At the plant-level, data from the machines can also be shared with other machines or

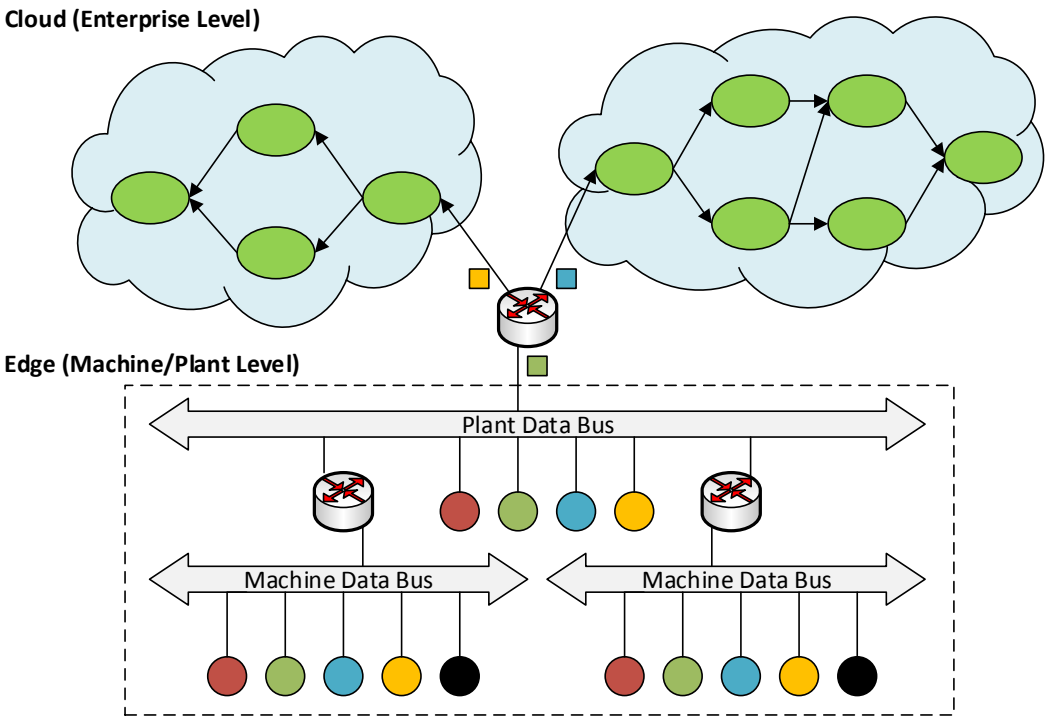

Figure 2: Distributed and Cyber-Physical Architecture for IIoT Systems

with operation centers via a plant data bus. Such information may be needed to evaluate health and performance of the plants. Finally, data from multiple plants can be collected and aggregated in the cloud, and real-time streaming or batching analytics can be conducted to help the global operations team prioritize maintenance, logistics, and other services for the different plants. The timing requirements at both these levels have soft real-time requirements where average response times are important though the plant-level demonstrates more stringent QoS requirements than the enterprise-level. Each level has its own reliability and security challenges.

To scope out the contours of this doctoral research, we have focused on the scientific challenges at the plant and enterprise-levels only. The challenges at the machine-level are predominantly concerned with meeting hard real-time deadlines, and is outside the scope of this research. A close scrutiny of the problem space involving the plant and enterprise

levels illustrates two key traits: (1) communication between endpoints is predominantly data-centric [64], and (2) communicating endpoints demonstrate publish/subscribe (pub/-sub) [20] semantics.

The pub/sub semantics stem from the fact that data sources make their data available to anyone interested in it while not requiring to know the identity of the recipients. Similarly, the receiving entities express interest in receiving one or more types of data without having to know the precise sources of this information. Moreover, the pub/sub paradigm involved event-based communications, and makes use of multicast so that it can disseminate data to many subscribers in real-time and a scalable way

The data-centricity stems from the fact that a large number of sensors generate data with different types and values as well as at different rates and volume. Among pub/sub protocols for IIoT, the Advanced Message Queuing Protocol (AMQP) [88], Message Queue Telemetry Transport (MQTT) [43], and Java Messaging Service (JMS) [30] are message-centric technologies while OMG's Data Distribution Service (DDS) [63] is a data-centric technology [66]. Both types of technologies are similar with respect to providing pub/sub communications in distributed systems, but the way they achieve it is different. A message-centric technology delivers the message itself without considering the data content and QoS. On the other hand, a data-centric technology provides infrastructure with higher level of abstraction to deliver messages with a control of data flows based on data content as well as QoS configurations.

The data-centric pub/sub communication semantics are of significant importance to the success of IIoT as they support scalability and reliability with higher levels of abstractions. This doctoral research investigates solutions to address a subset of challenges in the context of data-centric pub/sub at the plant and enterprise levels of IIoT. The focus areas for this dissertation are situated according to the levels of IIoT systems and are summarized in Figure 3.

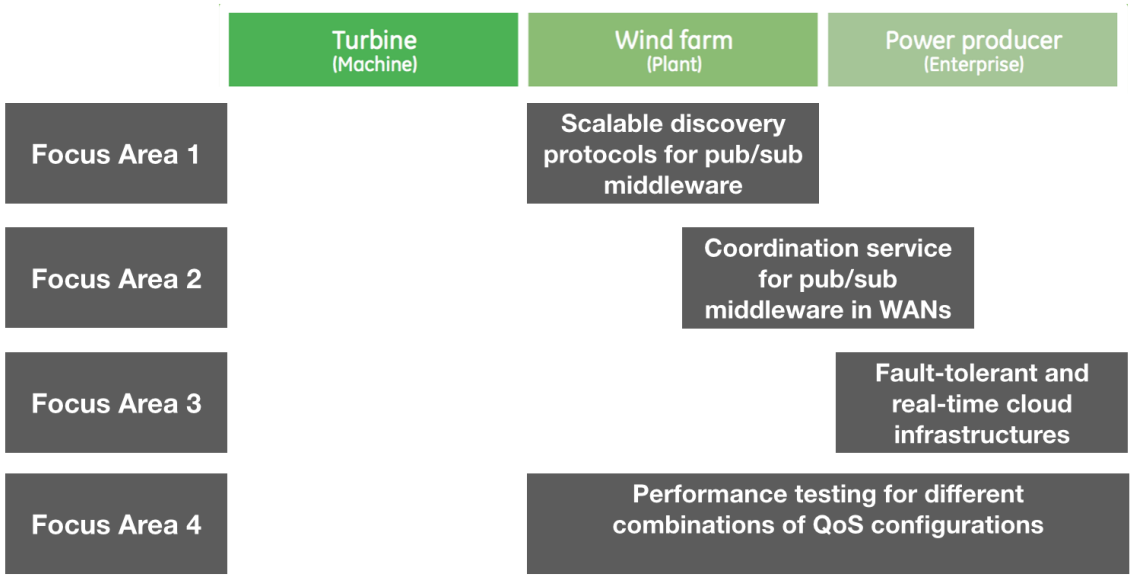

Figure 3: Dissertation Focus Areas

Challenge 1: Scalable Discovery of Publishers and Subscribers at the Plant level

IIoT systems often involve publishers and subscribers that may join or leave the system dynamically. This may happen because entities may be mobile, may fail (*e.g.*, sensors may die), or they may be deactivated as part of maintenance and technology refresh cycles. Due to the dynamic churn in publishers and subscribers, effective data communication between publishers and subscribers requires dynamic and reliable discovery of pub/sub endpoints. For discovering peers in pub/sub systems, several mechanisms such as static [72], centralized [73], federated [58], and peer-to-peer [61] have been used.

Even though existing data-centric pub/sub technologies, such as the OMG Data Distribution Service (DDS) [57] currently support dynamic peer-to-peer endpoint discovery via a standardized approach called the Simple Discovery Protocol (SDP), for large-scale systems, however, SDP scales poorly since the discovery completion time grows as the number of applications and endpoints increases. To scale to much larger systems, a more efficient discovery protocol is required.

Challenge 2: Coordination Across the Plant and Enterprise Levels

To share the data and results of analytics across the connected devices at the edge, information must cross the plant boundaries reaching the enterprise level to conduct the analytics, and results must be delivered to other plants so they reach the machines in those destination plants. To address these requirements, existing data-centric pub/sub technologies such as DDS can be used because they provide benefits including scalable data dissemination with support for multiple transport protocols such as TCP and UDP, and a rich set of configurable QoS policies including content-based filtering.

Despite the many benefits of DDS, there are challenges in using DDS for WAN-based applications as in IIoT. First, DDS uses multicast as a default transport for discovery messages (*i.e.* control-plane messages) to locate endpoints in a system. If endpoints are located in isolated networks not supporting multicast, these endpoints cannot be discovered by other peers. In addition, because of network firewalls and network address translation (NAT), even if endpoints are discovered, peers may not be able to deliver data-plane messages to the destination endpoints. DDS brokering solutions [28, 74] show promise, however, for IIoT systems where a number of heterogeneous devices and networks exist, a middleware solution to efficiently and scalably discover and coordinate DDS brokers located in isolated networks remains an unresolved issue. Specifically, to deliver data from source to destination in WANs, multiple brokers need to collaborate to form the data dissemination paths. Moreover, coordination among brokers to (re)establish dissemination paths for dynamic endpoints in a consistent and efficient manner also remains as a challenging problem.

Challenge 3: Reliable Soft Real-time Cloud Infrastructure

At the enterprise level of IIoT systems, big data analytics involving both real-time stream processing and batch computing will often be performed in the cloud. In cloud infrastructures, machines or processes running a virtual machine (VM) may fail, which

may be detrimental and in some cases catastrophic for the overall health, efficiency and operation of IIoT systems. To guarantee both timeliness (soft real-time) and high availability for big data analytics in the cloud, cloud infrastructures should provide fault-tolerant mechanisms and intelligent resource management.

Solutions to address these requirements exist. For example, Remus [18] periodically creates snapshots of states of primary VMs and rollback to synchronized backup VMs when failures happen. However, these solutions do not consider replica placement, which incurs shortcomings in the context of real-time applications when contention for shared resources occurs. If backup VMs are placed on highly overloaded physical machines in the case that does not consider resource constraints for placement, it cannot assure timeliness requirements of applications due to high latency incurred by resource sharing after a failover happens. Moreover, optimizing deployment of backup replica VMs considering operation costs such as energy consumption is desired.

Challenge 4: Validating Properties of Pub/Sub Algorithms

Using existing data-centric pub/sub solutions, such as DDS, in a proper way is complex as it is hard to predict performance of applications after combining multiple QoS configurations because the flexibility offered due to configurability further increases application complexity in terms of validating their resultant behavior. To date, most deployments of these technologies have been in controlled environments, such as local area networks. IIoT systems represent a vastly different deployment environment where deploying and configuring pub/sub technologies is a hard task. Validating new data-centric, pub/sub solutions that address the inherent IIoT challenges makes this task even harder.

To overcome this problem, design-time formal methods have been applied with mixed success, but they lack sufficient accuracy in prediction, tool support, and an understanding of formalism has prevented wider adoption of the formal techniques. Therefore, a technique

is needed to reduce manual testing efforts and accidental complexity to find an optimal configuration in a planned emulation environment.

I.3 Proposed Doctoral Research: Scalable and QoS-enabled IIoT Middleware for Cloud and Edge

To address the challenges identified in Section I.2, this dissertation discusses algorithms and techniques to enhance the scalability, timeliness, and reliability of QoS-enabled data-centric pub/sub middleware and virtualized infrastructures used in IIoT systems at the plant and enterprise levels. We have used the OMG DDS technology as the vehicle to implement and evaluate our ideas. The novel contributions of this dissertation focus on addressing the three identified inherent complexities (Challenges 1, 2, and 3) and the one significant source of accidental complexity (Challenge 4) as follows:

1. **Content-based Filtering Discovery Protocol (CFDP)** is our new endpoint discovery mechanism that employs content-based filtering to conserve computing, memory and network resources used in the DDS discovery process. It embraces the design of a CFDP prototype implemented in a popular DDS implementation. We analyze the results of empirical studies conducted in a testbed we developed to evaluate the performance and resource usage of our CFDP approach compared with the traditional SDP approach. Chapter II describes this dimension of the research in details.

2. **PubSubCoord** is a cloud-based coordination and discovery service for QoS-enabled data-centric pub/sub for wide area network (WAN) operations. PubSubCoord realizes a WAN-scale, low-latency data dissemination architecture by (a) balancing the load using elastic cloud resources, (b) clustering brokers by topics for affinity, and (c) minimizing the number of data delivery hops in the pub/sub overlay. PubSubCoord's coordination mechanism uses ZooKeeper to support dynamic discovery of brokers and pub/sub endpoints located in isolated networks. Empirical results evaluating the

performance of PubSubCoord are presented for (1) scalability of data dissemination and coordination, and (2) deadline-aware overlays employing configurable QoS to provide low-latency data delivery for topics demanding strict service requirements. Chapter III describes the design and implementation as well as experimental results of PubSubCoord.

3. We propose **a cloud middleware for high availability of soft real-time applications** that automatically deploys replicas of VMs in cloud data centers in a way that optimizes resources while assuring availability and responsiveness. It realizes the design of a pluggable framework within the fault-tolerant architecture that enables plugging in different placement algorithms for VM replica deployment. Experimental results using a case study that involves a specific replica placement algorithm are presented to evaluate the effectiveness of our architecture. Chapter IV describes our contributions.

4. **AUTOMATIC** is a model-based performance testing framework with generative capabilities to reduce manual efforts in generating and verifying a large number of relevant QoS configurations of data-centric pub/sub applications and deploy them on a cloud platform. Chapter V describes the design and implementation of our automated testing framework and provide a case study with experimental results.

The rest of this dissertation describes the research we have conducted on the focus areas and summarizes contributions of the dissertation and alludes to remaining challenges which are proposed as future work.

CHAPTER II

CONTENT-BASED FILTERING DISCOVERY PROTOCOL (CFDP): SCALABLE AND EFFICIENT OMG DDS DISCOVERY PROTOCOL

II.1 Motivation

The pub/sub communication paradigm [21] is attractive due to its inherent scalability, which decouples publishers and subscribers of event data in time and space, and enables them to remain anonymous and communicate asynchronously. A data subscriber is an entity that consumes data by registering its interest in a certain format (*e.g.*, topics, types or content) at any location and at any time. Likewise, a data publisher is an entity that produces data for consumption by interested subscribers.

A key requirement for the pub/sub paradigm is the discovery of publishers by subscribers. Although anonymity is often a key trait of pub/sub, the underlying pub/sub mechanisms that actually deliver the data from publishers to subscribers must map subscribers to publishers based on matching interests. To support spatio-temporal decoupling of publishers and subscribers, an efficient and scalable discovery mechanism is essential for pub/sub systems since publishers and subscribers need not be present at the same time and in the same location.

Achieving efficient and scalable discovery is even more important for pub/sub systems that require QoS properties, such as latency of data delivery, reliable delivery of data, or availability of historical data. To meet the requirements of various systems, a range of discovery mechanisms exist. Example mechanisms include using static and predetermined lookups, using a centralized broker, or using a distributed and decentralized approach.

The DDS [60, 63] is a standardized pub/sub middleware that provides a range of QoS properties to distributed real-time and embedded (DRE) systems in which discovery is a key challenge. Since DDS supports QoS policies that enable it to disseminate data in a reliable

and real-time manner, it has been used to build many mission-critical DRE systems, such as air traffic control, unmanned vehicles, and industrial automation systems.

The discovery mechanism defined in the DDS standard is based on a peer-to-peer (P2P) protocol, where a peer automatically discovers other peers by matching the topic names, their data types, and their selected QoS configurations. DDS peers that contain the endpoints (*i.e.*, actual data publishers or subscribers) are required to locate remote matching peers with their endpoints to establish communication paths. Each peer thus runs a discovery protocol to find matching remote endpoints.

The DDS standard adopts a distributed and decentralized approach called the *Simple Discovery Protocol* (SDP) [61]. SDP provides simple and flexible system management by using discovery traffic for joining and leaving endpoints. It also supports updating the QoS status of endpoints.

II.1.1 Challenges

Although SDP is standardized, one drawback is that it scales poorly as the number of peers and their endpoints increases in a domain since each peer sends/receives discovery messages to/from other peers in the same domain [75]. When a large-scale DRE system is deployed with SDP, therefore, substantial network, memory and computing resources are consumed by every peer just for the discovery process. This overhead can significantly degrade discovery completion time and hence the overall scalability of a DDS-based pub/sub system.

The root cause of SDP's scalability issues is that peers send discovery messages to every other peer in the domain, yet perhaps only a fraction of the peers are actually interested in conversing with any other peer. As a result, unnecessary network, computing and memory resources are used for this discovery protocol. For example, a data consumer receives and keeps discovery objects (these objects describe all of topic names and data type formats of the data that each consumer uses) for all other consumers even though they never

communicate with each other since they are identical types of endpoints (*e.g.*, they are all subscribers).

II.1.2 Solution Approach

To overcome this limitation with SDP, we suggest a new mechanism for scalable DDS discovery called the *Content-based Filtering Discovery Protocol* (CFDP). CFDP employs content filtering on the sending peer(s) to filter discovery messages by exchanging filtering expressions that limit the range of interests *a priori*. To implement SDP, we created a special DDS topic called the *Content Filtered Topic (CFT)*, which includes filtering properties that are composed of a filtering expression and a set of parameters used by that expression. By using CFT, peers on the sending side that use CFDP can filter unwanted discovery messages and enhance the efficiency and scalability of the discovery process. The results of our empirical evaluations presented in Section II.4 demonstrate a linear reduction in the number of transferred and stored messages.

II.2 Background

This section summarizes the key elements of the OMG *Data Distribution Service* (DDS) and its *Simple Discovery Protocol* (SDP).

II.2.1 OMG DDS

The OMG DDS specification defines a distributed pub/sub communications architecture [60]. At the core of DDS is a data-centric architecture for connecting anonymous data publishers with data subscribers, as shown in Figure 4. The DDS architecture promotes loose coupling between system components. The data publishers and subscribers are decoupled with respect to (1) time (*i.e.*, they need not be present at the same time), (2) space (*i.e.*, they may be located anywhere), (3) flow (*i.e.*, data publishers must offer equivalent or better quality-of-service (QoS) than required by data subscribers), and behavior (*i.e.*,

business logic-independent), (4) platforms, and (5) programming languages (*e.g.*, DDS applications can be written in many programming languages, including C, C++, Java, and Scala).

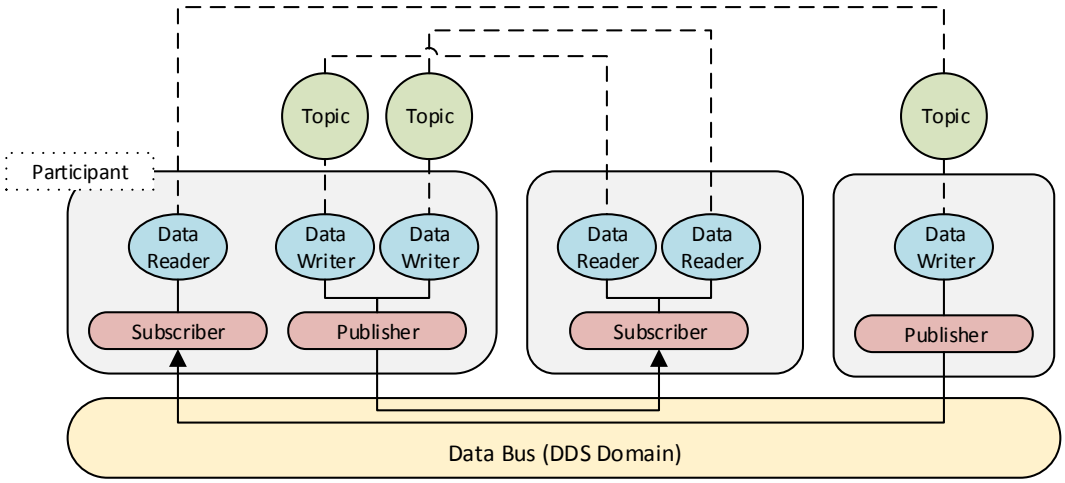

Figure 4: DDS Architecture

A DDS data publisher produces typed data-flows identified by names called *topics*. The coupling between a publisher and subscriber is expressed only in terms of topic name, its data type schema, and the offered and requested QoS attributes of publishers and subscribers, respectively. Below we briefly describe the key architectural elements of the DDS specification.

- **Domain** is a logical communication environment used to isolate and optimize network communications within the group of distributed applications that share common interests (*i.e.*, topics and QoS). DDS applications can send and receive data among themselves only if they have the same domain ID.

- **Participant** is an entity that represents either a publisher or subscriber role of a

DDS application in a domain, and behaves as a container for other DDS entities (*i.e.*, DataWriters and DataReaders), which are explained next.

- **DataWriter and DataReader.** *DataWriters* (data publishers) and *DataReaders* (data subscribers) are endpoint entities used to write and read typed data messages from a global data space, respectively. DDS ensures that the endpoints are compatible with respect to topic name, their data type, and QoS configurations. Creating a DataReader with a known topic and data type implicitly creates a *subscription*, which may or may not match with a DataWriter depending upon the QoS.

- **Topic** is a logical channel between DataWriters and DataReaders that specifies the data type of publication and subscription. The topic names, types, and QoS of DataWriters and DataReaders must match to establish communications between them.

- **Quality-of-service (QoS).** DDS supports around two dozen QoS policies that can be combined in different ways. Most QoS policies have requested/offered semantics, which are used to configure the data flow between each pair of DataReader and DataWriter, and dictate the resource usage of the involved entities.

- **Reliability QoS** controls the reliability of the data flow between DataWriters and DataReaders. BEST_EFFORT and RELIABLE are two possible alternatives. BEST_EFFORT reliability does not use any cpu/memory resource to ensure delivery of samples. RELIABLE, on the other hand, uses an ack/nack based protocol to provide a spectrum of reliability guarantees from *strict* (*i.e.,* fully reliable) to BEST_EFFORT.

- **Durability QoS** specifies a mechanism for DataWriters to store and deliver previously published data to late-joining DataReaders. This QoS provides 3 different levels of storing history data: VOLATILE (not storing), TRANSIENT (storing in memory), PERSISTENT (storing in persistent storage like a disk).

- **Content Filtered Topic (CFT).** Content filters refine a topic subscription and help to eliminate samples that do not match the defined application-specified predicates. The predicate is a string encoded SQL-like expression based on the fields of the data type. The filter expression and the parameters may change at run-time. Data filtering can be performed by the DataWriter or DataReader.

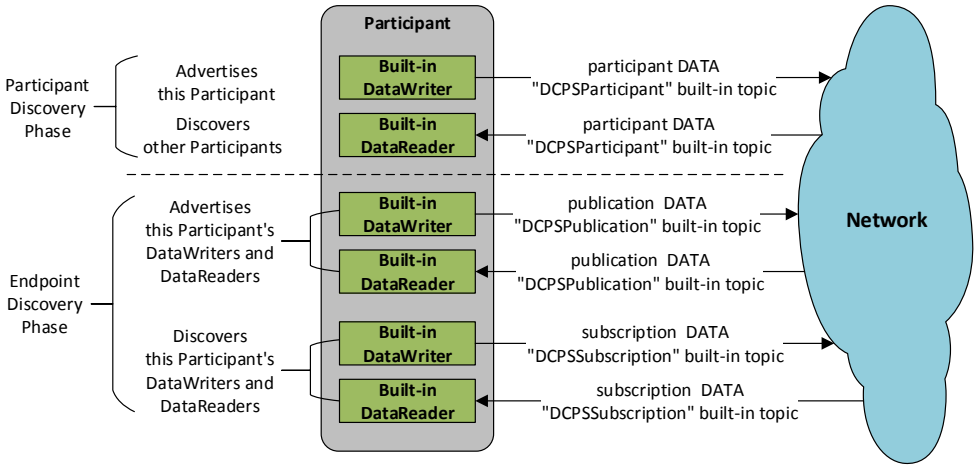

Figure 5: DDS Discovery Protocol Built-in Entities

II.2.2 OMG DDS SDP

The OMG DDS *Real-Time Publish-Subscribe* (RTPS) standard [61] defines a discovery protocol that splits the discovery process into two phases: the *Participant Discovery Protocol* (PDP) and the *Endpoint Discovery Protocol* (EDP). The PDP defines the means for discovering participants in a network. After participants have discovered each other, they exchange discovery messages for endpoints via the EDP.

The standard DDS specification describes a concrete discovery protocol called the *Simple Discovery Protocol* (SDP) as the default discovery protocol to be used by different DDS implementations for interoperability. SDP also uses the two phase approach, which

are called the *Simple Participant Discovery Protocol* (SPDP) and *Simple Endpoint Discovery Protocol* (SEDP). These discovery protocols are suitable for deployments of DDS in *Local-Area Networks* (LANs).

In the SPDP phase, a participant in a DDS application uses multicast or unicast discovery messages to announce named *participant DATA* to other participants periodically. These messages use built-in topics (*i.e.*, special topics for discovery messages) for discovery to let existing participants know of a new "announcing" participant. Figure 5 depicts the different built-in DDS entities (topics and endpoints) used by SDP. The SPDP message contains a participant's *Globally Unique Identifier* (GUID), transport locators (IP addresses and port numbers), and QoS policies. The message is periodically sent with the BEST_EFFORT reliability QoS to maintain liveliness of discovered participants. When *participant DATA* messages are received from other participants, the received messages for remote participants are archived in a database managed by the DDS middleware.

After a pair of remote participants discover each other by exchanging discovery messages, they transition to the SEDP phase. In this phase, remote entities (*i.e.*, remote participants or endpoints) imply the remotely-located entities from the entity that initiated the discovery service. Likewise, local entities (*i.e.*, local participants or endpoints) indicate the locally-located entities in the same process address space.

After the SPDP phase, SEDP begins to exchange discovery messages of endpoints using the RELIABLE reliability QoS. These messages are known as *publication DATA* for DataWriters and *subscription DATA* for DataReaders, respectively. They include topic names, data types, and QoS of discovered endpoints. In the SEDP phase, participants archive received remote endpoints' information into an internal database and start the matching process. During this process the communication paths for publication and subscription are established between the matching endpoints if the remote endpoints have the same topic name, data types, and compatible QoS configurations with the ones of their local endpoints.

Rather than using an out-of-band mechanism for discovery, SDP uses DDS entities (*i.e.*, built-in DDS entities) as an underlying communication transport for exchanging discovery information outlined above. As an initial step, an SDP-enabled participant creates and uses built-in DDS entities to publish and subscribe discovery data. The built-in entities use predefined built-in topics (*DCPSParticipant*, *DCPSPublication*, and *DCPSSubscription*) to discover remote participants and endpoints, as shown in Figure 5.

The built-in entities have two variants: DataWriters for announcement and DataReaders for discovery. Each discovery topic for different entities (*participant*, *publication*, and *subscription*) therefore has a pair of DataWriters and DataReaders. After built-in entities are initially created, the participants use it to exchange discovery messages for SDP.

II.2.3 Discovery Services for Pub/Sub Communications

In this section, we describe existing discovery mechanisms for pub/sub communications. For pub/sub communications, initially, peers running on devices need to search other peers in a system. Next, discovered peers begin exchanging discovery information for pub-/sub endpoints as topics and types. Once these discovery phases are done, communications of matching pub/sub endpoints are established. We partition these procedures into three phases: device discovery, endpoint discovery, and data communication.

Figure 6 shows procedures of the peer-to-peer discovery service with enabling multicast. In the device discovery, peers dynamically announce their locator information to others in a system via multicast. In the endpoint discovery, discovered peers exchange endpoint information (*i.e.*, topics, types, QoS parameters if supported) to find matching pub/sub endpoints. The peer-to-peer discovery service with multicast provides the zero configuration infrastructure that does not require any centralized services and manual configuration efforts for discovery to users, so it is called dynamic or automatic discovery protocol.

If a system does not support multicast, it is possible to handle discovery procedures

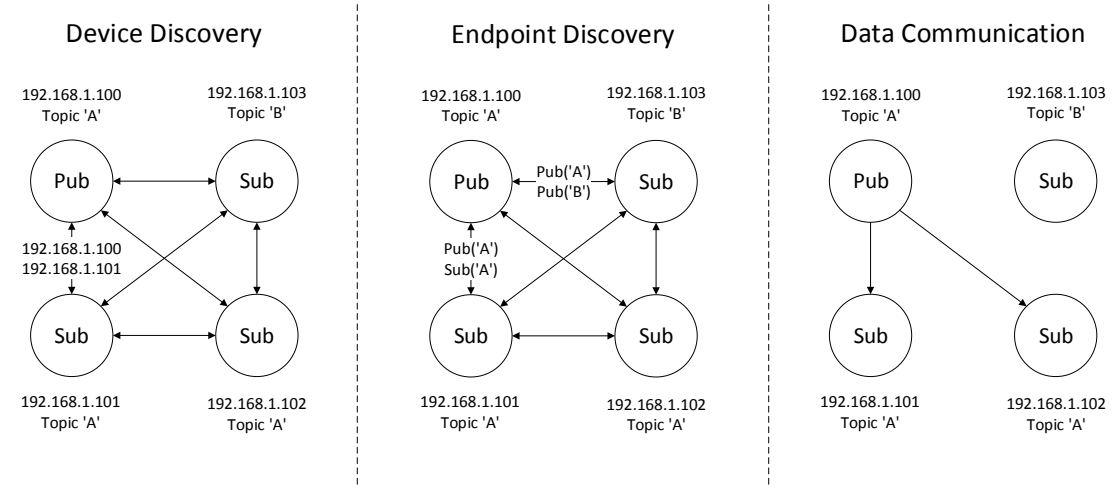

Figure 6: Peer-to-peer Discovery Service with Multicast

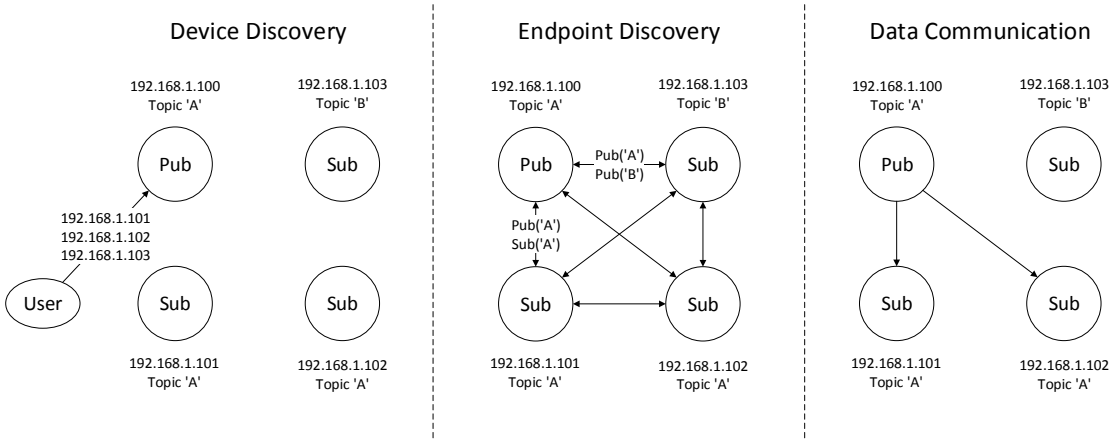

Figure 7: Peer-to-peer Discovery Service with Unicast

using unicast transmission. This mechanism requires manual configurations by users to set locators of other peers at the device discovery phase. Like the multicast peer-to-peer approach, this way also does not require external centralized services for discovery.

The static discovery involves user's efforts to configure discovery information by hand. This approach is beneficial to systems with limited network bandwidth as it does not use resources for discovery. Still, this discovery method cannot be used in large-scale systems where many peers and endpoints exist that cause a lot of configuration efforts.

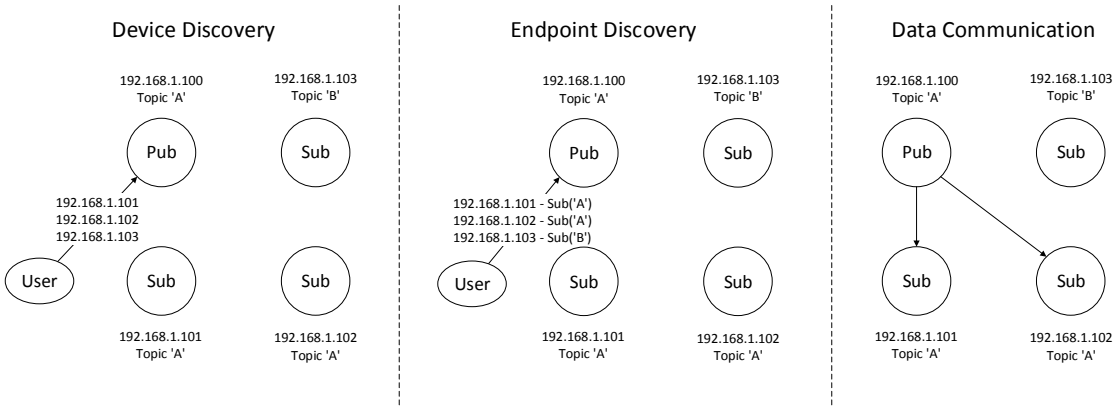

Figure 8: Static Discovery Service

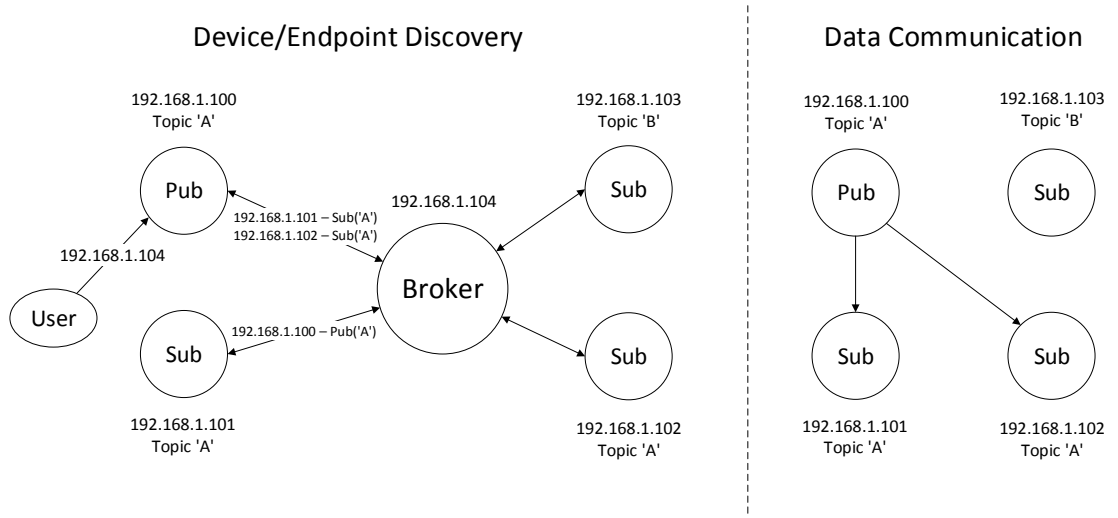

Figure 9: Centralized Discovery Service

The centralized discovery service is a commonly used way to share discovery information between peers as it is easy to manage with avoiding duplicated discovery information and requires minimal manual efforts. However, this approach causes problems when the centralized service is failed or congested and these incidents would affect the whole system. In the discovery phase, each peer access a well-known discovery server and exchange discovery information via the server.

To resolve the single point of failure issue in the centralized discovery, the federated

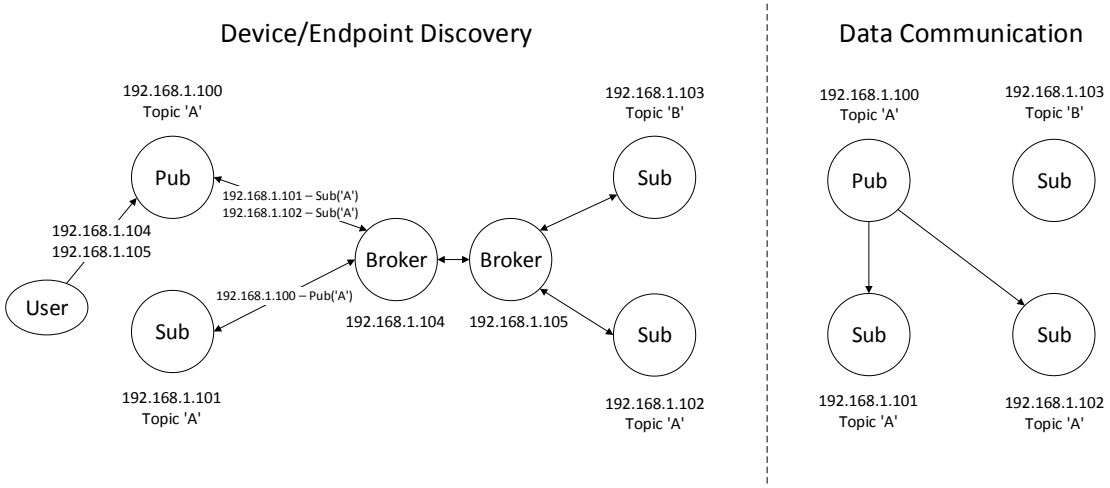

Figure 10: Federated Discovery Service

discovery has been proposed. This mechanism is basically similar to the centralized discovery except for the federation of the centralized discovery servers so that it is fault-tolerant and scalable by balancing loads on multiple servers compared to the centralized approach. Implementations of synchronizing information between servers and failover logic vary by solutions.

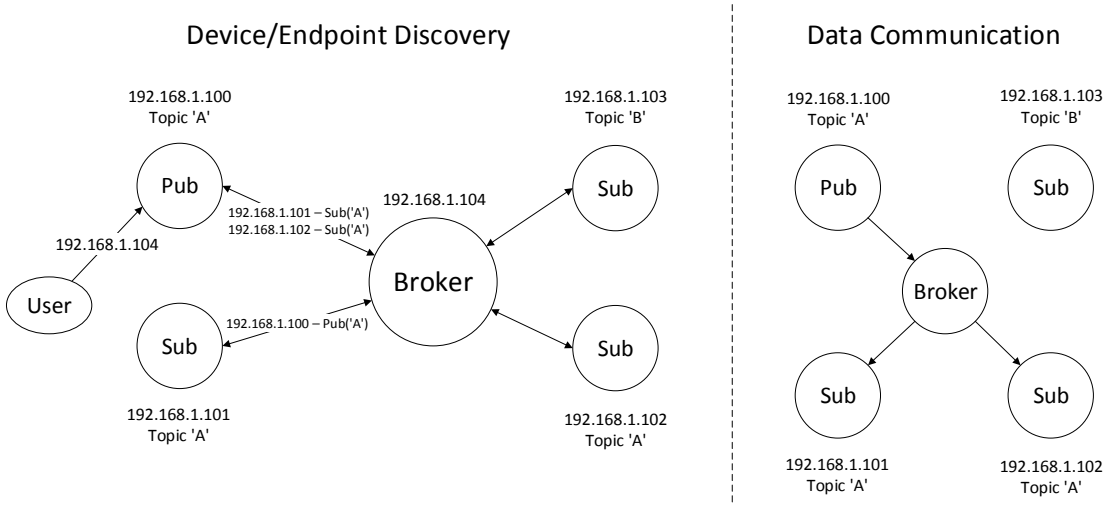

Figure 11: Broker-based Discovery and Pub/Sub Communication

Many pub/sub technologies such as AMQP, MQTT, and JMS use centralized or federated servers called *brokers* for not only discovery but also pub/sub communications. This approach is better than peer-to-peer in terms of memory management for persistent data (*i.e.*, sending previously published data to late joining subscribers) and scalability by delegating distribution overheads to federated brokers. Still, it leads to extra delivery hops causing higher latency and is delicate to failures of brokers.

II.3 Design and Implementation

Section II.1.1 highlighted the drawbacks of the existing standardized protocol for peer discovery in DDS called *Simple Discovery Protocol* (SDP). The key limitations in SDP stemmed from multicasting discovery messages to all other peers in the LAN, many of whom may not have a match between the publication and subscription. This protocol unnecessarily wastes network, storage and computation resources. To overcome this limitation, we designed the *Content-based Filtering Discovery Protocol* (CFDP). This section first describes the design of CFDP and then outlines key elements of its implementation.

II.3.1 The Design of CFDP

CFDP filters discovery messages based on topic names and endpoint types, thereby reducing the number of resources wasted by the traditional SDP approach. Like SDP, CFDP utilizes the first phase of SDP called SPDP (described in Section II.2.2) for participant discovery. It differs from SDP in the endpoint discovery phase known as SEDP, where key modifications have been designed for CFDP.

Similar to how SEDP applies DDS built-in entities as a communication transport to share discovery information, CFDP also uses DDS built-in entities to exchange discovery messages, however with some modifications.

In our design, we have used a special feature called *Content Filtered Topic* (CFT) that is supported in our underlying DDS implementation. CFT filters data samples on the

DataWriter or DataReader side in accordance with the filtering expression defined in a DataReader.

CFDP's key enhancement was to create built-in entities with CFTs that filter discovery messages on topic names stored in *subscription DATA* and *publication DATA*. Since built-in topics already exist for discovering publication and subscription in the SEDP design, the CFDP creates separate built-in CFTs and filtering expressions for DataWriters and DataReaders. The application logic is completely oblivious to these steps because everything is handled at the DDS middleware-level.

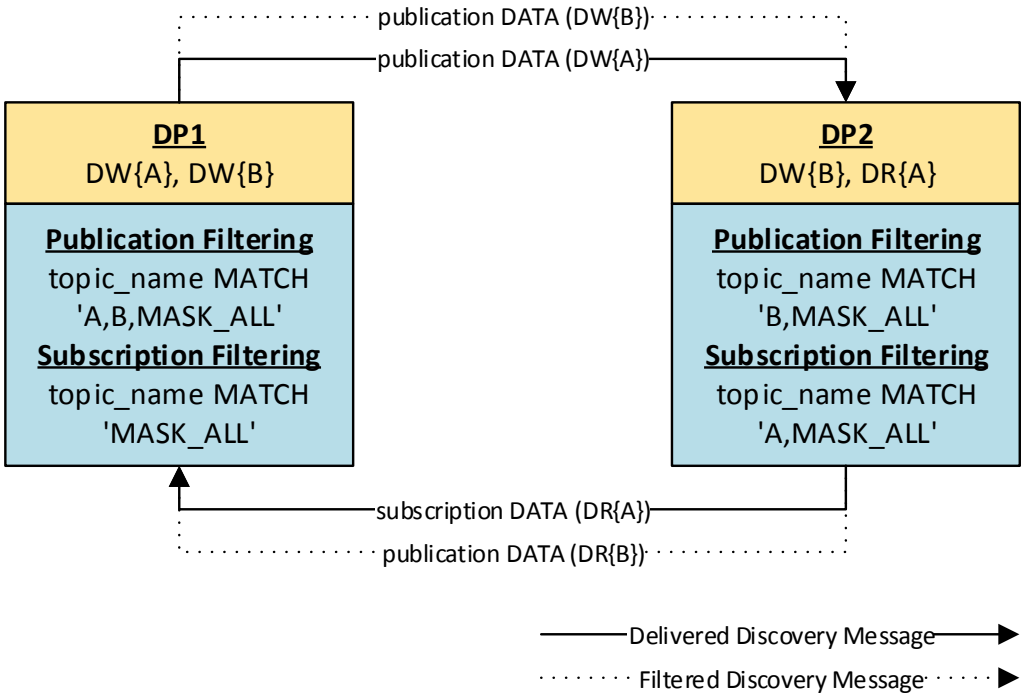

Figure 12: Filtering Discovery Messages by Topic Names

Figure 12 shows an example of filtering discovery messages based on topic names. The following notations are used in the figure.

- DP*x* - Domain Participant named *x*

- Discovery DB - Internal in-memory database stored at the core level of the DDS middleware.
- DW{y} - DataWriter publishing a topic y
- DR{z} - DataReader subscribing for a topic z
- DPx[DW{y}, DR{z}] - Discovery object indicating a Domain Participant containing DW{y} and DR{z} stored in the Discovery DB.

DPs using SDP disseminate all the discovery messages for created endpoints to other DPs in the same domain. Our CFDP approach, however, filters out unmatched discovery messages on the DataWriter side utilizing CFTs by harnessing topic names of local endpoints. This avoids unwanted messages being sent over the network.

In this example, *DP1* does not forward any discovery messages about *DW{B}* to *DP2* since *DP2* does not have *DR{B}*. Likewise, *DP2* does not send discovery messages for *DW{B}* to *DP1*. Discovery messages about unmatched endpoints in CFDP are filtered out on the announcing side and implemented via filtering expressions defined in each participant.

The publication filtering expression in *DP1* (*topic_name MATCH 'A,B,MASK_ALL'*) means that it only accepts subscription discovery messages for topics *A* and *B*. Here, *topic_name* is a variable defined in the built-in topic for the discovery process. *MATCH* is a reserved relational operator for a CFT expression. If a value of the variable is matched with the right-hand operator (*'A,B,MASK_ALL'*), a sample containing the value is accepted by the participant. Likewise, the subscription filtering expression does not allow any publication discovery messages by using a reserved filter, *MASK_ALL*, because DataReaders do not exist.

Figures 13 and 14 compare SDP and CFDP by showing an example having the same application topology. In this comparison, we assume that unicast is used for the discovery protocol. In Figure 13, each participant contains DataWriters and DataReaders with topic

names ranging from *A* to *D*. Every participant publishes SEDP messages of endpoints to other participants in the same domain, and participants receiving the discovery messages store the messages as discovery objects in internal database. Six discovery objects including discovery objects of local endpoints are stored in each participant resulting in a total of 18 discovery objects consuming memory resources in this system. Likewise, there are a total of 18 network transfers.

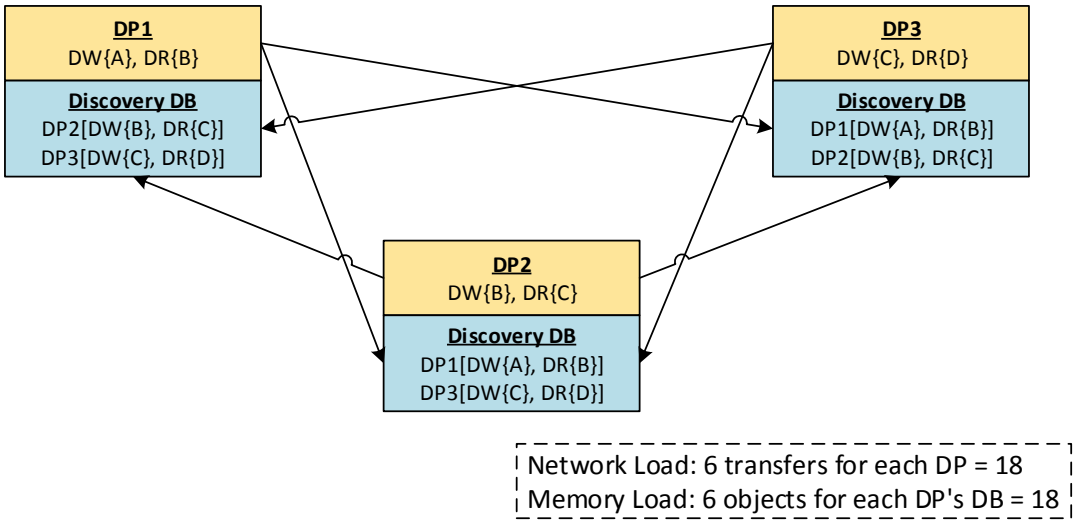

Figure 13: SDP Example

Figure 14 shows a case using CFDP, which has the same setup and topology used in the SDP example. However, in this case, each participant filters discovery messages based on topic names and endpoint types, so it transfers and stores only the required discovery messages. As a result, a total of ten discovery objects are stored in each local database resulting in ten network transfers. This comparison demonstrates that CFDP can conserve memory and network resources.

Algorithm 1 describes the pseudo code for the event callback functions for CFDP.

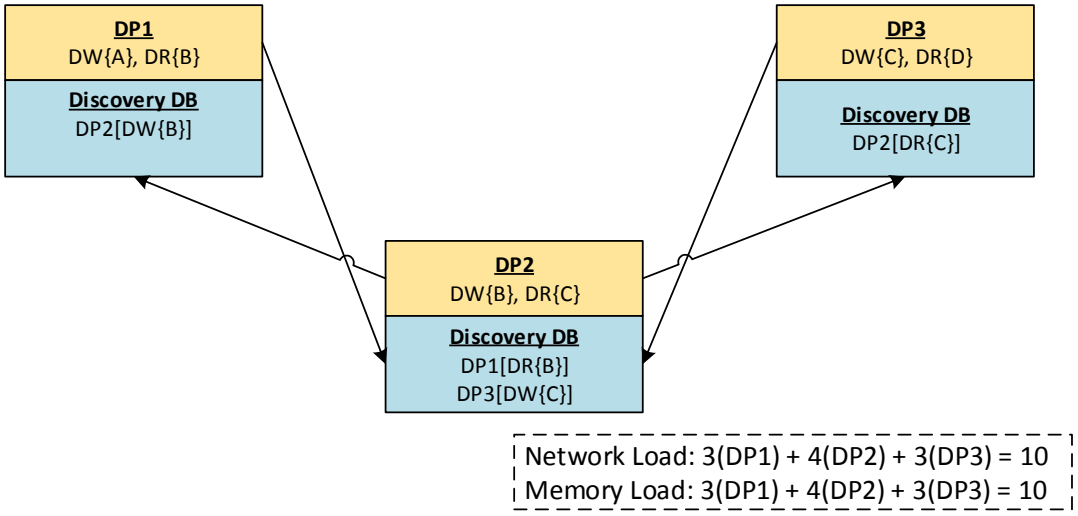

Figure 14: CFDP Example

The callback function is invoked by the pluggable discovery framework described in Section II.3.3.1, which we have used in our solution.

Each callback function is invoked when the following events occur:

- *LocalEndpointEnabled* - when a local endpoint (DataWriter or DataReader) is created
- *LocalEndpointDeleted* - when a local endpoint (DataWriter or DataReader) is deleted
- *RemoteDataWriterReceived* - when a remote DataWriter is created
- *RemoteDataReaderReceived* - when a remote DataReader is created

A discovery object of a created local endpoint is delivered as a parameter value to the *LocalEndpointEnabled* callback function. When the discovery object is for a local DataWriter, the function adds the DataWriter's topic name to the subscription filtering expression(*SubFiltering*). The subscription filtering expression is used for finding matching remote DataReaders for local DataWriters.

Algorithm 1 CFDP Callback Function Algorithms

function LOCAL ENDPOINT ENABLED(EP)
  if EP_{type} *is DataWriter* **then**
    if $EP_{topicName} \in SubFiltering$ **then**
      Add $EP_{topicName}$ *to SubFiltering*
  else if EP_{type} *is DataReader* **then**
    if $EP_{topicName} \in PubFiltering$ **then**
      Add $EP_{topicName}$ *to PubFiltering*
  Send EP to remote Participants
function LOCAL ENDPOINT DELETED(EP)
  if EP_{type} *is DataWriter* **then**
    if $EP_{topicName} \in SubFiltering$ **then**
      Delete $EP_{topicName}$ *from SubFiltering*
  else if EP_{type} *is DataReader* **then**
    if $EP_{topicName} \in PubFiltering$ **then**
      Delete $EP_{topicName}$ *from PubFiltering*
function REMOTE DATA WRITER RECEIVED(EP_{DW})
  Assert EP_{DW} *into Internal DB*
function REMOTE DATA READER RECEIVED(EP_{DR})
  Assert EP_{DR} *into Internal DB*

Similarly, in the case for finding matching remote DataWriters for local DataReaders, the publication filtering expression (*PubFiltering*) is used and must be updated with a topic name of a local DataReader if it does not exist. After updating topic names in the filtering expressions, it sends the discovery objects to other participants to let them know about the newly created local endpoints.

When a local endpoint is deleted, the topic name of the local endpoint is removed from the relevant filtering expressions in the callback function *LocalEndpointDeleted*. The callback functions for remote endpoints (*RemoteDataWriterReceived* and *RemoteDataReaderReceived*) insert discovery objects of remote endpoints to the internal database. The underlying DDS middleware then executes the matching process by comparing data types and QoS policies stored in the discovery objects to establish communication paths between endpoints.

II.3.2 Analysis of SDP and CFDP Complexity

We performed a complexity analysis of SDP and CFDP to determine the expected number of transmitted discovery messages and stored discovery objects. Tables 1 and 2 show notations and metrics used for the complexity analysis, respectively.

Table 1: Notation used for Complexity Analysis

Notation	Definition
P	Number of participants in a domain
E	Number of endpoints in a domain
F	Number of endpoints per participant, $Fanout = E/P$
R	Ratio of the number of matching endpoints to the number of endpoints in a participant, $0 \leq R \leq 1$

Table 2: Metrics used for Complexity Analysis

Metric	Definition
$N_{multi\_participant}$	Number of messages sent/received by a participant using multicast
N_{multi_total}	Total number of messages sent/received by participants using multicast in a domain
$N_{uni\_participant}$	Number of messages sent/received by a participant using unicast
N_{uni_total}	Total number of messages send/received by participants using unicast in a domain
$M_{participant}$	Number of endpoint discovery objects stored in a participant
M_{total}	Total number of endpoint discovery objects stored in a domain

The notations use the number of participants and the number of endpoints in a domain because these variables are crucial to measuring the performance of discovery protocols. A ratio of matching endpoints in each participant can also be used for CFDP complexity analysis because it is the primary factor affecting network transfers and stored objects of

CFDP. For analysis metrics, we decided to inspect the number of transferred and stored discovery messages per participant, as well as per domain (entire set of participants) for both unicast and multicast.

II.3.2.1 SDP Complexity Analysis

In the case of multicast-enabled SDP, the number of messages sent from a participant is the number of endpoints divided by participants, which yields the number of endpoints in a participant, E/P. Regardless of the number of participants and endpoints in a domain, if multicast is used, a message is sent to other participants only when a local endpoint is created. The delivery of a message to multiple destinations (one-to-many communication) is performed by a network switch.

The number of messages received by a participant is the number of endpoints except for local endpoints in a participant ($\frac{E}{P}$ in Equation (1)). The number of messages sent from a participant is the number of local endpoints multiplied by the number of remote participants ($\frac{E}{P} \cdot (P-1)$ in Equation (1)). As a result, in Equation (2), $\frac{E}{P}$ is cancelled out and therefore the sum of sent and received number of messages per participant can be simplified to the number of endpoints in a domain, E in Equation (3). The total number of transfers in a domain is the number of transfers per participant multiplied by the number of participants in a domain, which is $E \cdot P$ as shown in Equation (4).

$$N_{multi\_participant} = \frac{E}{P} + \frac{E}{P} \cdot (P-1) \tag{II.1}$$

$$= \frac{E}{P} + E - \frac{E}{P} \tag{II.2}$$

$$= E \tag{II.3}$$

$$N_{multi\_total} = E \cdot P \tag{II.4}$$

Unicast-enabled SDP incurs more overhead when it sends discovery messages because it handles one-to-many data distribution by itself, rather than using network switches. Each participant disseminates the discovery message as many times as the number of endpoints without including local endpoints ($\frac{E}{P} \cdot (P-1)$ in Equation (5)). The number of received messages is the same as when using multicast ($\frac{E}{P} \cdot (P-1)$ in Equation (5)). The total number of network transfers incurred by the unicast enabled SDP is 2 times $\frac{E}{P} \cdot (P-1)$, and therefore $2 \cdot \frac{E}{P} \cdot (P-1)$, as shown in Equation (6).

$$N_{uni\_participant} = \frac{E}{P} \cdot (P-1) + \frac{E}{P} \cdot (P-1) \tag{II.5}$$

$$= 2 \cdot \frac{E}{P} \cdot (P-1) \tag{II.6}$$

$$\because (P-1) \sim P \tag{II.7}$$

$$\sim 2 \cdot E \tag{II.8}$$

$$N_{uni\_total} \sim 2 \cdot E \cdot P \tag{II.9}$$

In the asymptotic limit (i.e., very large P), as shown in Equation (7), $(P-1) \sim P$, and hence it can be approximated as $2 \cdot E$ in Equation (8), and accordingly the total number of transfers in a domain is roughly $2 \cdot E \cdot P$ as shown in Equation (9).

Unicast-enabled SDP incurs more overhead (which grows as a factor of E) compared to multicast-enabled SDP since every participant in a domain must send more messages since it uses point-to-point unicast data dissemination instead of the one-to-many multicast dissemination. SDP keeps all discovery objects of endpoints in the same domain, and therefore the memory consumption per participant for SDP is directly tied to the number of endpoints in a domain, E in Equation (10). The total memory capacity used by all

participants in a domain is thus $E \cdot P$, as shown in Equation (11).

$$M_{participant} = E \tag{II.10}$$

$$M_{total} = E \cdot P \tag{II.11}$$

II.3.2.2 CFDP Complexity Analysis

We applied the same analysis as SDP to measure the number of transfers for CFDP, *i.e.*, a sum of received and sent network transfers. There is no change in the number of sent messages because each participant with multicast sends one message for each corresponding endpoint it contains ($\frac{E}{P}$ in Equation (12)). We can reduce the number of received messages, however, since only matched discovery messages are received by the filtering mechanism. This factor is therefore multiplied by the ratio of matching endpoints, R (for all, $0 \leq R \leq 1$). Hence, the number of received messages can be $\frac{E}{P} \cdot (P-1) \cdot R$ as shown in Equation (12). From Equation (12) to Equation (13), $\frac{E}{P} \cdot (P-1) \cdot R$ can be transitioned to $E \cdot R - \frac{E}{P} \cdot R$. Then, $\frac{E}{P} - \frac{E}{P} \cdot R$ in Equation (13) can be $\frac{E}{P} \cdot (1-R)$, and thus simply be $F \cdot (1-R)$ in Equation (14) because $\frac{E}{P} = F$.

$$N_{multi\_participant} = \frac{E}{P} + \frac{E}{P} \cdot (P-1) \cdot R \tag{II.12}$$

$$= \frac{E}{P} + E \cdot R - \frac{E}{P} \cdot R \tag{II.13}$$

$$= F \cdot (1-R) + E \cdot R \tag{II.14}$$

$$\sim E \cdot R \tag{II.15}$$

$$N_{multi\_total} \sim E \cdot P \cdot R \tag{II.16}$$

As shown above, the number of transfers per participant is approximated by $E \cdot R$ in Equation (15) because $F \cdot (1 - R)$ in Equation (14) can be a very small number in most cases such that it can be ignored. Using the total transfers per participant ($E \cdot R$ in Equation (15)), the total transfers in a domain is $E \cdot P \cdot R$ in Equation (16).

CFDP reduces both the number of sent and received transfers proportional to the matching ratio if unicast is enabled, so both the number of sent and received messages can be $\frac{E}{P} \cdot (P-1) \cdot R$ in Equation (17). Accordingly, the number of transfers per participant is $2 \cdot \frac{E}{P} \cdot (P-1) \cdot R$ as shown in Equation (18). This number can be approximated to $2 \cdot E \cdot R$ shown in Equation (19) via the same analysis used for SDP (See Equation (7)). The total transfers in a domain is thus $2 \cdot E \cdot P \cdot R$, as shown in Equation (20) because it is the total number of transfers per participant ($2 \cdot E \cdot R$) multiplied by the number of participants in a domain (P).

$$N_{uni\_participant} = \frac{E}{P} \cdot (P-1) \cdot R + \frac{E}{P} \cdot (P-1) \cdot R \tag{II.17}$$

$$= 2 \cdot \frac{E}{P} \cdot (P-1) \cdot R \tag{II.18}$$

$$\sim 2 \cdot E \cdot R \tag{II.19}$$

$$N_{uni\_total} \sim 2 \cdot E \cdot P \cdot R \tag{II.20}$$

CFDP also decreases memory use as a consequence of reducing the number of remote endpoints by removing unmatched ones. Similarly, CFDP conserves resources proportional to the ratio of matching endpoints for cases per participant as shown in Equation (22), as well as per domain as shown in Equation (23).

$$M_{participant} = \frac{E}{P} + \frac{E}{P} \cdot (P-1) \cdot R \tag{II.21}$$

$$\sim E \cdot R \tag{II.22}$$

$$M_{total} \sim E \cdot P \cdot R \tag{II.23}$$

II.3.3 Implementing CFDP

We prototyped our solution at the application level, rather than make invasive and non-standard changes to the underlying DDS middleware, which is RTI Connext [73]. In particular, our implementation leveraged DDS discovery event callbacks using interfaces provided by the middleware, though our approach could be incorporated inside the middleware itself to optimize performance. Our prototype implementation assumes that when participants are created, users determine which topics are published or subscribed by participants (this decision is usually made when endpoints are created).

We made this assumption in our prototype implementation of CFDP since it uses the DDS TRANSIENT_LOCAL durability QoS for both late joiners and CFTs. When a parameter in a filtering expression of a CFT is changed, however, the changed value is not reflected in the list of late joiners. For example, a peer filters a discovery message of topic x because it does not have any endpoints interested in the topic x. Later when an endpoint interested in the topic x is created in the peer, then the peer should be considered a late joiner, but it is not. If the discovery message for the topic x is already filtered, it is not resent even though the filtering expression is updated with having the interest of topic x.

In addition, the prototype implementation of our CFDP discovery plugin had to address the following two issues:

1. It needs to operate with the DDS middleware core by interchanging discovery events since event information is available on different threads. Section II.3.3.1 describes how we resolved this issue via the Pluggable Discovery Framework (PDF) provided by RTI Connext DDS [73].

2. It needs to have a proper data model according to the DDS RTPS specification for

discovery messages exchanged between remote peers. Section II.3.3.2 describes the data model we devised to address this issue.

II.3.3.1 CFDP Plugin Using the Pluggable Discovery Framework (PDF)

The PDF allows DDS users to develop pluggable discovery protocols, and utilize different discovery approaches under various system environments, such as a network environment with limited bandwidth or high loss rates. The PDF offers callback functions invoked when DDS entities are created, deleted, or changed. It also provides a function to assert a discovery object to internal database of the DDS middleware to delegate managing discovery objects and the matching process to the core level of the middleware. We therefore provided interfaces in PDF between the discovery plugins and associated participants as channels to exchange information in both directions: *local-to-remote* (announcing local entities) and *remote-to-local* (discovering remote entities).

Figure 15 shows the software architecture of our CFDP prototype. In this figure, there are six built-in entities present at the DDS core level. The CFDP uses SDP built-in entities to discover application-level built-in entities. The discovery plugin employs callback functions provided by the PDF to retrieve discovery events from the middleware.

Figure 16 introduces a procedure of discovery with the PDF enabled CFDP when a DataWriter is created.

The PDF also furnishes a function to deliver discovery information from the application level to the core level of the middleware. The CFDP exchanges discovery messages between peers with four built-in entities like SDP, and sets up the same QoS configurations of SDP. For example, to deliver discovery messages in a reliable way, the RELIABLE reliability QoS is selected and the TRANSIENT_LOCAL durability QoS is configured to guarantee durability of discovery messages for late joiners as endpoints usually are not created at the same time.

Interfaces of functions for DataWriter discovery provided by the PDF are shown in

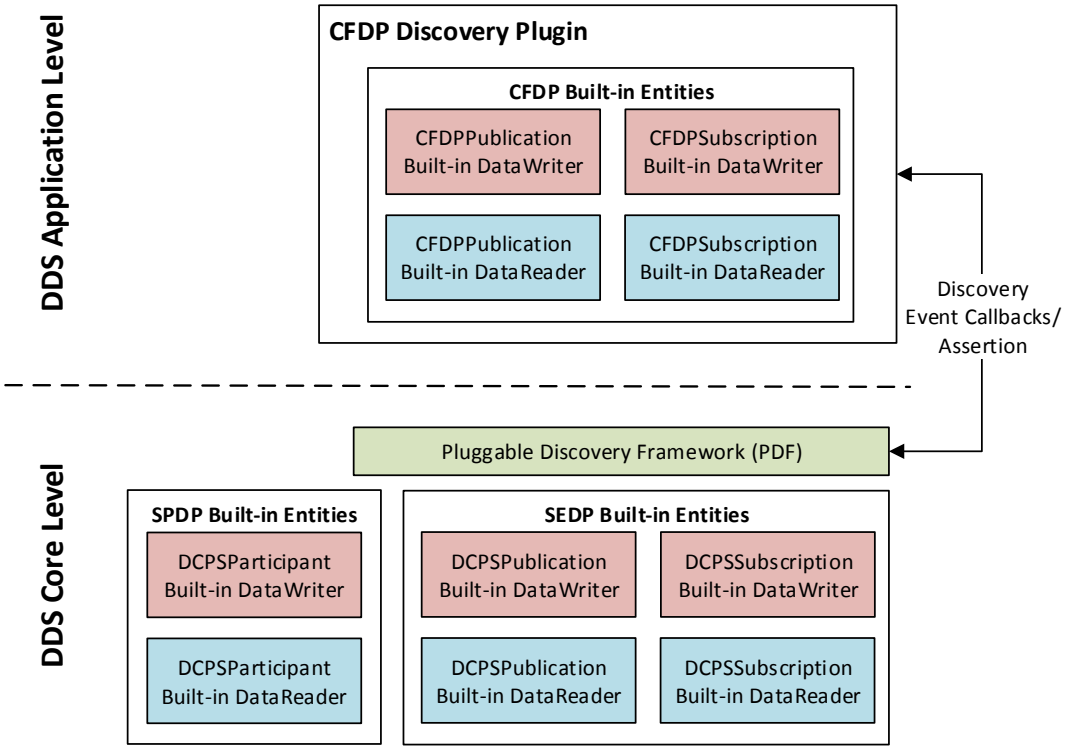

Figure 15: CFDP Prototype Software Architecture

Figure 17. The *NDDS_Endpoint Discovery_AfterLocalDataWriterEnabledCallback* is a callback function invoked when a local DataWriter is created. CFDP uses this function to add a topic name of a created DataWriter to a proper filtering expression and to publish discovery information of the created DataWriter to announce to other remote peers. *NDDS_EndpointDiscovery _assert_remote_datawriter* is a function that asserts a discovery object of a discovered remote DataWriter to the internal database in the underlying middleware.

For example, if a DataWriter is produced in *Peer1* the *AfterLocalDataWriterEnabled-Callback* is invoked and the callback function sends *Publication DATA* (discovery data for DataWriters) to other peers. The *publication DATA* then arrives at *Peer2* and is delivered

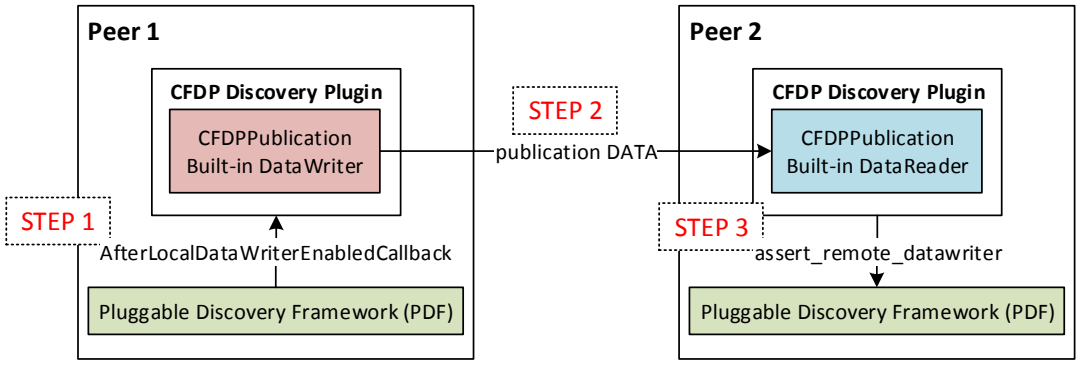

Figure 16: CFDP Sequence with PDF

```
typedef void(* NDDS_EndpointDiscovery_AfterLocalDataWriterEnabledCallback)(
    NDDS_EndpointDiscovery_Plugin *discovery_plugin,
    const struct DDS_PublicationBuiltinTopicData *local_datawriter_data);

NDDS_Discovery_AssertResult_t NDDS_ParticipantDiscovery_assert_remote_participant(
    const NDDS_ParticipantDiscovery_Plugin *discovered_by,
    const struct DDS_ParticipantBuiltinTopicData *remote_participant,
    const struct NDDS_Discovery_Cookie_t *cookie);
```

Figure 17: PDF Functions for DataWriter Discovery

to the core by calling the function *assert_remote_datawriter*. Finally, the DDS middleware establishes the communication by matching topic, type, and QoS policies stored in the *publication DATA*.

II.3.3.2 CFDP Data Model

A data model containing the required information for endpoints is needed to develop discovery built-in entities at the application level. DDS supports diverse QoS configurations and makes the data model for endpoint discovery complicated because QoS configurations for endpoints are exchanged at the endpoint discovery phase. Thus, defining the data model from scratch is hard. We therefore exploited OMG *Interface Definition Language* (IDL) definitions already used in the DDS core middleware and generated source code for

the data model, thereby reducing the time and effort needed to define and implement the data model.

Figures 18 and 19 depict data models for publication and subscription discovery defined in IDL. Attributes for keys to identify discovery entities are contained and *topic_name* and

```
struct PublicationBuiltinTopicData {
  BuiltinTopicKey_t                    key;//@key
  BuiltinTopicKey_t                    participant_key;
  BuiltinTopicKey_t                    publisher_key;
  string                               topic_name;
  string                               type_name;

  DurabilityQosPolicy                  durability;
  DurabilityServiceQosPolicy           durability_service;
  DeadlineQosPolicy                    deadline;
  LatencyBudgetQosPolicy               latency_budget;
  LivelinessQosPolicy                  liveliness;
  ReliabilityQosPolicy                 reliability;
  LifespanQosPolicy                    lifespan;
  UserDataQosPolicy                    user_data;
  OwnershipQosPolicy                   ownership;
  OwnershipStrengthQosPolicy           ownership_strength;
  DestinationOrderQosPolicy            destination_order;

  PresentationQosPolicy                presentation;
  PartitionQosPolicy                   partition;
  TopicDataQosPolicy                   topic_data;
  GroupDataQosPolicy                   group_data;
  TypeConsistencyEnforcementQosPolicy  type_consistency;
};
```

Figure 18: IDL Definition for Publication Discovery Data Model

type_name attributes can be used to find matching endpoints by topics and type structures. The data models include the basic attributes required for the DDS specification, but additional attributes can be added to support an advanced discovery process. For example, the type name can be used for type matching, but an object for type structure can be used to realize extensible and compatible type definitions.

QoS policies required for endpoint matching are also defined in the data models. The QoS policies are different depending on the types of entities (publication or subscription). For example, the time-based filter QoS controls data arrival rates on the DataReader side

```
struct SubscriptionBuiltinTopicData {
  BuiltinTopicKey_t                       key;//@key
  BuiltinTopicKey_t                       participant_key;
  BuiltinTopicKey_t                       subscriber_key;
  string                                  topic_name;
  string                                  type_name;

  DurabilityQosPolicy                     durability;
  DeadlineQosPolicy                       deadline;
  LatencyBudgetQosPolicy                  latency_budget;
  LivelinessQosPolicy                     liveliness;
  ReliabilityQosPolicy                    reliability;
  OwnershipQosPolicy                      ownership;
  DestinationOrderQosPolicy               destination_order;
  UserDataQosPolicy                       user_data;
  TimeBasedFilterQosPolicy                time_based_filter;

  PresentationQosPolicy                   presentation;
  PartitionQosPolicy                      partition;
  TopicDataQosPolicy                      topic_data;
  GroupDataQosPolicy                      group_data;
  TypeConsistencyEnforcementQosPolicy     type_consistency;
};
```

Figure 19: IDL Definition for Subscription Discovery Data Model

even though DataWriters may publish at a faster rate. This QoS policy can therefore be applied only to DataReaders (subscription). Likewise, the lifespan QoS validates how long samples are alive on the DataWriter side, and is only applicable to DataWriters (publication). DDS QoS policies can be reconfigured at run-time, and endpoint discovery messages are used to propagate those changes.

II.4 Experimental Results

This section presents the results of empirical tests we conducted to compare CFDP with SDP. We conducted these tests to evaluate the scalability and efficiency of CFDP over SDP in terms of discovery completion time, which is defined as the duration of the discovery process to locate every matching endpoint in a domain. We measured CPU, memory, and network usage for both CFDP and SDP to determine how discovery completion time is affected by computation and network resource usage.

II.4.1 Overview of the Testbed

Our testbed consists of six 12-core machines. Each machine has a 1 GB Ethernet connected to a single network switch. We implemented our CFDP plugin and the standard SDP implementation with RTI Connext DDS 5.0 [73]. We concurrently created 480 applications for each test, where each application contained a single participant. Specifically, we maintain equal number of publishers and subscribers, each with 20 endpoints (*i.e.*, a data writer or reader, respectively).

All test applications are evenly distributed and executed across the six machines. Each participant has 20 endpoints with an identical entity kind (only DataWriters or DataReaders), and the total number of endpoints in a test is 9,600 (*i.e.*, 480 applications with 20 endpoints each). We set the default matching ratio in each test to 0.1 (10%). It means two out of 20 endpoints in each participant are matched with other endpoints in a domain at a probability of 0.1. Our experiments have tested other matching ratios also.

CFDP uses unicast to filter messages on the DataWriter side because the filtering occurs on DataReader side if multicast is enabled. Filtering on the DataReader side filtering does not reduce the number of discovery messages transferred over a network, so it may have little improvement on performance. SDP uses multicast because it is the default discovery transport for this protocol.

We also developed a test application that measures discovery completion time. The application starts a timer after synchronizing distributed tests in a domain since we remotely execute the applications from a single machine sequentially. It then stops the timer when the expected number of endpoints are discovered. This implementation does not count the discovery time incurred for discovering participants, but measures only the time to discover endpoints because we compare the performance of the endpoint discovery phase (recall that the first phase of the discovery process is common to both SDP and CFDP).

II.4.2 Measuring Discovery Time

Discovery completion time is a critical metric to measure performance of discovery protocols. In the context of our tests, discovery completion time is defined as the time needed to discover all endpoints in a domain. We measured discovery completion time of 480 test applications for CFDP and SDP, respectively. Figure 20 presents the minimum, average, and maximum of discovery completion times for the test applications.

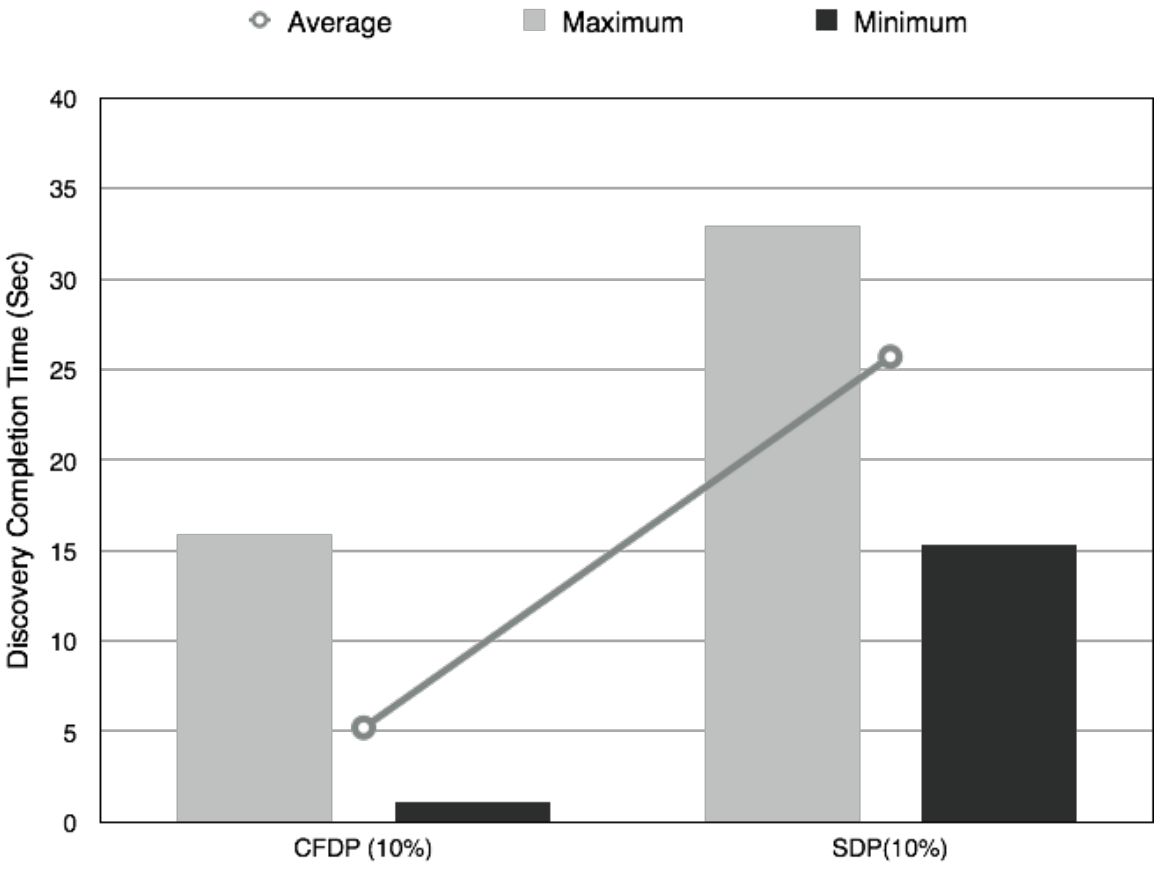

Figure 20: CFDP and SDP Discovery Time Comparison

The minimum discovery time for CFDP is 1.1 seconds and the maximum discovery is 15.92 seconds. Most test applications for CFDP complete the discovery process within 1 to 2 seconds. The average discovery time for all applications is 5.2 seconds. Some of the worst-case results were caused due to CPU saturation as discussed below.

In the case of SDP, the earliest discovery completion time is 15.3 seconds and the latest finish time is 32.9 seconds. The average discovery time is 15.3 seconds. As a result, the maximum discovery time of CFDP is 2 times faster than SDP; the average time is about 5 times faster, and the minimum discovery completion time is 15 times faster.

As mentioned above, the matching ratio used in this experiment is 0.1 (10%). In fact, SDP does not filter any discovery messages. So the different matching ratios make no difference for SDP. The performance of CFDP can be different based on the matching ratio, however, because it is determined by the number of messages to be filtered and its computational complexity.

If CFDP uses unicast as a transport, its performance can sometimes be worse than SDP with a higher matching ratio because it consumes computation resources for the filtering process as well as delivering messages which requires serializing and deserializing messages. CFDP therefore cannot always outperform SDP in a system with a smaller number of topics and endpoints. On the other hand, in large-scale DRE systems with many topics and endpoints, CFDP can always perform the discovery process more efficiently and scalably than SDP. Naturally, if the matching ratios are very high, then the benefit accrued may not be significant but we surmise that in general the matching ratios will remain small.

II.4.3 Measuring Resource Usage

We measured CPU utilization used by the discovery process by exploiting *gnome-system-monitor*. All distributed test applications are executed nearly simultaneously and the discovery process of each test application begins immediately after the endpoints are created. As shown in the Figure 21, however, there is a spike at a specific time range that occurs due to the discovery process overhead.

Figure 21 shows that more CPU cycles of SDP are consumed than for CFDP (10%). SDP's CPU utilization is higher since the number of transferred discovery messages is larger than for CFDP, so it incurs more processing overhead. These results indicate that

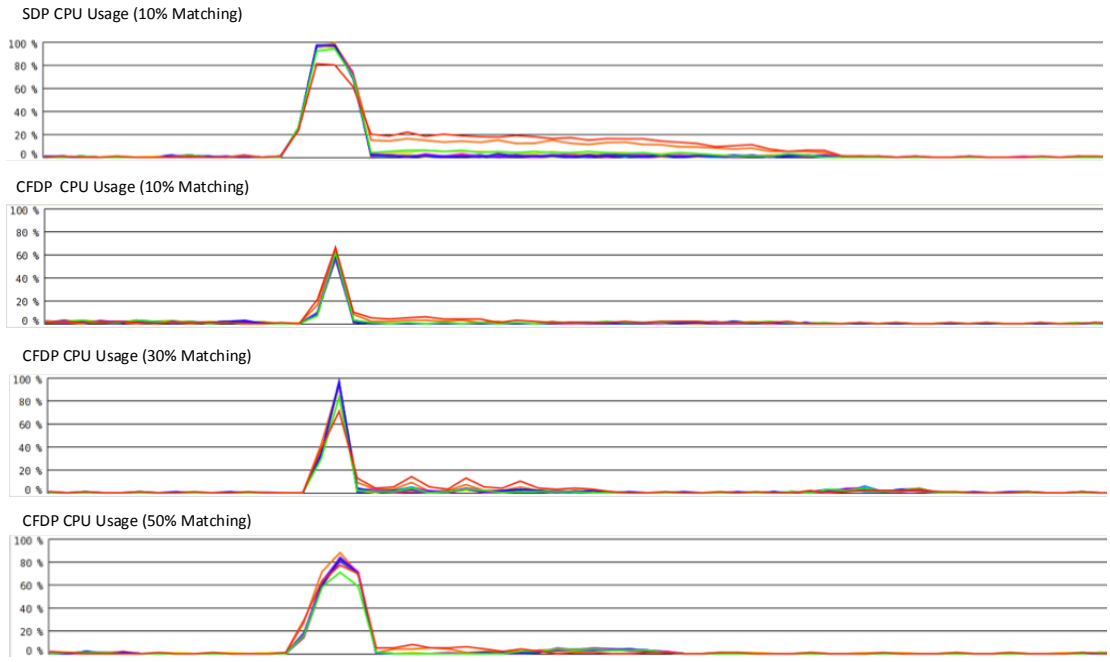

Figure 21: CFDP and SDP CPU Usage Comparison

the processing overhead incurred by filtering messages on DataWriters is lower than the processing costs incurred by transferring discovery messages. We therefore conclude that the CFDP discovery scheme uses fewer CPU resources than SDP.

The processing resources used by CFDP can be different for different matching ratios, as shown by CPU utilization results in the Figure 21. As expected, CPU utilization increases with higher matching ratios. This figure indicates that CFDP is not effective for every case, especially with a high matching ratio. It should therefore be used for large-scale DRE system having many endpoints where the matching ratio is lower than a small-scale system.

To analyze and compare network and memory usage of CFDP and SDP, we counted the number of sent and received messages for the discovery process by each protocol. The DataWriters and DataReaders provide detailed messaging information, such as how many messages and bytes are sent, received, or filtered via the functions named

get_datawriter_protocol_status() and *get_datareader_protocol_status()*. Based on this information, we counted the number of messages, as shown in Figure 22.

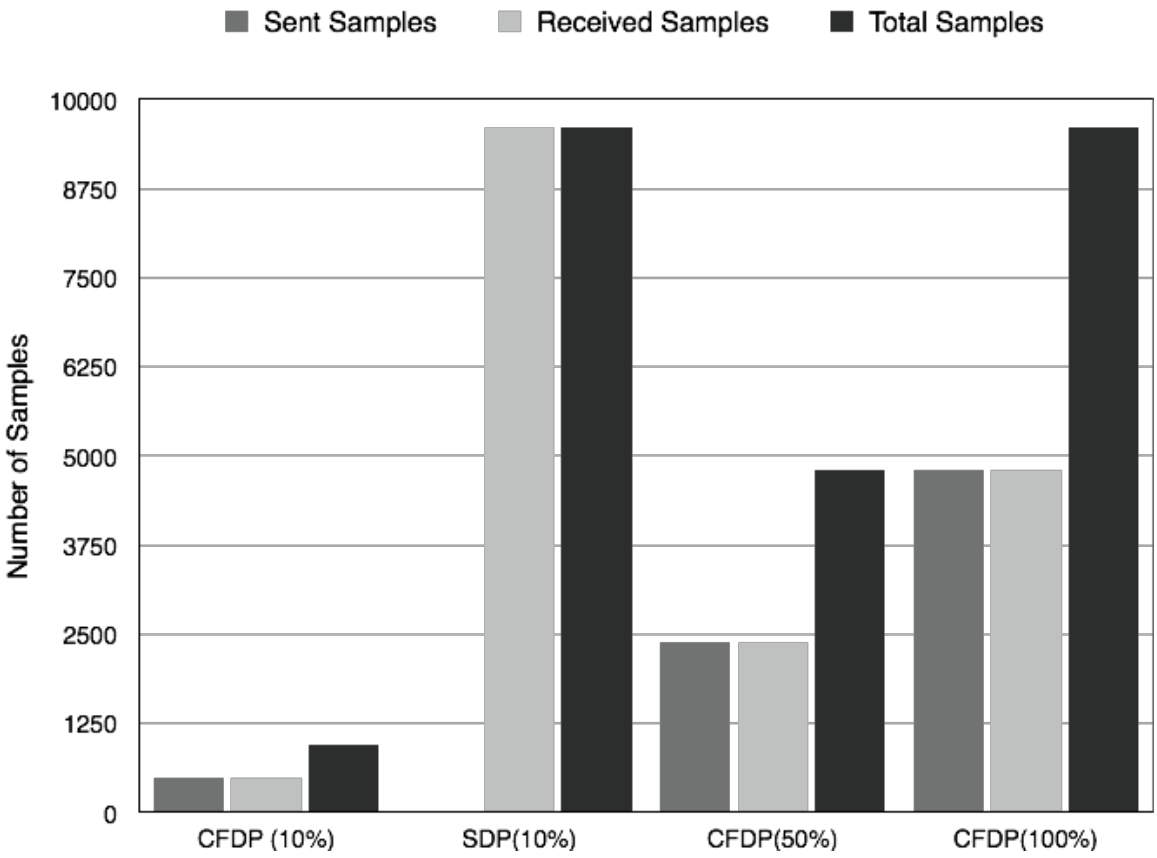

Figure 22: Number of Sent/Received Discovery Messages

CFDP exchanges discovery messages via unicast and each test application creates the same number of endpoints. The number of received and sent messages are therefore identical. CFDP (10%) sent and received 480 samples each and the total was 960.

SDP uses multicast for discovery messages, so the number of received messages is substantially more than the number of sent messages. In this experiment, only 20 samples are sent by a participant as each participant creates 20 endpoints. Although SDP sends fewer messages than CFDP, the total number of messages transferred by a participant using

SDP is still larger than any case of CFDP because the number of received samples is always large.

In this test the number of received transfers for SDP is 9,600, which is 10 times larger than CFDP. Even for 100% matching ratio of CFDP, CFDP uses less messages than SDP because filtering expressions for publication and subscription exist separately, and discovery messages for the same kind of endpoints are automatically filtered as they do not need to communicate.

The size of *publication DATA* is 292 Bytes and the size of *subscription DATA* is 260 Bytes. Based on these sizes, we can calculate memory and network usage approximately. The total transferred bytes of CFDP (10%) is 264.96 KB and the total transferred bytes of SDP is 2808.4 KB. The estimated memory consumed per participant for CFDP is thus 124.8 KB, while SDP requires 2,496 KB.

II.5 Related Work

This section compares and contrasts our CFDP DDS peer discovery mechanism with other discovery mechanisms reported in the literature. Several DDS discovery protocols have been developed to satisfy different system requirements and deployment scenarios. The *Simple Discovery Protocol* (SDP) [61] is the standard DDS discovery protocol. SDP is a decentralized and distributed approach to discover remote peers and their endpoints as each peer independently manages discovery information. It requires no effort in discovery configurations and avoids a single point of failure. The motivation for our work on the CFDP approach is to overcome the limitations of scalability inherent in the standard SDP approach.

OMG DDS also supports a centralized approach [73] [58], which requires a dedicated service to manage all of the participants and endpoints in a domain. This approach can be more scalable than SDP in certain configurations because every peer in a domain need not exchange discovery messages with all other peers in a domain, but only communicate

with the central node where the dedicated service runs. This centralized scheme, however, has several drawbacks. First, its centralized design can become a single point of failure. Second, if the dedicated service is overloaded, the performance of the discovery process of peers in a system can deteriorate considerably.

To avoid the run-time overhead incurred by both the decentralized and centralized discovery protocols, an alternative is a static discovery protocol [72]. In this model users manually configure the discovery information of peers and their endpoints at design- and deployment-time, which requires significant configuration efforts. Such approaches can be useful, however, for closed DRE systems that are deployed in networks with limited resources because no additional resources are used during the run-time discovery process. The key distinction of this static approach with our work on CFDP is that the latter is not a static approach and operates in an open environment.

An improvement to SDP is presented in [75] by utilizing bloom filters to address scalability problems incurred in large-scale DRE systems. The peers send bloom filters to other peers, where the bloom filter is a summary of endpoints deployed in a peer. Peers use the received bloom filters to check if the matching endpoints are in the set represented by the filter. The peers store information about all endpoints, but leverage the smaller size enabled by bloom filters to use network and memory resources more efficiently. Although this related work has similar goals as CFDP (*i.e.*, both approaches attempt to solve the same problem), each approach is designed and implemented differently. In particular, rather than using a bloom filter, CFDP uses a content filter topic (CFT) to filter unmatched endpoint discovery messages.

In [89], the authors investigate the challenges of using DDS in large-scale, network-centric operations and suggest an adaptive discovery service framework to meet key performance requirements. This work also mentions the scalability problem of DDS discovery, but focus on the discovery scalability problems incurred in WANs. Likewise, the work presented by [49] outlines an extension to the IETF *REsource Location And Discovery*

(RELOAD) protocol [39] for content delivery in WAN-based IoT systems using DDS. The authors conducted experiments with 500 to 10,000 peers over a simulated network to show its scalability. Although this paper addressed the discovery scalability issue of DDS, their approach centers on a structured P2P overlay architecture in WANs, which is different from our work on CFDP, which is based on an unstructured P2P scheme.

Discovery is an important issue for the domain of peer-to-peer (P2P) systems, which can be classified into structured P2P and unstructured P2P schemes [51]. The structured P2P scheme, such as Chord [83] and Pastry [69], assigns keys to data and organizes peers into a graph (a distributed hash table) that maps each data key to a peer, and therefore realizes efficient discovery of data using the keys. The standard discovery approach for DDS in LANs based on SDP can be classified as an unstructured P2P scheme since it organizes peers in a random graph by allowing anonymously joining and leaving participants via multicast. The unstructured P2P scheme is not efficient compared to the structured one because discovery messages must be sent to a large number of peers in the network to build a graph of peers that remain anonymous to each other, however, this approach is needed to support the spatio-temporal decoupling of peers. Since our CFDP solution is designed to improve SDP, it operates in the unstructured P2P environment.

A taxonomy that compares and analyzes existing technologies for discovery services in *Ultra-large-Scale* (ULS) systems [54] is presented in [32]. The authors evaluate discovery services along four dimensions: *heterogeneity*, *discovery QoS*, *service negotiation*, and *network scope and type*. To support interoperability between heterogeneous systems, the OMG DDS supports diverse operating platforms (*i.e.* Linux(x86), Windows, and VxWorks), and also has standardized the `Real-TimePublish-Subscribe` (RTPS) protocol [61] for different DDS implementations by vendors. DDS uses built-in DDS entities, such as topics and endpoints, to discover peers, as explained in Section II.2.2. DDS' QoS policies can in turn be used by these entities to support QoS of discovery. For service negotiation, DDS discovery protocols compare the requested and offered QoS policies by

data producers and data consumers at the endpoint discovery phase. DDS discovery mechanisms originally focused on the *Local Area Network* (LAN) scope. Recent research [27] has broadened their scope to support *Wide Area Networks* (WANs) by deploying additional capabilities, such as the DDS Routing Service [74] that transform and filter local DDS traffic to different data spaces (*i.e.*, network domain).

II.6 Concluding Remarks

This paper motivated the need to improve the efficiency and scalability of the standard *Simple Discovery Protocol* (SDP) [61] used for DDS applications. We then presented the design and implementation of our *Content-based Filtering Discovery Protocol* (CFDP), which enhances SDP for a large-scale systems by providing a content filtering mechanism based on standard DDS features to eliminate unnecessary discovery messages for participants according to matching topic names and endpoint types. We also analyzed the results of empirical tests to compare the performance of SDP and CFDP, which indicate that CFDP is more efficient and scalable than SDP in terms of CPU, memory, and network usage. The following is a summary of the lessons we learned from this research and the empirical evaluations.

- **CFDP is more efficient and scalable than SDP**. The experimental results show that CFDP is more efficient and scalable than SDP in terms of discovery completion time, as well as in the use of computing and network resources. In particular, for discovery completion time, CFDP is 5 times faster than SDP on average and minimum discovery completion time is 15 times faster when the matching ratio is 0.1 (10%). For CFDP the computing and network usage linearly decreases as the matching ratio decreases. CFDP therefore disseminates and processes discovery of peers and endpoints more efficiently and scalably than SDP.

- **CFDP's current lack of support for multicast can impede scalability**. Our empirical tests indicated that the number of messages sent by SDP is smaller than those sent by CFDP since CFDP does not use multicast to filter messages on the data publisher side. SDP can seamlessly use multicast because only a single multicast address is needed for all participants to use. For CFDP, however, each content filter used by CFDP will need a separate multicast address. To overcome this limitation, our future work will enhance CFDP to support multicast thereby reducing the number of discovery messages sent by delegating the overhead to network switches. This approach will group peers with a set of multicast addresses by topic names so that built-in discovery DataWriters will publish data only to assigned multicast channels (groups). We also plan to leverage *Multi-channel DataWriters*, which is a DataWriter that is configured to send data over multiple multicast addresses according to the filtering criteria applied to the data [73]. By using this feature, the underlying DDS middleware evaluates a set of filters configured for the DataWriter to decide which multicast addresses to use when sending the data.

- **Instance-based filtering can help to make CFDP scalable in a large-scale system with a small set of topics**. DDS supports a *key* field in a data type that represents a unique identifier for data streams defined in a topic. A data stream identified by a key is called *instance*. The current CFDP filters discovery messages based on topic names, which limits its scalability in a system where most data streams are differentiated by a key of a topic, rather than by a topic itself. For example, a regional air traffic management system may have many endpoints that exchange data by using a single topic (such as flight status), but are interested in only a specific set of flights that are identified by their keys. In such a system, even though all endpoints are involved in the same topic, they do not need to be discovered by each other because its interest is not based on the topic name, but on the key value of the topic. In future work we will enhance CFDP to filter discovery messages based on topic names as

well as instance IDs (keys). This enhancement should provide performance benefits for DRE systems that contain numerous endpoints and instances with a single or less number of topics.

CHAPTER III

A CLOUD-ENABLED COORDINATION SERVICE FOR INTERNET-SCALE OMG DDS APPLICATIONS

III.1 Motivation

Many pub/sub messaging solutions [31, 56, 88] including research efforts [13, 19, 65] exist that can operate in WANs. Some of these even support QoS properties, such as availability [19, 65], configurable reliability [56], durability [31], and timeliness [12, 41]. However, these solutions tend to support only one QoS property at a time and in most cases, support for configurability is lacking. Moreover, dynamic discovery of endpoints, which is a key requirement for IIoT, is often missing in these solutions.

The presence of large amounts of generated data in IIoT motivates the need for data-centric pub/sub with support for configurable, multiple QoS properties. The Object Management Group (OMG)'s Data Distribution Service (DDS) [60] standard for data-centric pub/sub holds substantial promise for IIoT applications because of its support for configurable QoS policies, dynamic discovery, and asynchronous and anonymous decoupling of data endpoints (*i.e.*, publishers and subscribers) in time and space.

However, there still remain many unresolved challenges in using DDS in WAN-based IIoT applications. For instance, DDS uses multicast as a default transport to dynamically discover peers in a system. If the endpoints are located in isolated networks that do not support multicast, then these endpoints cannot be discovered by each other. Secondly, even if these endpoints were discoverable, because of network firewalls and network address translation (NAT), peers may not be able to deliver messages to the destination endpoints.

III.1.1 Challenges

One approach to supporting DDS in WAN-based pub/sub relies on broker-based solutions [28, 50]. It is conceivable to think that these broker-based solutions in conjunction with the data-centric and configurable QoS features provided by DDS can readily make it useful for IIoT. However, this is not the case for the following reasons. IIoT use cases illustrate heterogeneity in the kinds of devices and networks involved, the number and types of data-centric topics of interest that must be managed, and significant number and churn (*i.e.*, joining and leaving) of the endpoints. Thus, a solution for dynamic discovery and QoS-enabled dissemination that can scale to large number of endpoints is desired. Since brokers are necessary to overcome issues with NAT and firewalls, the scalable discovery and dissemination solution desired for IIoT must also provide effective coordination among potentially large number of distributed brokers.

III.1.2 Solution Approach

To fill this gap, we present *PubSubCoord,* which is a cloud-based coordination service for geographically distributed pub/sub brokers to transparently connect endpoints and realize internet-scale data-centric pub/sub systems. To that end, this paper makes the following contributions:

- To address the scalability and low latency requirements of data dissemination across WANs, PubSubCoord introduces a two-level broker hierarchy deployed over a pub-/sub overlay network, which provides a maximum two-hop dissemination path for data across distributed, isolated networks.

- To achieve dynamic discovery and data routing between brokers, PubSubCoord exploits and extends the ZooKeeper coordination service [35] to synchronize dissemination paths for the dynamic network of brokers and endpoints.

- For those dissemination paths that need both low latency and reliability assurances,

PubSubCoord trades off resource usage in favor of deadline-aware overlays that build multiple, redundant paths between brokers.

PubSubCoord preserves the endpoint discovery and data dissemination model of the underlying pub/sub messaging system while adding a two-level broker hierarchy by tunneling discovery and dissemination messages across the broker hierarchy. Our contributions are discussed and demonstrated concretely in the context of endpoints that use the OMG DDS as the pub/sub messaging system, however, the solution is generic and can be used for other pub/sub messaging systems.

III.2 Background

Since we have used the OMG DDS as the concrete pub/sub technology and ZooKeeper as the coordination service to describe PubSubCoord' contributions, this section provides an overview of these underlying technologies.

III.2.1 OMG DDS QoS Policies

OMG DDS supports a number of different QoS policies that can be mixed and matched. Each QoS policy has offered and requested semantics (*i.e.*, offered by publishers and requested by subscribers) and are used in conjunction with the topic data type to match pairs of endpoints, *i.e.*, the DataReader and DataWriter. We briefly describe only those policies that we have used either in the design of PubSubCoord or in our empirical studies.

The ***reliability*** QoS controls the reliability of data flows between DataWriters and DataReaders at the transport level. It can be of two kinds: BEST_EFFORT and RELIABLE. The ***durability*** QoS specifies whether or not the DDS middleware stores and delivers previously published data samples to endpoints that join the network later. The reliability and persistency can be affected by the ***history*** QoS policy, which specifies how much data must be stored in in-memory cache allocated by the middleware. Along with the *history* QoS policy, the ***lifespan*** QoS helps to control memory usage and lifecycle of data by setting

expiration time of the data on DataWriters, so that the middleware can delete expired data from the cache.

The ***deadline*** QoS policy specifies the deadline between two successive updates for each data sample. The middleware will notify the application via callbacks if a DataReader or a DataWriter breaks the deadline contract. Note that DDS makes no effort to meet the deadline; it only notifies if the deadline is missed. The ***liveliness*** QoS specifies the mechanism that allows DataReaders to detect disconnected DataWriters. The ***ownership*** QoS specifies whether it allows multiple DataWriters to write data on a stream simultaneously. If it is set to have an exclusive owner, the exclusive owner is determined by the configured strength of DataWriters. The primary DataWriter with the highest strength is switched to a backup if it violates the *deadline* QoS or is disconnected.

III.2.2 DDS Routing Service

Since PubSubCoord relies on a broker-based architecture, we have leveraged and extended an existing DDS broker solution. Specifically, we have used the DDS Routing Service, which is a content-aware bridge service for connecting geographically dispersed DDS systems [50]. It integrates DDS applications across LANs as well as WANs. DDS Routing Service leverages all the entities of DDS and enables DDS applications to publish and subscribe data across domains in multiple networks without any changes to the applications.

As shown in Figure 23, DDS Routing Service exploits DDS entities (*e.g.*, DataWriter and DataReader) to forward data from a domain to other domains, and therefore it also supports features provided by DDS entities such as a rich set of QoS policies and content-based filtering.

Our solution utilizes DDS Routing Service for brokers to establish data dissemination paths based on routing decisions by coordination logic. As our system is deployed in

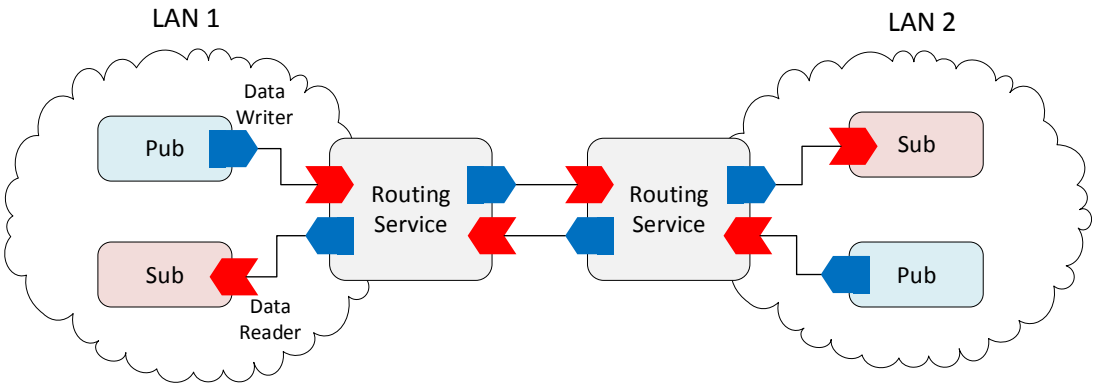

Figure 23: DDS Routing Service

internet-scale WAN environments, we use TCP communications between brokers for reliable data delivery in geographically dispersed networks. DDS Routing Service supports IP and port translation as well as Transport Layer Security (TLS) for security reasons in WANs. DDS Routing Service can be administered remotely by sending control commands (*e.g.*, add peers' locators and create routing paths), so it is easy to set up routing paths in a programmable way in our solution. Each broker in PubSubCoord sends commands to Routing Service based on its coordination algorithms based on event notifications from ZooKeeper servers.

III.2.3 ZooKeeper

ZooKeeper is a service for coordinating processes within distributed applications [35]. The ZooKeeper service consists of an ensemble of servers that use replication to accomplish high availability with high performance and relaxed consistency. ZooKeeper provides the *watch* mechanism to notify a client of a change to a *znode* (*i.e.*, a ZooKeeper data object containing its path and data content). There exist many coordination recipes using ZooKeeper that are often needed for distributed applications, such as leader election, group membership, and sharing configuration metadata. PubSubCoord exploits these capabilities in its design.

III.3 Design and Implementation

This section describes the architecture and design rationale for the PubSubCoord design. We also provide details on the implementation.

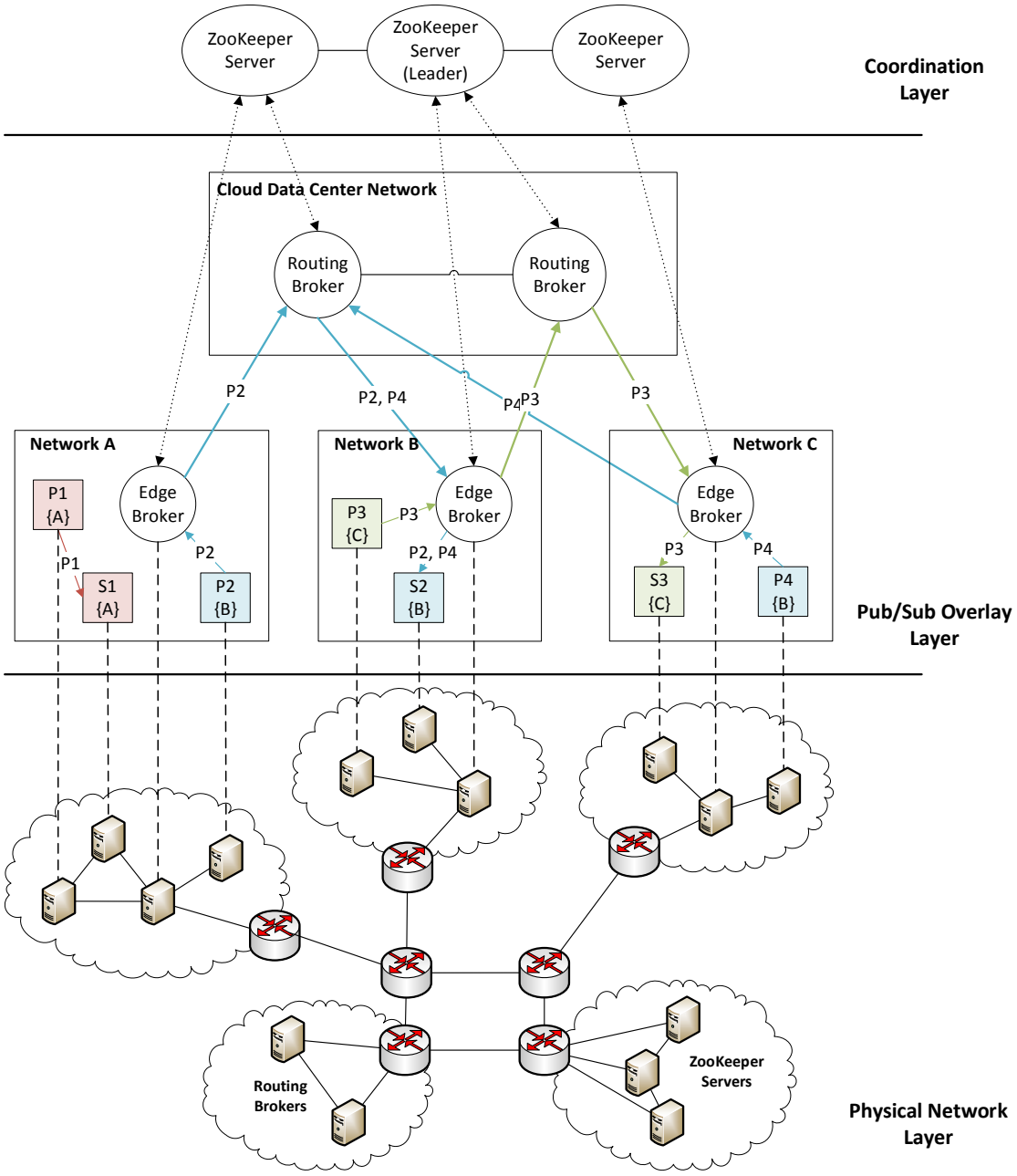

Figure 24: PubSubCoord Architecture

III.3.1 PubSubCoord Architecture

Figure 24 shows the PubSubCoord architecture depicting three layers: a coordination layer, a pub/sub overlay layer, and the physical network layer. The pub/sub overlay comprises the two-level broker hierarchy representing the logical network of brokers and endpoints in a system. An *edge broker* is directly connected to endpoints in a LAN (*i.e.*, an isolated network) to serve as a bridge to other endpoints placed in different networks. A *routing broker* serves as a mediator to route data between edge brokers according to assigned and matched topics that are present in the global data space. The coordination layer comprises an ensemble of ZooKeeper servers used for coordination between the brokers.

The data dissemination in PubSubCoord is explained using an example in Figure 24. P$i\{T\}$ denotes a publisher i that publishes topic T (similarly for a subscriber S). Since there are no endpoints interested in topic A other than publisher $P1$ and subscriber $S1$, they communicate only within the local network A via either UDP-based multicast or unicast for scalability and low latency. $P2$, $P4$, and $S2$ are interested in topic B but are deployed in different networks. So their communications are routed through a routing broker that is responsible for topic B. The network transport protocol between brokers is configurable, but TCP is used as a default transport to ensure reliable communication over WANs. As seen from this example, a maximum of 2 hops on the overlay network are incurred by data flowing from one isolated network to another (*e.g.*, network A to B).

III.3.2 Rationale for PubSubCoord Design Decisions

We now offer a justification for the various design decisions we made in our architecture.

III.3.2.1 2-level Broker Hierarchy and Scalability

Traditional WAN-based pub/sub systems form an overlay network with brokers to which endpoints can be connected. The brokers exchange subscriptions they receive from

subscribers, by which they build routing paths from publishers. The main challenge of this approach is how to build routing states among brokers to route data according to matching subscribers efficiently. To resolve this challenge, our solution clusters brokers by matching topics and routes data through routing brokers responsible for specific topics to minimize the overall number of data exchange and connections between brokers in a system.

In the broker-based pub/sub systems, if a local broker fails, it halts not only a service for endpoints connected to this broker but also service for endpoints connected to other brokers because local brokers can be used as intermediate routing brokers. To overcome these limitations, PubSubCoord is structured by harnessing a two-tier architecture similar to the BlueDove system [46] that classifies brokers into *dispatcher* (similar to edge broker in our solution) and *matcher* (similar to routing broker in our solution).

Having only one routing broker in the top level will be problematic since it cannot handle the substantial routing load stemming from the dissemination of various topic data. On the other hand, multiple layers of hierarchy similar to DNS would have complicated the management of topics and recovery from failures, and could introduce multiple routing hops. For that reason, the top layer comprises a cluster of routing brokers that balance the load among themselves.

Although the edge brokers are always placed at the edge of their respective isolated networks, we had to reason about where to place the routing brokers. We decided to place the routing brokers in the cloud because the cloud enables us to elastically scale the number of routing brokers depending on the load.

III.3.2.2 Need for a Coordination Layer

Although a 2-level broker hierarchy resolves issues with maintaining substantial pub/-sub routing states, we needed an approach that the brokers can form this broker hierarchy and set the connections between the edge and routing brokers. To that end, PubSubCoord incorporates the coordination layer comprising an ensemble of ZooKeeper servers, which

help brokers discover each other and build broker overlay networks using coordination logic.

The data model of ZooKeeper used for coordination of distributed processes is structured like a file system in the form of znodes. This hierarchical namespace is actually meant to manage group membership, however, we repurpose it to manage pub/sub endpoints that are grouped by topics. Figure 25 shows the znode data tree structure of PubSubCoord. The root znode contains three znodes: *topics*, *leader*, and *broker*. The *topics* znode contains children znodes for every unique topic that has endpoints interested in it, which in turn become the children of the specified topic znode. The *leader* znode is used to elect a leader among routing brokers. The *broker* znode has children znodes for each routing broker where its locator information (*i.e.*, IP address and port number of a routing broker) is stored. The leader uses this information to associate a selected routing broker's locator to a topic znode after the topic assignment.

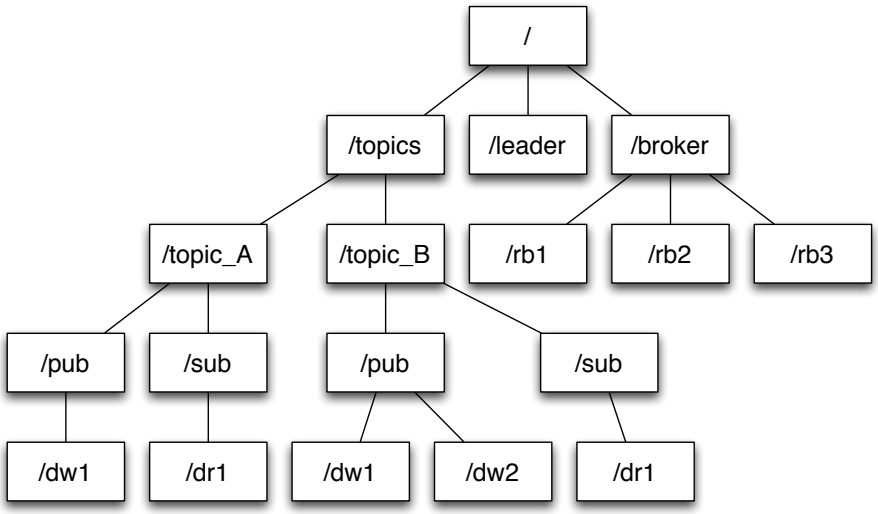

Figure 25: PubSubCoord ZNode Data Tree Structure

Brokers connect to the coordinating servers as clients and create, update, and delete znodes in stored in the servers. They also set watches on interesting znodes to receive notifications (*e.g.*, broker join/leave). ZooKeeper provides different modes for znode: *ephemeral* and *persistent*. A znode with ephemeral mode is automatically deleted when a session of a client that creates the znode is lost. We utilize this ephemeral mode to manage events when brokers join or leave our system.

III.3.2.3 Load Balancing and Fault Tolerance

To achieve load balancing at the routing broker layer, the cluster of routing brokers elect a leader. To elect a leader in a consistent manner, PubSubCoord uses ZooKeeper's *leader* znode for routing brokers to write themselves on the znode so as to be elected as a leader (*i.e.*, voting process). The routing broker that gets to write first becomes a leader since the znode is locked thereafter (*i.e.*, no one can write on the znode unless the leader fails).

When an endpoint is created with a new topic, an edge broker informs ZooKeeper of the new topic which inserts it into its znode tree and informs the leader routing broker of the new topic. The routing broker leader selects one of the existing routing brokers to handle that topic. This selection is made based on the load on each routing broker.

If a routing broker fails, the leader reassigns topics handled by that failed broker to another routing broker to avoid service cessation. If the load is too high, the cloud will elastically scale the number of routing brokers. If a leader fails, the routing brokers vote for another leader again. On assignment or failure and reassignment of routing broker, ZooKeeper notifies the appropriate edge brokers to update their paths to the right routing broker.

To provide a scalable and fault-tolerant service at the coordination layer, multiple ZooKeeper servers can exist as a quorum, and a leader of the quorum synchronizes data between distributed servers to provide consistent coordination events to clients (*i.e.*, brokers in our solution) and avoid single points of failure.

III.3.2.4 Deadline-aware Overlay Optimizations

PubSubCoord also supports an optimization to both improve reliability and latency by providing an additional one hop path over the overlay that directly connects communicating edge brokers. Figure 26 illustrates the concept. These optimizations can be leveraged by pub/sub streams that require stringent assurances on reliable and deadline-driven data delivery.

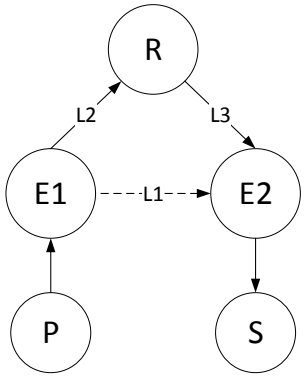

Figure 26: Multi-path Deadline-aware Overlay Concept

To achieve this feature, PubSubCoord exploits the capabilities of the underlying pub/-sub messaging system. To that end, we use the deadline values configured by DDS' *deadline* QoS. Recall that this parameter is used to express the maximum duration of a sample to be updated. For those event streams requiring strict deadlines, multi-path overlay networks build an alternative, additional path directly between edge brokers thereby reducing the number of hops to just one.

III.3.3 Broker Interactions

In this section we describe how the brokers interact and the actual process of updating their internal states used in routing the streamed pub/sub data. Routing brokers can be divided into two kinds: leader routing broker and worker routing broker. A leader routing

broker manages the cluster of routing brokers and assigns topics to workers in a way that balances the load. Worker routing brokers relay pub/sub data between edge brokers. The leader routing broker can also serve as a worker routing broker.

Figure 27 presents the sequence diagram showing the interactions of the routing brokers. Each routing broker initially connects to the ZooKeeper servers as a client. The cluster of routing brokers subsequently elect a leader among themselves. The leader routing broker registers a listener (*i.e.*, event detector that is notified when the registered znode changes) on the *topics* znode (shown in Figure 25) to receive topic relevant events (*e.g.*, creation or deletion of topics). For example, as shown in Figure 27, when *TopicA* is created, the leader assigns the topic to the least loaded worker, which currently is decided based on the number of adopted topics by that worker. However, other strategies can also be used in the load balancing decisions (*e.g.*, least loaded based on CPU utilization or the number of connections). Next, the leader updates a locator of the assigned worker broker on the corresponding znode that is created for *TopicA*, *i.e.*, a child of *topics* znode – see the leftmost node in row three of Figure 25. This locator information will then be used by edge brokers interested in *TopicA*.

A worker routing broker initially registers listeners on a znode for itself (*i.e.*, a child of *broker* znodes) to receive topic assignment events, which occur when the assigned topics znode is updated by a leader routing broker. When the worker routing broker is informed that it must handle a specific topic, such as *TopicA*, it then registers a listener on pub/sub znodes for that particular assigned topic (*e.g.*, children of *topic_A* znode) to receive endpoint discovery events, such as creation of publisher or subscriber endpoints interested in *TopicA*. When an endpoint for *TopicA* is created and the worker routing broker is notified, it establishes data dissemination paths to edge brokers. For this data dissemination, PubSubCoord relies on the underlying pub/sub messaging systems' broker capabilities, such as the DDS Routing Service we have leveraged in our work.

Figure 28 shows the corresponding sequence diagram for edge brokers. Like routing

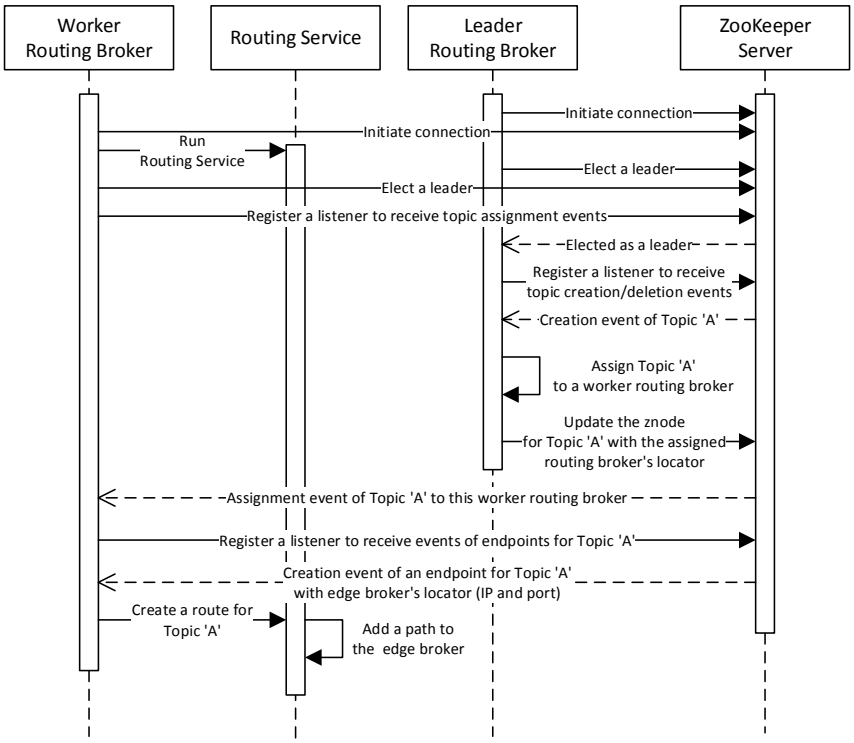

Figure 27: Routing Broker Sequence Diagram

brokers, edge brokers initially connect to ZooKeeper servers as clients. Edge brokers make use of *built-in entities* (*i.e.*, special pub/sub entities for discovering peers and endpoints in a network supported by the underlying pub/sub messaging system) to discover endpoints in local networks. For example, when a pub or sub endpoint interested in *TopicA* is created, built-in entities receive discovery events via multicast, and then edge brokers create znodes for the created endpoints.

Edge brokers register a listener on a topic znode (*e.g.*, *topic_A* in Figure 25) in which the created endpoint is interested in to obtain the locator for the routing broker that is in charge of that particular topic. Once a locator of a routing broker is obtained, an edge broker initiates a data dissemination path to the routing broker through the Routing Service. If the created endpoints move to different networks or are deleted, a timeout event occurs

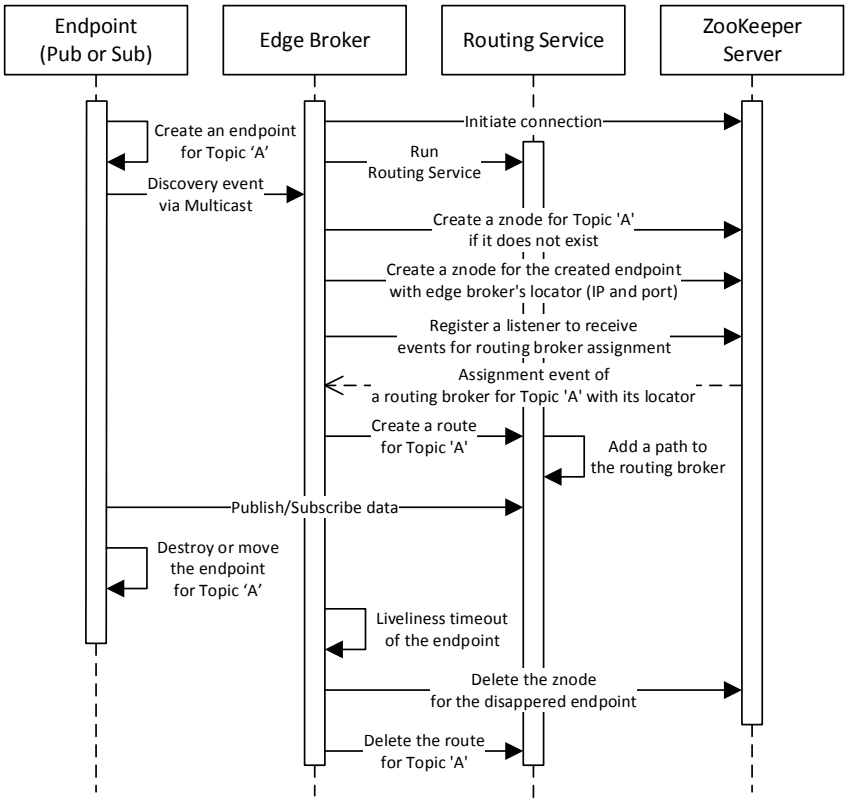

Figure 28: Edge Broker Sequence Diagram

by virtue of using the *liveliness* QoS (*i.e.*, a DDS QoS policy used to detect disconnected endpoints where the timeout values are configurable) and accordingly the znodes for endpoints are deleted from coordination servers and a route created in Routing Service is also terminated. Thus, mobility of publisher and subscriber endpoints is also supported by the PubSubCoord design.

III.3.4 Broker Implementation Details

In this section we describe some implementation details that use the OMG DDS pub/sub messaging system as the underlying pub/sub technology and ZooKeeper. Algorithms 2 and 3 describe the pseudo code of callback functions implemented in edge brokers and

routing brokers, respectively. Callback functions are invoked by either the DDS endpoint discovery events in built-in DDS DataReaders (*i.e.*, built-in entities) or notifications by ZooKeeper services.

III.3.4.1 Edge Broker Implementation

Algorithm 2 Edge Broker Callback Functions

```
function ENDPOINT CREATED(ep)
    create_znode (ep)
    if ! topic_multi_set.contains(ep_topic) then
        ep_node_cache = create_node_cache(ep)
        set_listener (ep_node_cache)
        routing_service.create_topic_route(ep)
    topic_multi_set.add(ep_topic)
function TOPIC NODE LISTENER(topic_node_cache)
    rb_locator = topic_node_cache.get_data()
    if ! rb_peer_list.contains(rb_locator) then
        rb_peer_list.add(rb_locator)
        routing_service.add_peer(rb_locator)
function ENDPOINT DELETED(ep)
    delete_znode (ep)
    topic_multi_set.delete(ep_topic)
    if ! topic_multi_set.contains(ep_topic) then
        delete_node_cache(ep)
        routing_service.delete_topic_route(ep)
```

Each callback function for edge brokers is invoked when the following events occur:

- ENDPOINT CREATED: This function is invoked when an endpoint in a network is created and activated by a built-in DDS DataReader.
- TOPIC NODE CACHE LISTENER: This function is invoked when a topic znode managed by an edge broker is updated with a locator of an assigned worker routing broker. It is activated by a ZooKeeper client process.

- ENDPOINT DELETED: This function invoked when an endpoint in a network is deleted and activated by a built-in DDS DataReader.

We use Curator[1], which is a high-level API that simplifies using ZooKeeper, and provides useful recipes such as leader election and caches of znodes. We use the cache recipe to locally reserve data objects accessed multiple times for fast data access and reducing loads on ZooKeeper servers.

The ENDPOINT CREATED callback function first creates a znode for a created endpoint (*i.e. ep* in Algorithms 2) that contains the topic name, type, QoS settings. If a relevant topic to the created endpoint has not appeared in an edge broker before, a cache for the topic znode and its listener for the topic are created to receive locator information of an assigned routing broker. When the znode for the topic is updated by a leader routing broker, it triggers the TOPIC CACHE LISTENER callback described in Algorithm 2.

In the TOPIC NODE LISTENER callback function, each topic znode stores a locator of a routing broker which is responsible for the topic. The locator of a routing broker is added to DDS Routing Service to establish a communication path from the edge broker to a routing broker.

The ENDPOINT DELETED callback function deletes the znode for the existing endpoint, and deletes it from the multi-set for topics. Next, it checks if the multi-set contains the topic of the deleted endpoint. If the topic is contained in the multi-set, it means other endpoints are still interested in the topic. If it is empty, it means no endpoints that are interested in the topic exists, and that the cache and its listener need to be removed. The multi-set data structure for topics is used because there may still exist endpoints interested in topics relevant to deleted endpoints.

III.3.4.2 Routing Broker Implementation

[1] `http://curator.apache.org`

Algorithm 3 Routing Broker Callback Functions

```
function BROKER NODE LISTENER(broker_node_cache)
    topic_set = broker_node_cache.get_data()
    for topic : topic_set do
        if ! topic_list.contains(topic) then
            ep_cache = create_children_cache (topic)
            set_listener(ep_cache)
            topic_list.add(topic)
function ENDPOINT LISTENER(ep_cache)
    ep = ep_cache.get_data()
    switch ep_cache.get_event_type() do
        case child_added
            if ! eb_peer_list.contains(ep_eb_locator) then
                eb_peer_list.add(ep_eb_locator)
                routing_service.add_peer(ep_eb_locator)
            if ! topic_list.contains(ep_topic) then
                routing_service.create_topic_route(ep)
            topic_multi_set.add(ep_topic)
        case child_deleted
            topic_multi_set.delete(ep_topic)
            if ! topic_multi_set.contains(ep_topic) then
                eb_peer_list.delete(ep_eb_locator)
                routing_service.delete_topic_route(ep)
```

Each callback function for routing brokers is invoked when the following events occur:

- BROKER NODE LISTENER - This function is invoked when a znode for a worker routing broker is updated with an assigned topic by a leader routing broker and activated by a ZooKeeper client process.

- ENDPOINT LISTENER - This function is invoked when children pub/sub endpoints of a znode for an assigned topic is created, deleted, or updated. It is activated by a ZooKeeper client process.

Every routing broker registers a listener on the znode for itself to receive topic assignment events updated by a leader routing broker. In the BROKER CACHE LISTENER callback function, the znode for the routing broker stores a set of topics. When the topic set is updated by the leader (*e.g.*, the leader assigns a new topic to the worker routing broker), and it applies the changes by creating a cache for the assigned topic and its listener to receive events relevant to endpoints interested in the assigned topic.

When an endpoint is created or deleted, the edge brokers create or delete znodes for endpoints and these events will trigger the ENDPOINT CACHE LISTENER function in routing brokers that are responsible for topics involved with the endpoints. The data of znode cache for an endpoint (*ep* in the ENDPOINT CACHE LISTENER callback function) contains the locator of an edge broker where the endpoint is located as well as the topic name, type, and QoS settings.

If the event type is creation, it adds the locator of the edge broker to the DDS Routing Service running in the routing broker if it does not exist. Thereafter, it requests the DDS Routing Service to create a route for the topic based on the information provided by the content of the *ep* znode from this routing broker to the edge broker, if it does not exist. If the event type is deletion, it has to delete the locator and the topic route from the DDS Routing Service on the condition that no endpoints for that topic still exist.

III.4 Experimental Results

This section presents the experimental results we conducted to evaluate scalability and validate deadline-aware overlays of PubSubCoord.

III.4.1 Overview of Testbed Configurations and Testing Methodology

Our testbed is a private cloud managed by OpenStack comprising 60 physical machines each with 12 cores and 32 GB of memory. To experiment with a WAN-scale environment, our cloud platform uses Neutron[2], an OpenStack project for networking as a service, that allows users to create virtual networks by using a Open vSwitch plugin[3]. For our experiments, we created 120 virtual networks, and 380 virtual machines (VMs) are placed across these virtual networks. Each VM is configured with one virtual CPU and 2 GB of memory. We use RTI Connext 5.1[4] as the implementation of the DDS Routing Service and for our test applications.

Our experiments use the *reliability* and *durability* DDS QoS policies for pub/sub communications to illustrate experimental results for higher service quality in terms of reliability and persistence of data delivery. Depending on the systems' requirements, QoS policies can be varied and performance results may change according to the different QoS settings. Specifically, we use RELIABLE *reliability* QoS to avoid data loss in a transport level through data retransmission. We use KEEP_ALL *history* QoS to keep all historical data and TRANSIENT *durability QoS* to make it possible for late-joining subscribers to obtain previously published samples. The *lifespan* QoS is set to 60 seconds so publishers guarantee persistence for 60 seconds.

To evaluate our solution, we measure end-to-end latency from publishers to subscriber, and CPU usage on brokers for scalability of data dissemination. CPU usage is shown along with latency to understand how different settings, *i.e.*, number of topics per network and

[2] https://wiki.openstack.org/wiki/Neutron

[3] http://www.openvswitch.org

[4] https://community.rti.com/rti-doc/510/ndds.5.1.0/doc/pdf/RTI_CoreLibrariesAndUtilities_UsersManual.pdf

number of routing brokers, affect dissemination scalability. Moreover, we measure latency of coordination requests and the number of data objects and notifications on ZooKeeper servers to show coordination scalability. To measure end-to-end latency from publishers to subscribers, we calculate time differences with timestamps of events on publishers and subscribers. Because publishers and subscribers run in different machines, we exploit the Precise Time Protocol (PTP) [11] that guarantees fine-grained time synchronization for distributed machines, and achieves clock accuracy in the sub-microsecond range on a local network.

III.4.2 Scalability Results

We used the 380 VMs for our scalability experiments. Each broker operates on a VM for which we used 160 VMs in total (120 VMs for edge brokers and 40 VMs for routing brokers). Of the remaining 220 VMs, 20 VMs are used for publishers and 200 VMs for subscribers. Each of these VMs runs 25 publisher or 50 subscriber test applications. We locate 50 publishers or 100 subscribers for each network (*i.e.*, 2 VMs for each network). The entire number of publishers and publishers is 1,000 and 10,000, respectively. Subscribers in each network are interested in 100 topics out of 1,000 topics in a system. Publishers push data every 50 milliseconds, and the size of a data sample is 64 bytes. We use settings described above as a default in our experiments.

For end-to-end latency of measurements, we collect latency values of 5,000 samples in total for each subscriber and use values only after 1,000 samples since the latency values of the initial samples are not consistent due to coordination and discovery process overhead until system stabilizes (*e.g.*, time for discovery of brokers and creating routes).

III.4.2.1 Scalability of the Broker Overlay Layer

Since the edge brokers are responsible for delivering data incoming from other brokers to subscribers in a local network, the computation overhead on edge brokers grows linearly

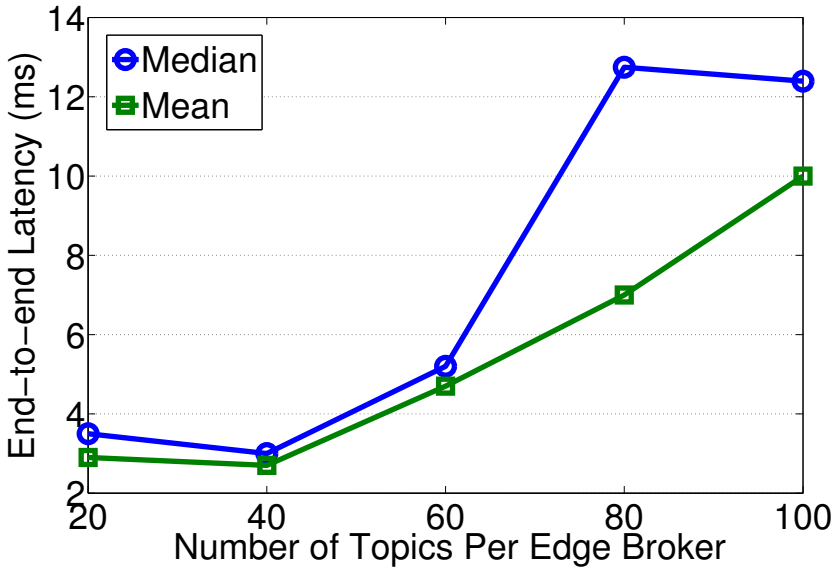

Figure 29: Mean and Median End-to-end Latency of Pub/Sub by Different Number of Topics Per Network

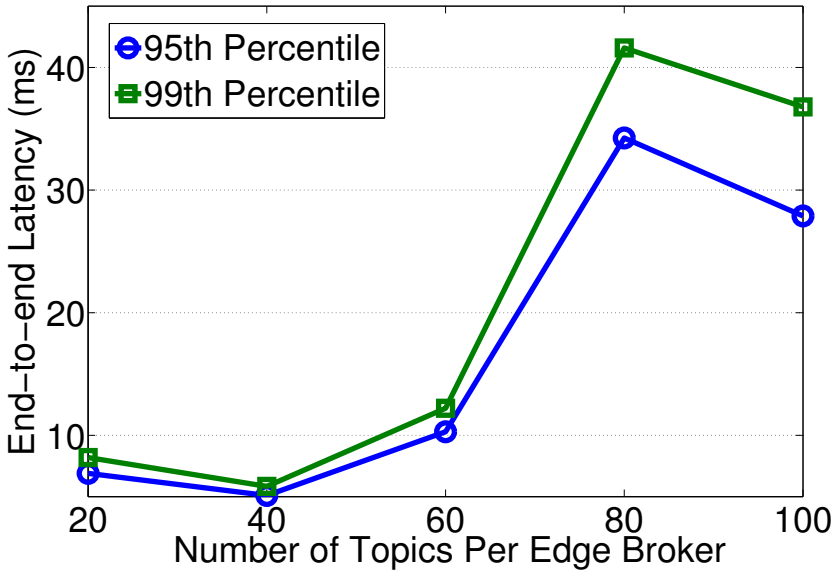

Figure 30: 95th and 99th Percentile End-to-end Latency of Pub/Sub by Different Number of Topics Per Network

as the number of adopted topics increases. Figure 29, 30, and 31 show results with different number of topics per edge broker, increasing the number of topics from 20 to 100 out of

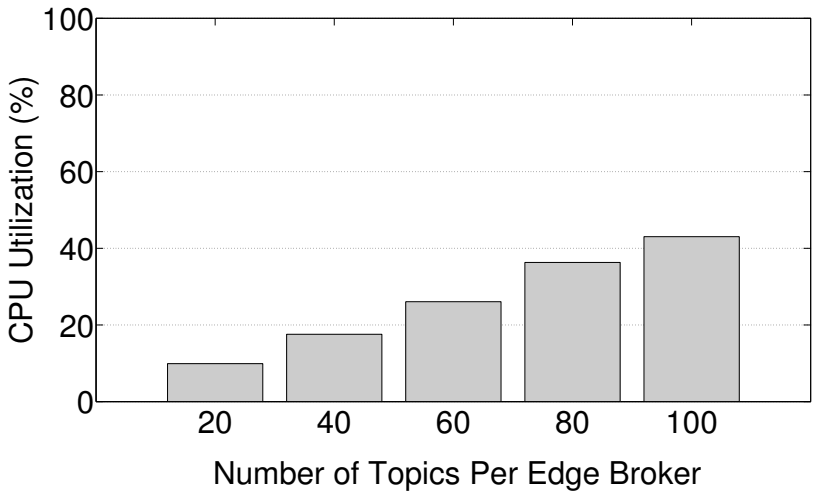

Figure 31: CPU Utilization by Different Number of Topics Per Network

1,000 topics in a system. The CPU utilization linearly increases by the number of adopted topics, and average and maximum latency values grow as well. From these results, we can infer that if the number of incoming streams increases due to more number of topics per network, it affects latency values because edge brokers need to manage more number of input and output pub/sub streams.

Our solution supports load balancing in the group of routing brokers and makes it possible to flexibly scale systems with the number of topics. Figure 32, 33, and 34 present latency and CPU usage by different number of routing brokers. When the number of routing brokers is small, in this case 5, the CPU of the routing brokers become saturated and latency values are adversely impacted. However, after increasing the number of routing brokers to 10, latency values improve. The results in Figure 34 also validate that CPU usage linearly decreases by increasing the number of routing brokers.

III.4.2.2 Scalability of the Coordination Layer

We evaluate the scalability of a ZooKeeper-based centralized coordination service by increasing the number of simultaneous joining subscribers. Figure 35 and 35 shows average and maximum latency, *i.e.*, the amount of time it takes for the server to respond to a

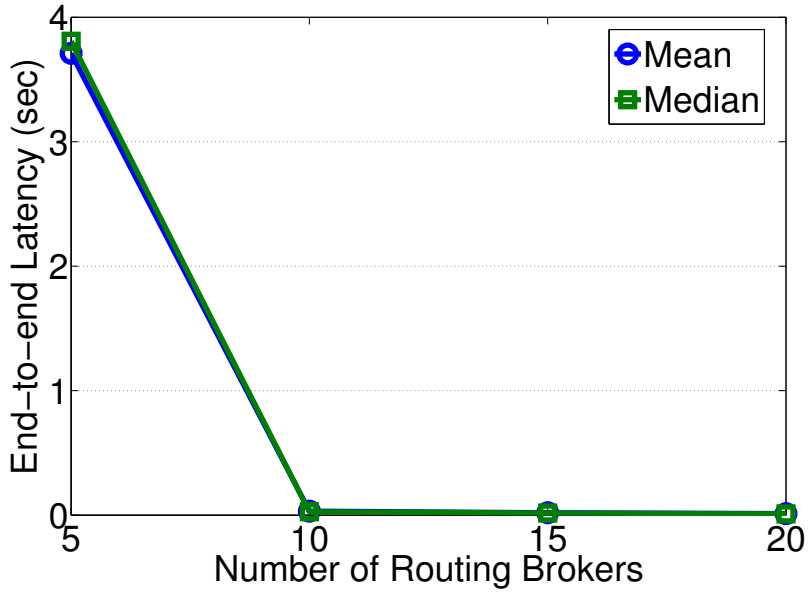

Figure 32: Mean and Median End-to-end Latency of Pub/Sub with Load Balance in Routing Brokers

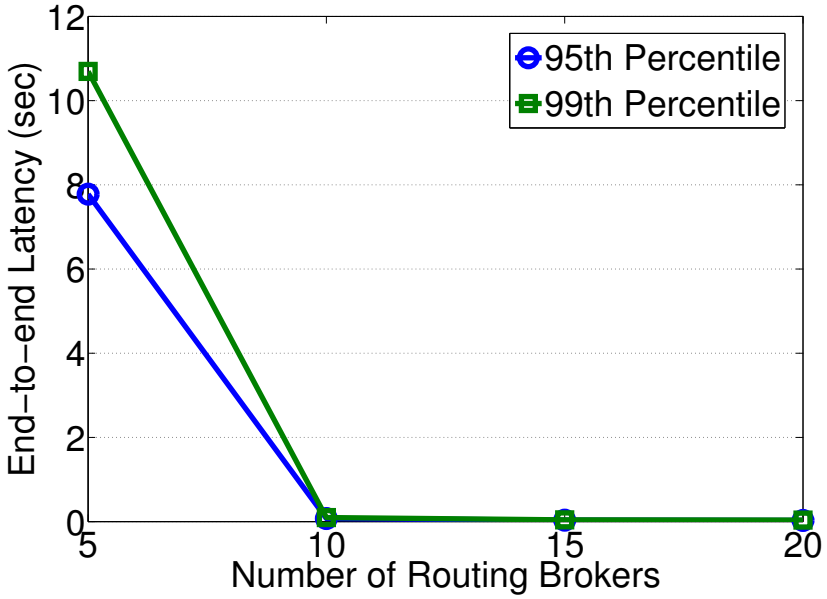

Figure 33: 95th and 99th Percentile End-to-end Latency of Pub/Sub with Load Balance in Routing Brokers

client request. Figure 37 presents the number of used znodes and watches. We use *mntr*,

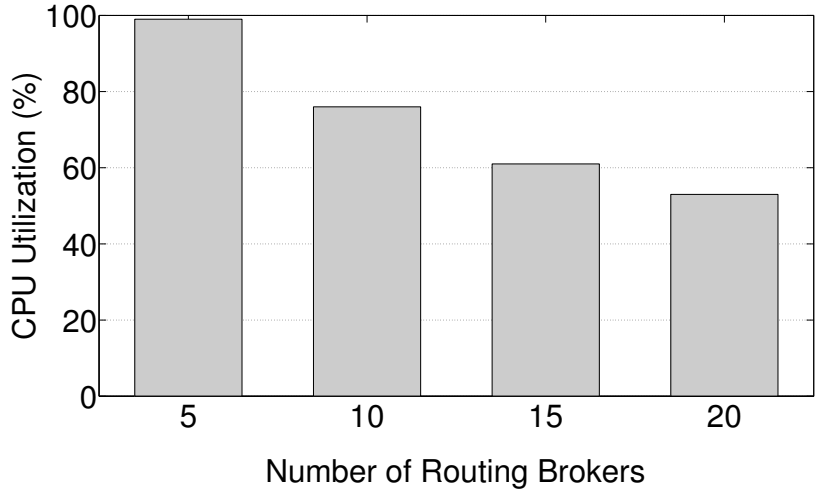

Figure 34: CPU Utilization with Load Balance in Routing Brokers

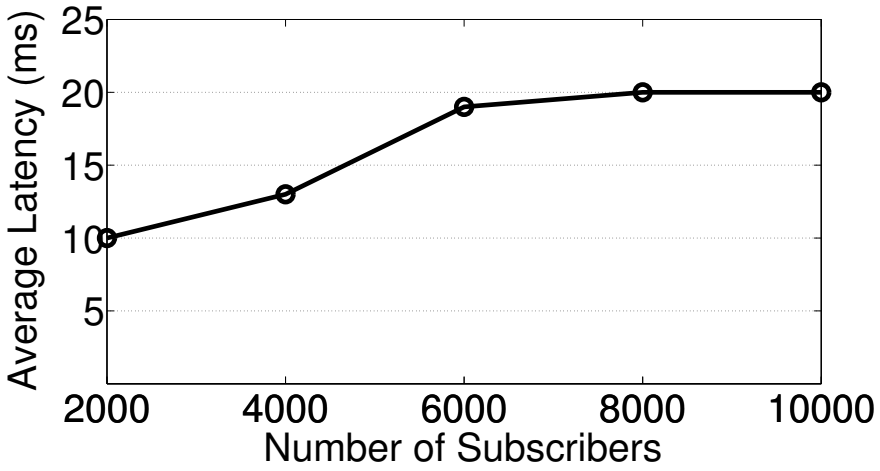

Figure 35: Average Latency of Coordination Service by Different Number of Joining Subscribers

a ZooKeeper command for monitoring service[5], to retrieve the experimental values presented in our results. We increase the number of subscribers from 2,000 to 10,000 in steps of 2,000. The average latency increases from 10 milliseconds to 20 milliseconds and the number of znodes and watches linearly increase approximately 2,000 and 4,000, respectively by the increased number of subscribers. The reason why the number of watches are

[5] `http://zookeeper.apache.org/doc/trunk/zookeeperAdmin.html`

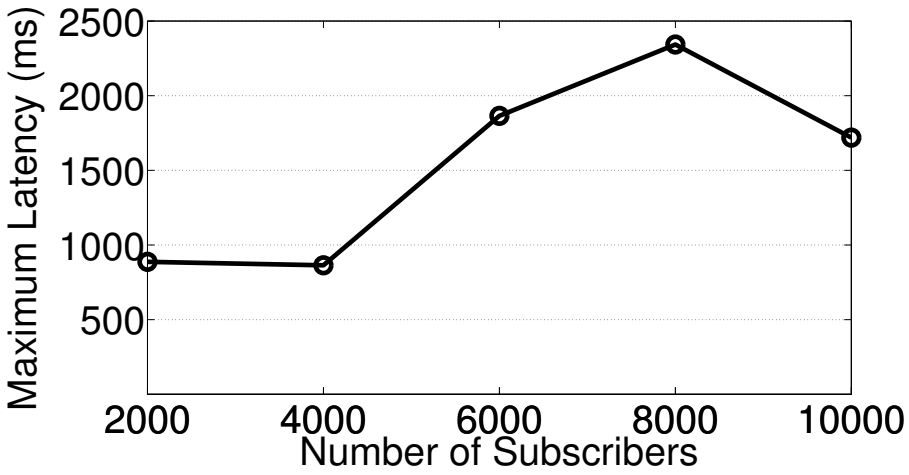

Figure 36: Maximum Latency of Coordination Service by Different Number of Joining Subscribers

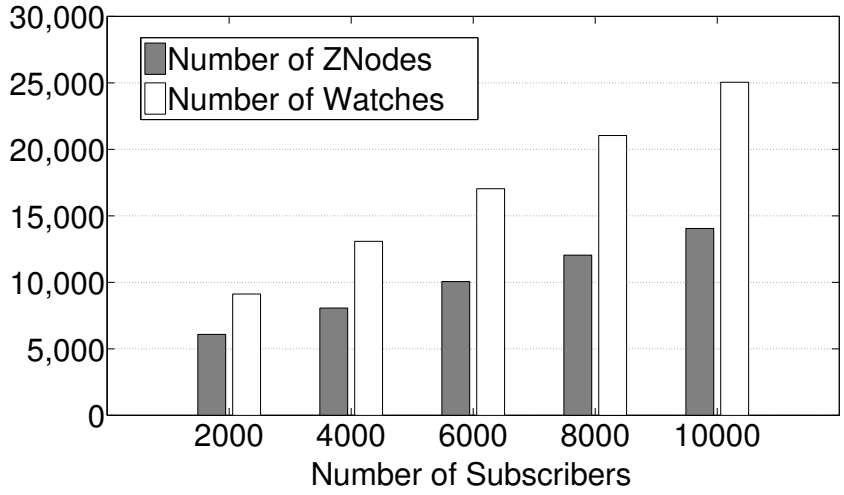

Figure 37: Number of ZNodes and Watches by Different Number of Joining Subscribers

twice compared to the number of znodes is that it needs to notify brokers for both publishers and subscribers if they have matching pub/sub endpoints.

III.4.3 Deadline-aware Overlays

We also conducted experiments to validate our deadline-aware overlays showing latency and overhead by comparing the performance parameters for multi-path and single-path overlays. A topology used for these experiments was shown in Figure 26. We use Dummynet [68] to simulate network delays and packet losses, which are common in WANs. These parameters are varied depending on geographic locations of brokers, which is a factor influencing the need for deadline-aware overlays. For multi-path overlay experiments, we use delay and loss data provided by Verizon, which shows latency and packet delivery statistics for communication between different countries across the globe.[6] We categorize delay and loss data into two groups (*i.e.*, A with 30ms delay and no packet loss, and B with 250 msec delay and 1% packet loss in Table 3) and experimented 8 possible combinations with given links (*i.e.*, L1, L2, and L3 as shown in Figure 26), and test cases described in Table 3.

Table 3: Deadline-aware Overlays Experiment Cases

Test Cases	L1	L2	L3
Case 1	A	A	A
Case 2	A	A	B
Case 3	A	B	A
Case 4	A	B	B
Case 5	B	A	A
Case 6	B	A	B
Case 7	B	B	A
Case 8	B	B	B

A = 30ms delay, no packet loss
B = 250ms delay, 1% packet loss

Figure 38 and 39 show average and maximum latency of single-path overlays with

[6] `http://www.verizonenterprise.com/about/network/latency`

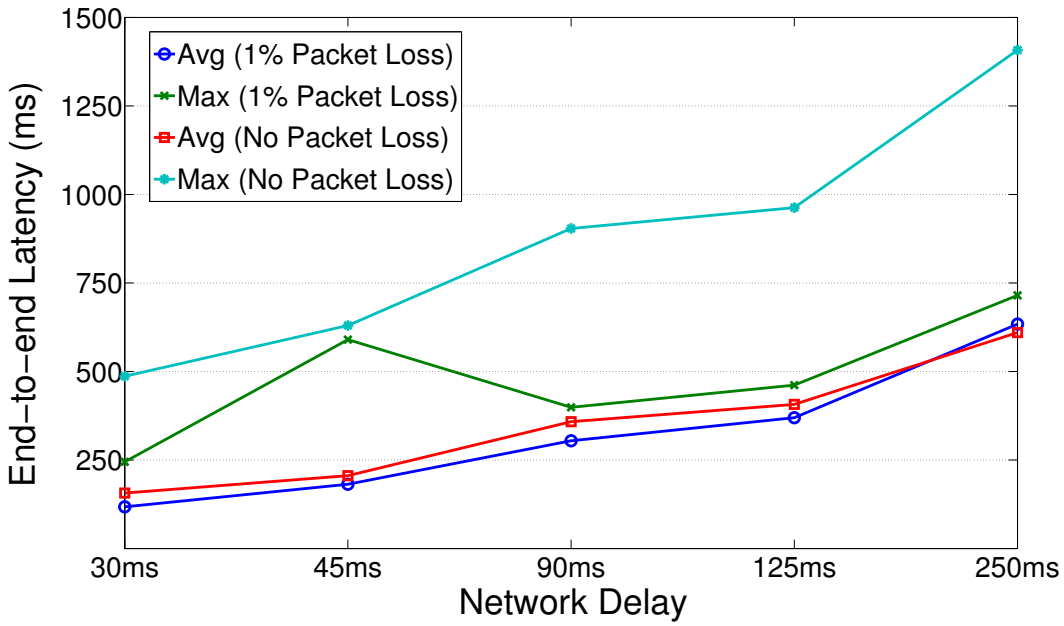

Figure 38: End-to-end Latency of Pub/Sub with Single-path Overlays

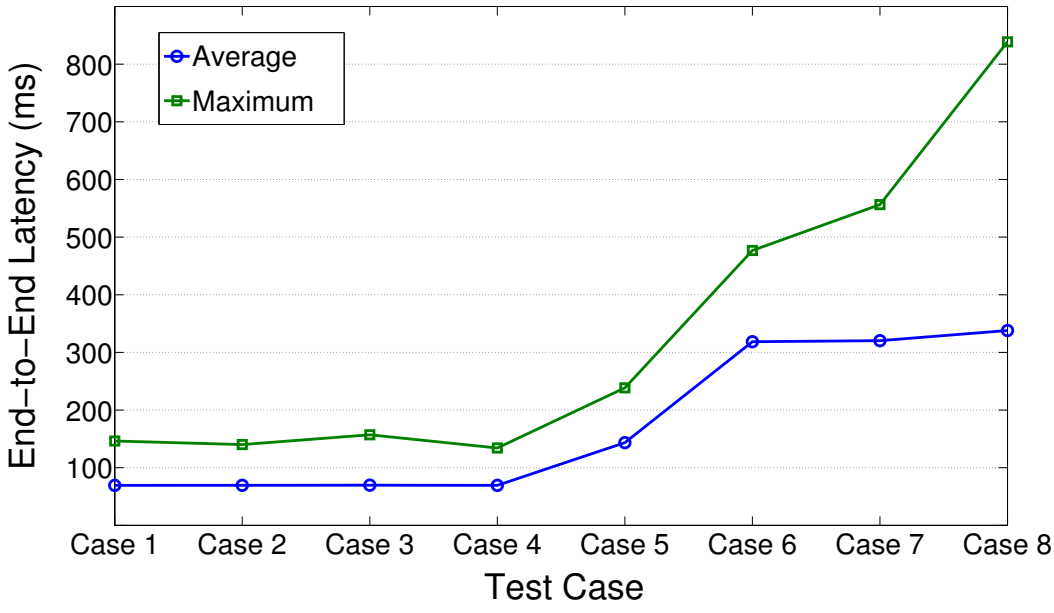

Figure 39: End-to-end Latency of Pub/Sub Multi-path Overlays

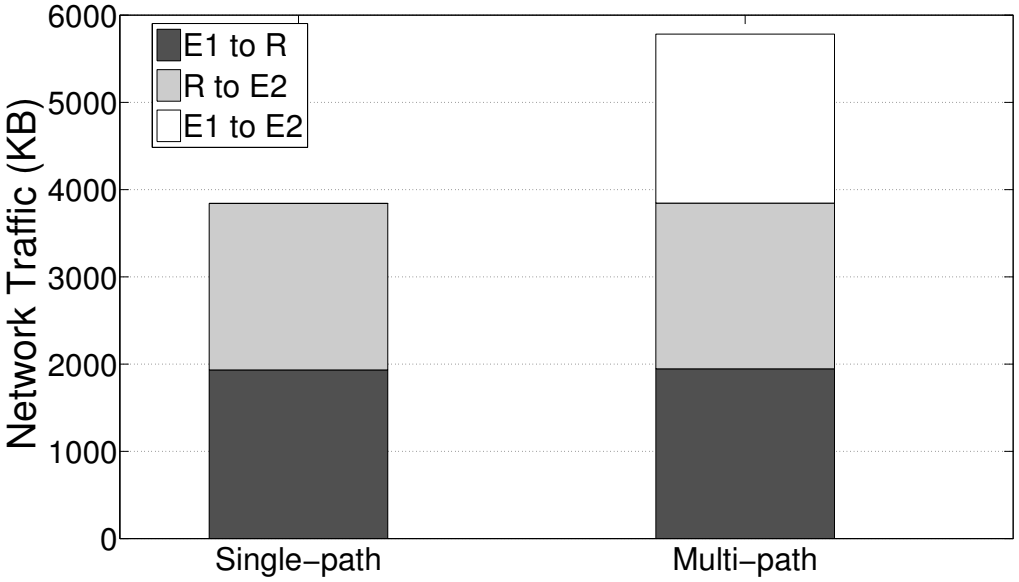

Figure 40: Overhead Comparison

different network delays and packet loss and multi-path overlays with 8 test cases, respectively. From case 1 to case 5, multi-path overlays perform better than any cases of single-path in terms of latency. All cases of multi-path overlays outperform a case with 125 milliseconds delay and 1% packet loss in single-path overlays. In spite of that, a multi-path overlay builds a duplicate path from an edge broker other than from a routing broker, so it causes extra overhead compared to a single-path overlay due to additional computations and extra network transfer at the edge broker. We measure network transfer overhead for 10,000 samples from a publisher to a subscriber to compare single-path and multi-path by using *tcpdump*[7] and the results are presented in Figure 40.

III.5 Related Work

Prior research on pub/sub systems can be classified into topic-based, attribute-based, and content-based depending on the subscription model. The topic-based model, such as Scribe [70], TIB/RV [59], and SpiderCast [15], groups subscription events in topics. In the

[7] `http://www.tcpdump.org`

attribute-based model, events are defined by specific types, and therefore this model helps developers to define data models in a robust way by type-checking. The content-based model [13, 65] allows subscribers to express their interests by specifying conditions on the data content of events, and the system filters out and delivers events based on the conditions. The OMG DDS adopts a data-centric model that groups subscriptions in topics, validates types of topic events, and also filters out events by conditions on data content using a special topic called *Content-Filtered Topic (CFT)*. Besides, it matches subscriptions based on offered and requested QoS parameters to disseminate data with assured service levels.

Pub/sub systems tend to form overlay networks to support application-level multicast rather than using IP-based multicast owing to the fact that IP multicast is not supported in WANs and the limited number of IP-based multicast addresses would not fit the potential number of logical channels for fine-grained subscription models [4]. Overlay architectures for pub/sub systems can be categorized into broker-based overlay [13, 59, 65], structured peer-to-peer [70], and unstructured peer-to-peer. GREEN [80] supports configurable overlay architectures for different network environments. PubSubCoord adopts a hybrid approach that constructs unstructured peer-to-peer overlays in LANs by dynamically discovering peers via multicast, and broker-based overlays in WANs.

BlueDove [46] is similar to our approach in that it achieves scalability and elasticity by harnessing cloud resources, and is a two-tier architecture to reduce the number of delivery hops and for simplicity. However, this service is designed for enterprise systems deployed in the cloud and does not consider the restrictions of physical locations of pub/-sub endpoints. In our system, pub/sub endpoints located in different networks dynamically discover each other with the help of edge brokers, and therefore we consider physical restrictions of pub/sub endpoints and they cannot connect to any edge brokers.

Bellavista et al. [7] study QoS-aware pub/sub systems over WANs and compare multiple existing pub/sub systems supporting QoS including DDS. In [41], the authors evaluate a pub/sub system for wide-area networks named Harmony and techniques for responsive and

high available messaging. The Harmony system delivers messages through broker overlays placed in different physical networks, and pub/sub endpoints communicate via local brokers located in the same network. Although this effort describes a WAN-scale pub/sub solution with QoS support, it centers on selective routing strategies to balance responsiveness and resource usage in view of the fact that its architecture is based on multi-hop broker networks unlike our 2-hop solution.

IndiQoS [12] also proposes a pub/sub system with QoS support to reduce end-to-end latency by exploiting network-level reservation mechanisms, where message brokers are structured using distributed hash table (DHT). Similar to IndiQoS, we pursue low-latency and high availability but our solution also supports other QoS policies such as configurable transport reliability, data persistence, ordering, and resource management by controlling depth of history data and subscribing rate. We do not use a DHT solution for brokers and so a comparison along these lines will be part of our future work.

Recent research [28, 50] has broadened the scope of DDS to WANs by bringing in routing engines to disseminate data from a local network to others. Our solution utilizes similar routing engines and additionally solves the discovery and coordination problem between routing engines that otherwise requires significant manual efforts for large-scale systems. Finally, [92] suggests separation of control and data plane in next generation pub/sub systems, which is motivated by software-defined networking (SDN).

III.6 Concluding Remarks

Emerging paradigms such as the Industrial Internet of Things illustrate the need to disseminate large volumes of data between a large number of heterogeneous entities that are geographically distributed, and require stringent QoS properties for data dissemination from the publishers of information to the subscribers. This paper presents the design, implementation, and evaluation of PubSubCoord, which is a cloud-enabled coordination and

discovery service for internet-scale pub/sub applications. PubSubCoord supports scalability in terms of data dissemination as well as coordination, dynamic discovery, and configurable QoS properties. The test harness and capabilities in PubSubCoord are available for download from `www.dre.vanderbilt.edu/pubsubcoord`.

The following is a summary of the insights we gained from this research and the empirical evaluations.

- **PubSubCoord disseminates data in a scalable manner for systems having many pub/sub endpoints and topics**. The experimental results show that PubSubCoord can deliver streamed data within 100 milliseconds for a system having 10,000 subscribers and 1,000 topics distributed across more than 100 networks. As the number of topics increases in a system, our solution uses elastic cloud resources and load balancing techniques to deliver data in a scalable way. However, if the number of adopted topics per edge broker increases, service quality becomes worse as shown in the experimental results because edge brokers need to deal with more number of forwarding operations between routing brokers and pub/sub endpoints. If a system requires higher frequency or more number of topics per network, edge brokers possibly become bottleneck, so an elastic solution for edge brokers will be needed.

- **Centralized coordination service like ZooKeeper can serve as a pub/sub control plane for large-scale systems**. Our solution employs a centralized service for coordinating pub/sub brokers for its consistency and simplicity, and our experimental results show that average latency of the coordination service is 20 milliseconds for 10,000 subscribers joining simultaneously and the number of data nodes and notifications linearly increase by organizing its data tree in a hierarchical way. Our experiments use a standalone server for coordination, but multiple servers as quorum can be used for scalability and fault-tolerance and ZooKeeper guarantees consistency of data between multiple servers. The quorum is more scalable for read operations,

but not for write operations that require synchronizing data between servers. In future, we plan to carry out experiments with increasing the number of coordination servers to understand its scalability for pub/sub broker coordination in depth.

- **Configurable QoS supported by DDS can be used for low-latency data delivery in WANs by building multi-path overlays**. We use configurable *deadline* QoS to deliver data at low-latency by establishing selective multi-path overlays, and validate this approach by providing experimental results. Since not every path can be a delay-sensitive path, we need some higher level policy management (*e.g.*, offered and requested QoS management between network domains) to decide what characterizes a delay-sensitive path. In addition, although this approach assures low-latency data delivery, it occurs extra overhead by duplicating data delivery from multiple paths. To reduce the costs, we can utilize *ownership* QoS that dynamically selects an owner of data streams to reduce data traffic from backups, and the owner is changed to a backup when the owner fails to meet its deadline. Our deadline-aware overlay optimizations are possible due to capabilities of DDS; implementing similar optimizations for other messaging systems will require identifying similar opportunities.

- **End-to-end QoS management is required for efficiency.** Most of the QoS policies in our solution are supported by hop-by-hop enforcement between brokers. Yet, some QoS policies for persistence, reliability, and ordering used in our experiments guarantee end-to-end QoS. However, this approach would be inefficient for some cases. For example, the *durability* QoS ensures sending previously published data to late joining subscribers. To support end-to-end data persistence with hop-by-hop QoS enforcement, each broker needs to keep history data in memory that will not be freed until it is acknowledged. This is beneficial for some late joining subscribers that require history data with low-latency. However, keeping duplicated history data on each broker unnecessarily consumes memory resources. To reduce this overhead,

we can suggest end-to-end acknowledgment mechanisms to provide persistence and reliability in an efficient way.

CHAPTER IV

A CLOUD MIDDLEWARE FOR ASSURING PERFORMANCE AND HIGH AVAILABILITY OF REAL-TIME APPLICATIONS

IV.1 Motivation

Cloud computing is a large-scale distributed computing platform based on the principles of utility computing that offers infrastructures such as CPU, storage, network as well as applications as services over the Internet [1]. The driving force behind the success of cloud computing is economy of scale. Traditionally, cloud computing has focused on enterprise applications. Lately, however, a class of real-time applications that demand both high availability and predictable response times are moving towards cloud-based hosting [14, 17, 84].

To support real-time applications in the cloud, it is necessary to satisfy low latency, reliability and high availability demands of such applications. Although the current cloud-based offerings can adequately address the performance and reliability requirements of enterprise applications, new algorithms and techniques are necessary to address the Quality of Service (QoS) needs, *e.g.*, low-latency needed for good response times and high availability, of performance-sensitive, real-time applications.

For example, in a cloud-hosted platform for personalized wellness management [14], high-availability, scalability and timeliness is important for providing on-the-fly guidance to wellness participants to adjust their exercise or physical activity based on real-time tracking of the participant's response to current activity. Assured performance and high availability is important because the wellness management cloud infrastructure integrates and interacts with the exercise machines both to collect data about participant performance and to adjust the intensity and duration of the activities.

IV.1.1 Challenges

Challenge 1: Assuring timeliness and high-availability in virtualized environments

Prior research in cloud computing has seldom addressed the need for supporting real-time applications in virtualized environment which is common in the cloud.[1] However, there is a growing interest in addressing these challenges as evidenced by recent efforts [91]. Since applications hosted in the cloud often are deployed in virtual machines (VMs), there is a need to assure the real-time properties of the VMs. A recent effort on real-time extensions to the Xen hypervisor [91] has focused on improving the scheduling strategies in the Xen hypervisor to assure real-time properties of the VMs. While timeliness is a key requirement, high availability is also an equally important requirement that must be satisfied.

Fault tolerance based on redundancy is one of the fundamental principles for supporting high availability in distributed systems. In the context of cloud computing, the Remus [18] project has demonstrated an effective technique for VM failover using one primary and one backup VM solution that also includes periodic state synchronization among the redundant VM replicas. The Remus failover solution, however, incurs shortcomings in the context of providing high availability for soft real-time systems hosted in the cloud in terms of intelligent deployment of backups.

Challenge 2: Lack of effective VM replica placement middleware

Current high availability solutions such as Remus do not focus on effective replica placement. Consequently, it cannot assure real-time performance after a failover decision because it is likely that the backup VM may be on a physical server that is highly loaded. The decision to effectively place the replica is left to the application developer or cloud administrators in a manual way. Unfortunately, any replica placement decisions made offline are not attractive for a cloud platform because of the substantially changing dynamics of the cloud platform in terms of workloads and failures. This requirement adds an inherent complexity for the developers who are responsible for choosing the right physical host with

[1] In this research we focus on soft real-time applications since it is unlikely that hard real-time and safety-critical applications will be hosted in the cloud.

enough capacity to host the replica VM such that the real-time performance of applications is met. It is not feasible for application developers to provide these solutions, which calls for a cloud platform-based solution that can shield the application developers from these complexities.

IV.1.2 Solution Approach

To address these requirements, this chapter makes the following three contributions described in Section IV.4:

1. We present a fault-tolerant architecture in the cloud geared to provide high availability and reliability for soft real-time applications. Our solution is provided as a middleware that extends the Remus VM failover solution [18] and is integrated with the OpenNebula cloud infrastructure software [23] and the Xen hypervisor [6]. Section IV.4.3 presents a hierarchical architecture motivated by the need for separation of concerns and scalability.

2. In the context of our fault-tolerant architecture, Section IV.4.4 presents the design of a pluggable framework that enables application developers to provide their strategies for choosing physical hosts for replica VM placement. Our solution is motivated by the fact that not all applications will impose exactly the same requirements for timeliness, reliability and high availability, and hence a "one-size-fits-all" solution is unlikely to be acceptable to all classes of soft real-time applications. Moreover, developers may also want to fine tune their choice by trading off resource usage and QoS properties with the cost incurred by them to use the cloud resources.

To evaluate the effectiveness of our solution, we use a representative soft real-time application hosted in the cloud and requiring high availability. For replica VM placement, we have developed an Integer Linear Programming (ILP) formulation that can be plugged

into our framework. This placement algorithm allocates VMs and their replicas to physical resources in a data center that satisfies the QoS requirements of the applications. We present results of experimentation focusing on critical metrics for real-time applications such as end-to-end latency and deadline miss ratio. Our goal in focusing on these metrics is to demonstrate that recovery after failover has negligible impact on the key metrics of real-time applications. Moreover, we also show that our high availability solution at the infrastructure-level can co-exist with an application-level fault tolerance capability provided by the application.

IV.2 Related Work

Prior work in the literature of high availability solutions and VM placement strategies are related to the research contributions we offer in this chapter. In this section, we present a comparative analysis of the literature and how our solutions fit in this body of knowledge.

IV.2.1 Underlying Technology: High Availability Solutions for Virtual Machines

To ensure high-availability, we propose a fault-tolerant solution that is based on the continuous checkpointing technique developed for the Xen hypervisor called Remus [18]. We discuss the details and shortcomings of Remus in Section IV.3.2.

Several other high availability solutions for virtual machines are reported in the literature. VMware fault-tolerance [76] runs primary and backup VMs in lock-step using deterministic replay. This keeps both the VMs in sync but it requires execution at both the VMs and needs high quality network connections. In contrast, our model focuses on a primary-backup scheme for VM replication that does not require execution on all replica VMs.

Kemari [86] is another approach that uses both lock-stepping and continuous checkpointing. It synchronizes primary and secondary VMs just before the primary VM has to send an event to devices, such as storage and networks. At this point, the primary

VM pauses and Kemari updates the state of the secondary VM to the current state of primary VM. Thus, VMs are synchronized with lower complexity than lock-stepping. External buffering mechanisms are used to improve the output latency over continuous check-pointing. However, we opted for Remus since it is a mature solution compared to Kemari.

Another important work on high availability is HydraVM [34]. It is a storage-based, memory-efficient high availability solution which does not need passive memory reservation for backups. It uses incremental check-pointing like Remus [18], but it maintains a complete recent image of VM in shared storage instead of memory replication. Thus, it claims to reduce hardware costs for providing high availability support and provide greater flexibility as recovery can happen on any physical host having access to shared storage. However, the software is not open-source or commercially available.

IV.2.2 Approaches to Virtual Machine Placement

Virtual machine placement on physical hosts in the cloud critically affects the performance of the application hosted on the VMs. Even when the individual VMs have a share of the physical resources, effects of context switching, network performance and other systemic effects [53, 67, 87, 93] can adversely impact the performance of the VM. This is particularly important when high availability solutions based on replication must also consider performance as is the case in our research.

The approach proposed in [36] is closely related to the scheme we propose in this paper. The authors present an autonomic controller that dynamically assigns VMs to physical hosts according to policies specified by the user. While the scheme we propose also allows users to specify placement policies and algorithms, we dynamically allocate the VMs in the context of a fault-tolerant cloud computing architecture that ensures high-availability solutions.

Lee et al. [45] investigated VM consolidation heuristics to understand how VMs perform when they are co-located on the same host machine. They also explored how the

resource demands such as CPU, memory, and network bandwidth are handled when consolidated. The work in [8] proposed a modified Best Fit Decreasing (BFD) algorithm as a VM reallocation heuristic for efficient resource management. The evaluation in the paper showed that the suggested heuristics minimize energy consumption while providing improved QoS. Our work may benefit from these prior works and we are additionally concerned with placing replicas in a way that applications continues to obtain the desired QoS after a failover.

IV.2.3 Comparative Analysis

Although there are several findings in the literature that relate to our three contributions, none of these approaches offer a holistic framework that can be used in a cloud infrastructure. Consequently, the combined effect of individual solutions has not been investigated. Our work is a step in the direction of fulfilling this void. Integrating this different approaches is not straightforward and requires good design decisions, which we have demonstrated with our work and presented in the remainder of this chapter.

IV.3 Background

Our middleware solution is designed to operate with existing cloud infrastructure platforms, such as OpenNebula [23], and hypervisor technologies, such as Xen [6]. In particular, our solution is based on Remus [18], which provides high availability to VMs that use the Xen hypervisor. For completeness, we describe these building blocks in more detail here.

IV.3.1 Cloud Infrastructure and Virtualization Technologies

Contemporary cloud infrastructure platforms, such as OpenStack [62], Eucalyptus [55], or OpenNebula [23], manage the artifacts of a cloud infrastructure including servers, networks, and other equipment, such as storage devices. One of the key responsibilities such

infrastructure is to manage virtualized servers where user applications are running in the data center. Often, these platforms are architected in a hierarchical manner with a master or controller node oversees the activities of the worker nodes that host VMs and user applications. In the OpenNebula platform we use, the master node is called the *Front-end Node*, and the worker nodes are called the *Cluster Nodes.*

Hypervisors, such as Xen [6] and KVM [42], offer virtualization environments that enable isolated multiple virtual machines to run within a shared physical machine. The hypervisor manages the virtual machines and ensures both performance and security isolation between different virtual machines hosted on the same physical server. To ensure that our solution can be adopted in a range of hypervisors, we use the *libvirt* [47] software suite that provides a portable approach to manage virtual machines. By providing a common API, libvirt is able to interoperate with a range of hypervisors and virtualization technologies.

IV.3.2 Remus High Availability Solution

Remus [18] is a software system built for the Xen hypervisor that provides OS- and application-agnostic high-availability on commodity hardware. Remus provides seamless failure recovery and does not require lock step-based, whole-system replication. Instead, the use of speculative execution in the Remus approach ensures that the performance degradation due to replication is kept to a minimum. Speculative execution decouples the execution of the application from state synchronization between replica VMs by interleaving these operations and, hence, not forcing synchronization between replicas after every update made by the application.

Remus uses a pair of replica VMs: a primary and a backup. Since Remus provides protection against single host fail-stop failures only, if both the primary and backup hosts fail concurrently, the failure recovery will not be seamless; however, Remus ensures that the system's data will be left in a consistent state even if the system crashes.

Remus is not concerned with where the primary and backup replicas are placed in the

data center. Consequently, it cannot guarantee any performance properties for the applications. The VM placement is the responsibility of the user, which we have shown to be a significant complexity for the user. Our VM failover solution leverages Remus while addressing these limitations in Remus.

IV.4 Design and Implementation

This section presents our contributions that collectively offer a high availability middleware architecture for soft real-time applications deployed in virtual machines in cloud data centers. We first describe the architecture and then describe the contributions in detail.

IV.4.1 Architectural Overview

The architecture of our high-availability middleware, as illustrated in Figure 41, comprises a Local Fault Manager (LFM) for each PM, and a replicated Global Fault Manager (GFM) to manage the cluster of PMs where LFMs are running. The inputs to the LFMs are the resource information of PMs and VMs gathered directly from the hypervisor. We collect information for resources such as the CPU, memory, network, storage, and processes. The GFM is responsible for making decisions on VM replica placement. Since no one-size-fits-all replica placement strategy is appropriate for all applications, our GFM supports a pluggable replica placement framework.

IV.4.2 Roles and Responsibilities

Before delving into the design rationale and solution details, we describe how the system will be used in the cloud. Figure 42 shows a use case diagram for our system in which roles and responsibilities of the different software components are defined. A user in the role of a system administrator will configure and run a GFM service and the several LFM services. A user in the role of a system developer can implement deployment algorithms

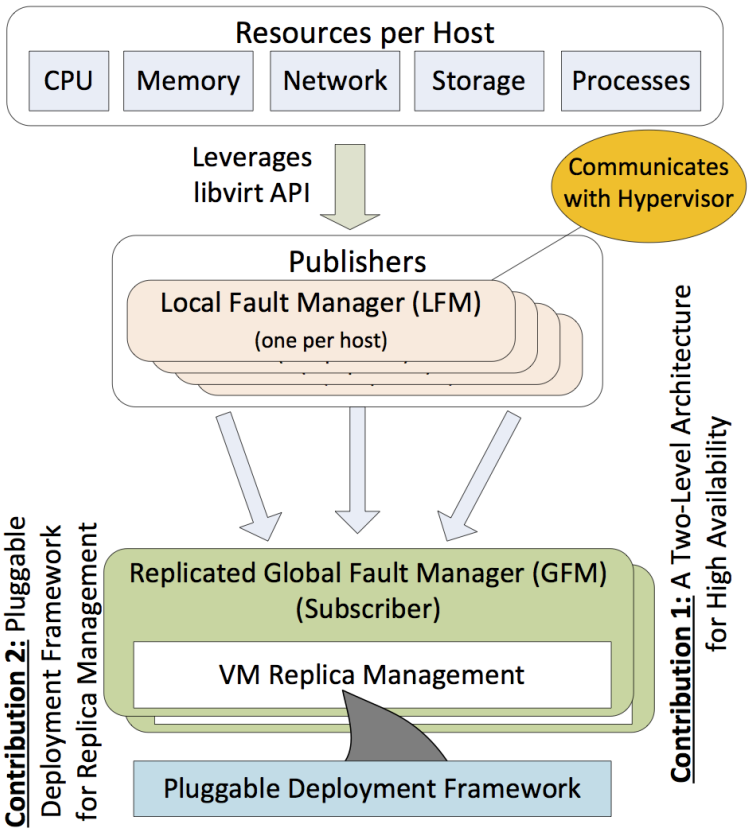

Figure 41: Conceptual System Design Illustrating Contributions

to find and use a better deployment solution. The LFM services periodically update resource information of VMs and PMs as configured by the user. The GFM service uses the deployment algorithms and the resource information to create a deployment plan for replicas of VMs. Then, the GFM sends messages to LFMs to run a backup process via high-availability solutions that leverages Remus.

IV.4.3 Contribution 1: High-Availability Solution

This section presents our first contribution that deals with providing a high availability middleware solution for VMs running soft real-time applications. Our solution assumes that a VM-level fault recovery is already available via solutions, such as Remus [18].

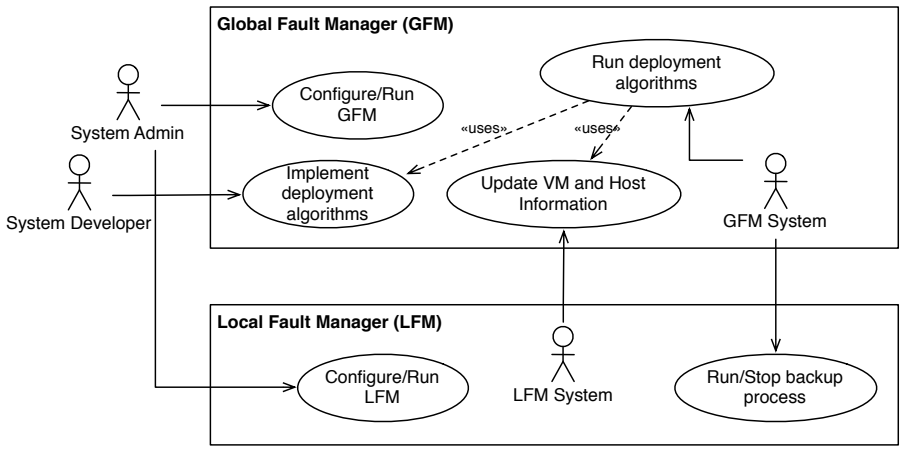

Figure 42: Roles and Responsibilities

IV.4.3.1 Rationale: Why a Hierarchical Model?

Following the strategy in Remus, we host the primary and backup VMs on different PMs to support the fault tolerance.

In a data center with hundreds of thousands of PMs, a Remus-based solution managing fault tolerance for different applications may be deployed on every server. Remus makes no effort to determine the effective placement of replica VMs; it just assumes that a replica pair exists. For our solution, however, assuring the QoS of the soft real-time systems requires effective placement of replica VMs.

A centralized solution that manages faults across an entire data center is infeasible. Moreover, it is not feasible for some central entity to poll every server in the data center for resource availability and their usage. Thus, an appropriate choice is to develop a hierarchical solution based on the principles of separation of concerns. At the local level (*i.e.*, host level), a fault management logic can interact with its local Remus software while also being responsible for collecting the local resource usage information. At the global level, a fault management logic can decide effective replica placement based on the timely resource usage information acquired from the local entities.

Although a two-level solution is described, for scalability reasons, multiple levels can

be introduced in the hierarchy where a large data center can be compartmentalized into smaller regions.

IV.4.3.2 Design and Operation

Our hierarchical solution is to utilize several Local Fault Managers (LFMs) associated with a single Global Fault Manager (GFM) in adjacent levels of the hierarchy. The GFM coordinates deployment plans of VMs and their replicas by communicating with the LFMs. Every LFM retrieves resource information from a VM that is deployed in the same PM as the LFM, and sends the information periodically to a GFM. We focus on addressing the deployment issue because existing solutions such as Remus delegates the responsibility of placing the replica VM onto the user. An arbitrary choice may result in severe performance degradation for the applications running in the VMs.

The replica manager is the core component of the GFM and is responsible for running the deployment algorithm provided by a user of the framework. This component determines the PM where the replica of a VM should be replicated as a backup. The location of the backup is then supplied to the LFM running on the host machine where the VM is located to take the required actions, such as informing the local Remus of its backup copy.

The LFM runs a high-availability (HA) service that is based on the Strategy pattern [24]. This service includes starting and stopping replica operations, and automatic failover from a primary VM to a backup VM in case of a failure. The use of the strategy pattern enables us to use a solution different from Remus, if one were to be available. This way we are not tightly coupled with Remus. Once the HA service is started and while it is operational, it keeps synchronizing the state of a primary VM to a backup VM. If a failure occurs during this period, it switches to the backup VM making it the active primary VM. When the HA service is stopped, it stops the synchronization process and high-availability is discontinued.

In the context of the HA service, the job of the GFM is to provide each LFM with

backup VMs that can be used when the service is executed. In the event of failure of a primary VM, the service ensures that the processing switches to the backup VM and it becomes the primary VM. This event is triggered when the LFM informs GFM of the failure event and requests additional backup VMs on which a replica can start. It is the GFM's responsibility to provide resources to the LFM in a timely manner so that the latter can move from a crash consistent state to seamless recovery fault tolerant state as soon as possible thereby assuring average response times of performance-sensitive soft real-time applications.

In the architecture shown in Figure 43, replicas of VMs are automatically deployed in hosts assigned by a GFM and LFMs. The following are the steps of the system described in the figure.

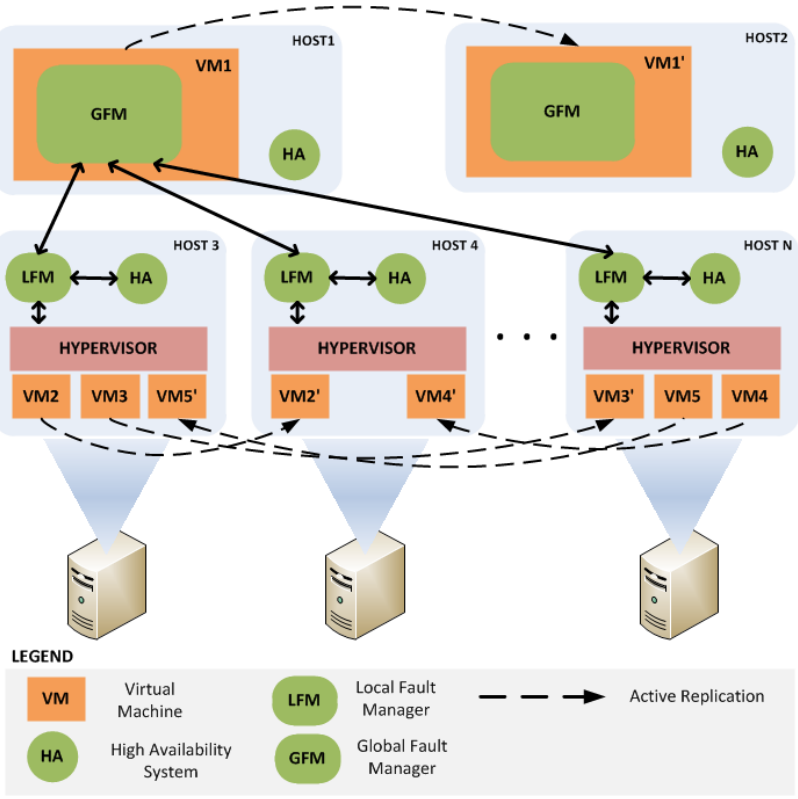

Figure 43: System Architecture

1. A GFM service is started, and the service waits for connections from LFMs.

2. LFMs will join the system by connecting to the GFM service.

3. The joined LFMs periodically send their individual resource usage information of VMs hosted on their nodes as well as that of the physical host, such as CPU, memory, and network bandwidth to the GFM.

4. Based on the resource information, the GFM determines an optimal deployment plan for the joined physical hosts and VMs by running a deployment algorithm, which can be supplied and parametrized by users as described in Section IV.4.4.

5. The GFM will notify LFMs to execute a HA service in LFMs with information of source VMs and destination hosts.

A GFM service can be deployed on a PM or inside a VM. In our system design, to avoid a single point of failure of a GFM service, a GFM is deployed in a VM and a GFM's VM replica is located in another PM. When the PM where the GFM is located fails, the backup VM containing the GFM service is promoted to primary and the GFM service continues to its execution via the high availability solution.

On the other hand, LFMs are placed in physical hosts used to run VMs in data centers. LFMs work with a hypervisor and a high availability solution (Remus in our case) to collect resource information of VMs and hosts and to replicate VMs to other backup hosts, respectively. Through the HA solution, a VM's disk, memory, and network connections are actively replicated to other hosts and a replication of the VM in a backup host is instantiated when the primary VM is failed.

IV.4.4 Contribution 2: Pluggable Framework for Virtual Machine Replica Placement

This section presents our second contribution that deals with providing a pluggable framework for determining VM replica placement.

IV.4.4.1 Rationale: Why a Pluggable Framework?

Existing solutions for VM high availability, such as Remus, delegate the task of choosing the PM for the replica VM to the user. This is a significant challenge since a bad choice of a heavily loaded PM may result in performance degradation. Moreover, a static decision is also not appropriate since a cloud environment is highly dynamic. To provide maximal autonomy in this process requires online deployment algorithms that make decisions on VM and replica VM placement.

Deployment algorithms determine which host machine should store a VM and its replica in the context of fault management. There are different types of algorithms to make this decision. Optimization algorithms, such as bin packing, genetic algorithms, multiple knapsack, and simulated annealing are some of the choices used to solve similar problems in a large number of industrial applications today. Moreover, different heuristics of the bin packing algorithm are commonly utilized techniques for VM replica placement optimization, in particular.

Solutions generated by such algorithms and heuristics have different properties. Similarly, the runtime complexity of these algorithms is different. Since different applications may require different placement decisions and may also impose different constraints on the allowed runtime complexity of the placement algorithm, a one-size-fits-all solution is not acceptable. Thus, we needed a pluggable framework to decide VM replica placement.

IV.4.4.2 Design of a Pluggable Framework for Replica VM Placement

In bin packing algorithms [9], the goal is to use minimum number of bins to pack the items of different sizes. Best-Fit, First-Fit, First-Fit-Decreasing, Worst-Fit, Next-Fit, and Next-Fit-Decreasing are the different heuristics of this algorithm. All these heuristics will be part of the middleware we are designing, and will be provided to the framework user to run the bin packing algorithm.

In our framework, we view VMs as items and the host machines as the bins. Resource information from the VMs, are utilized as weights to employ the bin packing algorithm. Resource information is aggregated into one single scalar value, and one dimensional bin packing is employed to find the best host machine where the replica of a VM will be stored. Our framework adopts the strategy pattern to enable plugging in different VM replica placement algorithms. A concrete problem we have developed and used in our replication manager is described in Section IV.5.

IV.4.5 Problem Fomulation

Our solution provides a framework that enables plugging in different user-supplied VM placement algorithms. We expect that our framework will compute replica placement decisions in an online manner in contrast to making offline decisions. We have formulated it as an Integer Linear Programming (ILP) problem.

In our ILP formulation we assume that a data center comprises multiple PMs. Each PM can in turn consist of multiple VMs. We also account for the resource utilizations of the physical host as well as the VMs on each host. Furthermore, not only do we account for CPU utilizations but also memory and network bandwidth usage. All of these resources are accounted for in determining the placement of the replicas because on a failover we expect our applications to continue to receive their desired QoS properties. Table 4 describes the variables used in our ILP formulation.

We now present the ILP problem formulation shown below with the defined constraints

Table 4: Notation and Definition of the ILP Formulation

Notation	Definition
x_{ij}	Boolean value to determine the i^{th} VM to the j^{th} physical host mapping
x'_{ij}	Boolean value to determine the replication of the i^{th} VM to the j^{th} physical host mapping
y_j	Boolean value to determine usage of the physical host j
c_i	CPU usage of the i^{th} VM
c'_i	CPU usage of the i^{th} VM's replica
m_i	Memory usage of the i^{th} VM
m'_i	Memory usage of the i^{th} VM's replica
b_i	Network bandwidth usage of the i^{th} VM
b'_i	Network bandwidth usage of the i^{th} VM's replica
C_j	CPU capacity of the j^{th} physical host
M_j	Memory capacity of the j^{th} physical host
B_j	Network bandwidth of the j^{th} physical host

that need to be satisfied to find an optimal allocation of VM replicas. The objective function of the problem is to minimize the number of PMs by satisfying the requested resource requirements of VMs and their replicas. Constraints (2) and (3) ensure every VM and VM's replica is deployed in a PM. Constraints (4), (5), (6) guarantee that the total capacity of CPU, memory, and network bandwidth of deployed VMs and VMs' replicas are packed into an assigned PM, respectively. Constraint (7) checks that a VM and its replica is not deployed in the same PM since the PM may become a single point of failure, which must be prevented.

$$\text{minimize} \sum_{j=1}^{m} y_j \tag{IV.1}$$

$$\text{subject to} \sum_{j=1}^{m} x_{ij} = 1 \qquad \forall i \tag{IV.2}$$

$$\sum_{j=1}^{m} x'_{ij} = 1 \qquad \forall i \tag{IV.3}$$

$$\sum_{i=1}^{n} c_i x_{ij} + \sum_{i=1}^{n} c'_i x'_{ij} \leq C_j y_j \qquad \forall j \tag{IV.4}$$

$$\sum_{i=1}^{n} m_i x_{ij} + \sum_{i=1}^{n} m'_i x'_{ij} \leq M_j y_j \qquad \forall j \tag{IV.5}$$

$$\sum_{i=1}^{n} b_i x_{ij} + \sum_{i=1}^{n} b'_i x'_{ij} \leq B_j y_j \qquad \forall j \tag{IV.6}$$

$$\sum_{i=1}^{n} x_{ij} + \sum_{i=1}^{n} x'_{ij} = 1 \qquad \forall j \tag{IV.7}$$

$$x_{ij} = \{0,1\},\ x'_{ij} = \{0,1\},\ y_j = \{0,1\} \tag{IV.8}$$

IV.5 Experimental Results

In this section we present results to show that our solution can seamlessly and effectively leverage existing solutions for fault tolerance in the cloud.

IV.5.1 Rationale for Experiments

Our high-availability solution for cloud-hosted soft real-time applications leverages existing VM-based solutions, such as the one provided by Remus. Moreover, it is also possible that the application running inside the VM itself may provide its own application-level fault tolerance. Thus, it is important for us to validate that our high availability solution can work seamlessly and effectively in the context of existing solutions.

IV.5.2 Representative Applications and Evaluation Testbed

To validate both the claims: (a) support for high-availability soft real-time applications, and (b) seamless co-existence with other cloud-based solutions, we have used two representative soft real-time applications. For the first set of validations, we have used an existing benchmark application that has the characteristics of a real-time application [71]. To demonstrate how our solution can co-exist with other solutions, we used a *word count* application that provides its own application-level fault-tolerance. We show how our solution can co-exist with different fault-tolerance solutions.

Our private cloud infrastructure for both the experiments we conducted comprises a cluster of 20 rack servers, and Gigabit switches. The cloud infrastructure is operated using OpenNebula 3.0 with shared file systems using Network File System (NFS) for distributing virtual machine images. Table 5 provides the configuration of each rack server used as a clustered node.

Table 5: Hardware and Software specification of Cluster Nodes

Processor	2.1 GHz Opteron
Number of CPU cores	12
Memory	32 GB
Hard disk	8 TB
Operating System	Ubuntu 10.04 64-bit
Hypervisor	Xen 4.1.2
Guest virtualization mode	Para

Our guest domains run Ubuntu 11.10 32-bit as operating systems, and each guest domain has 4 virtual CPUs and 4GB of RAM.

IV.5.3 Measuring the Impact on Latency for Soft Real-time Applications

To validate our high-availability solution including the VM replica placement algorithm, we used the RTI DDS Connext DDS latency benchmark. RTI Connext DDS is an implementation of the OMG DDS standard [57]. The RTI Connext DDS benchmark comprises code to evaluate the latency of DDS applications, and the test code contains both the publisher and the subscriber.

Our purpose in using this benchmark was to validate the impact of our high- availability solution and replica VM placement decisions on the latency of DDS applications. For this purpose, the DDS application was deployed inside a VM. We compare the performance between an optimally placed VM replica using our algorithm described in Section IV.4.5 and a potentially worse case scenario resulting from a randomly deployed VM. In the experiment, average latency and standard deviation of latency, which is a measure of the jitter, are compared for different settings of Remus and VM placement. Since a DDS application is a one directional flow from a publisher to a subscriber, the latency measurement is estimated as half of the round-trip time which is measured at a publisher. In each experimental run, 10,000 samples of stored data in the defined byte sizes in the table are sent from a publisher to a subscriber. We also compare the performance when no high-availability solution is used. The rationale is to gain insights into the overhead imposed by the high-availability solution.

Figure 44 shows how our Remus-based high-availability solution along with the effective VM placement affects latency of real-time applications. The measurements from the experimental results for the case of *Without Remus*, where VM is not replicated, shows consistent range of standard deviation and average of latency compared to the case of *Remus with Efficient Placement*. When Remus is used, average latency does not increase significantly, however, a higher fluctuation of latency is observed by measuring standard deviation values between both cases. From the results we can conclude that the state replication overhead from Remus incurs a wider range of latency fluctuations.

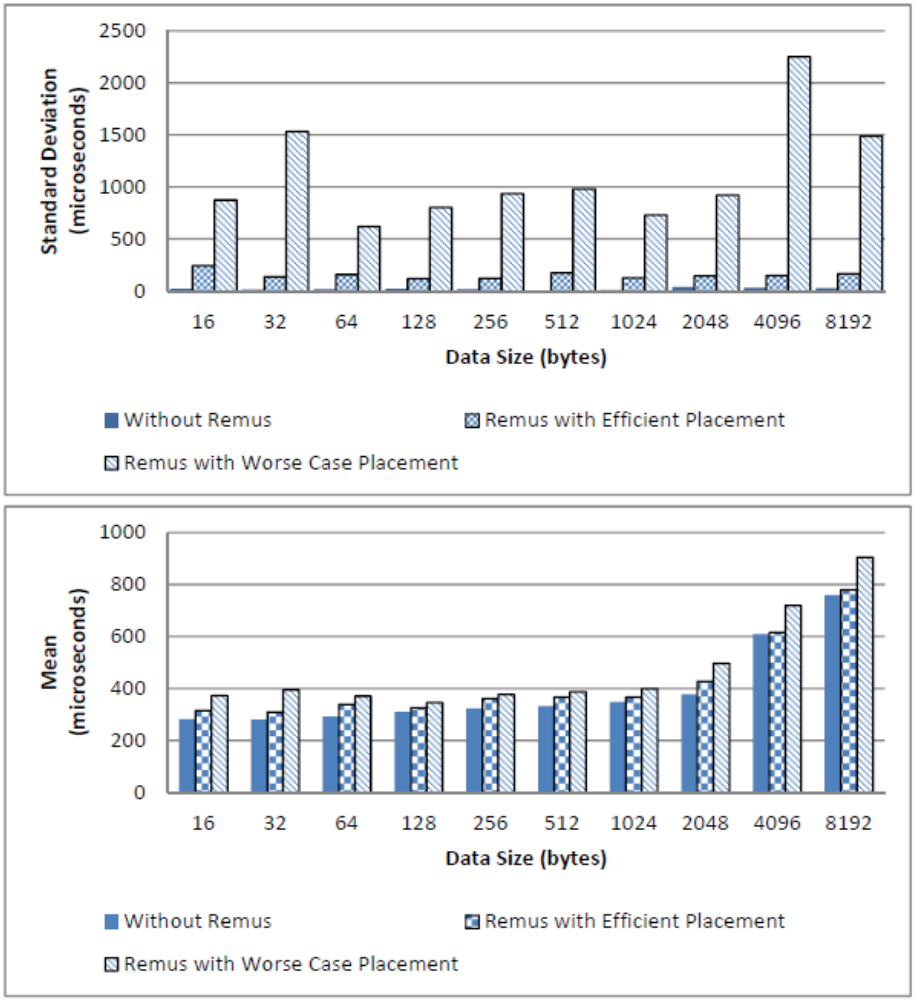

Figure 44: Latency Performance Test for Remus and Effective Placement

However, the key observation is that significantly wider range of latency fluctuations are observed in the standard deviation of latency in *Remus with Worst Case Placement*. On the contrary, the jitter is much more bounded using our placement algorithm. our framework guarantees that the appropriate number of VMs are deployed in PMs by following the defined resource constraints so that contention for resources between VMs does not occur even though a VM or a PM has crashed. However, if a VM and its replica is randomly placed without any constraints, unexpected latency increases for applications running on the VM could occur. The resulting values of latency's standard deviation in *Remus with*

Worst Case Placement demonstrate how the random VM placement negatively influences timeliness properties of applications.

IV.5.4 Validating Co-existence of High Availability Solutions

Often times the applications or their software platforms support their own fault-tolerance and high-availability solutions. The purpose of this experiment is to test whether it is possible for both our Remus-based high availability solution and the third party solution could co-exist.

To ascertain these claims, we developed a *word count* example implemented in C++ using OMG DDS. The application supports its own fault tolerance using OMG DDS QoS configurations as follows. OMG DDS supports a QoS configuration called *Ownership Strength,* which can be used as a fault tolerance solution by a DDS pub/sub application. For example, the application can create redundant publishers in the form of multiple data writers that publish the same topic that a subscriber is interested in. Using the *OWNERSHIP_STRENGTH* configuration, the DDS application can dictate who the primary and backup publishers are. Thus, a subscriber receives the topics only from the publisher with the highest strength. When a failure occurs, a data reader (which is a DDS entity belonging to a subscriber) automatically fails over to receive its subscription from a data writer having the next highest strength among the replica data writers.

Although such a fault-tolerant solution can be realized using the ownership QoS, there is no equivalent method in DDS if a failure occurs at the source of events such as a node that aggregates multiple sensors data and a node reading a local file stream as a source of events. In other words, although the DDS ownership QoS takes care of replicating the data writers and organizing them according to the ownership strength, if these data writers are deployed in VMs of a cloud data center, they will benefit from the replica VM placement strategy provided by our approach thereby requiring the two solutions to co-exist.

To experiment with such a scenario and examine the performance overhead as well

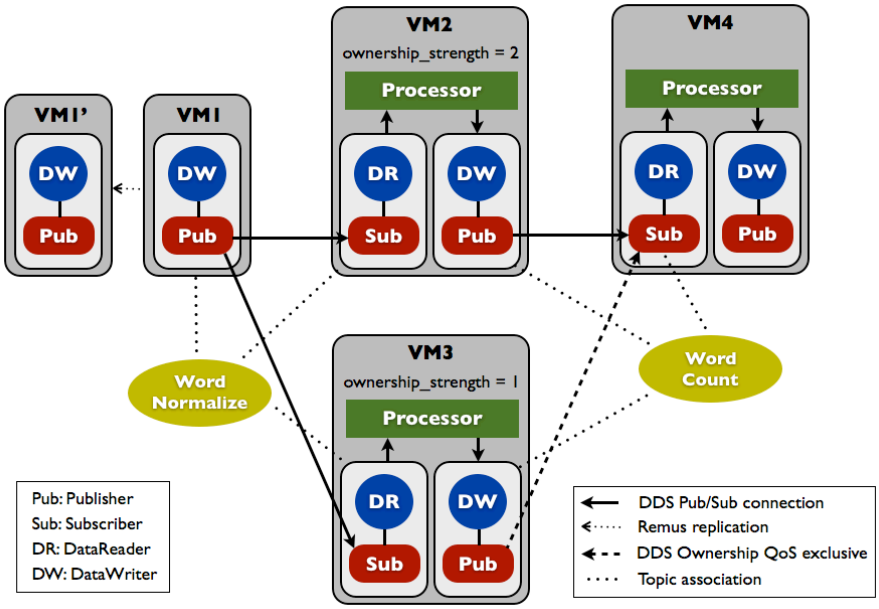

Figure 45: Example of Real-time Data Processing: Word Count

as message missed ratio (*i.e.*, lost messages during failover), we developed a DDS-based "word count" real-time streaming application. The system integrates both the high availability solutions. Figure 45 shows the deployment of the *word count* application running on the highly available system. Four VMs are employed to execute the example application. VM1 runs a process to read input sentences and publishes a sentence to the next processes. We call the process running on the VM1 as the *WordReader*. In the next set of processes, a sentence is split into words. These processes are called *WordNormalizer*. We place two VMs for the normalizing process and each data writer's Ownership QoS is configured with the exclusive connection to a data reader and the data writer in VM3 is set to the primary with higher strength. Once the sentences get split, words are published to the next process called the *WordCounter*, where finally the words are counted. In the example, we can duplicate processes for *WordNormalizer* and *WordCounter* as they process incoming events, but a process for *WordReader* cannot be replicated by having multiple data writers in different physical nodes as the process uses a local storage as a input source. In this case, our VM-based high availability solution is adopted.

Table 6: DDS QoS Configurations for the Word Count Example

DDS QoS Policy	Value
Data Reader	
Reliability	Reliable
History	Keep All
Ownership	Exclusive
Deadline	10 milliseconds
Data Writer	
Reliability	Reliable
Reliability - Max Blocking Time	5 seconds
History	Keep All
Resource Limits - Max Samples	32
Ownership	Exclusive
Deadline	10 milliseconds
RTPS Reliable Reader	
MIN Heartbeat Response Delay	0 seconds
MAX Heartbeat Response Delay	0 seconds
RTPS Reliable Writer	
Low Watermark	5
High Watermark	15
Heartbeat Period	10 milliseconds
Fast Heartbeat Period	10 milliseconds
Late Joiner Heartbeat Period	10 milliseconds
MIN NACK Response Delay	0 seconds
MIN Send Window Size	32
MAX Send Window Size	32

Table 6 describes the DDS QoS configurations used for our *word count* application. The throughput and latency of an application can be varied by different DDS QoS configurations. Therefore, our configurations in the table can provide a reasonable understanding of our performance of experiments described below. In the *word count* application, since consistent word counting information is critical, *reliable* rather than *best effort* is designated as the Reliability QoS. For reliable communication, history samples are all kept in the reader's and writer's queues. As the Ownership QoS is set to *exclusive*, only one primary data writer among multiple data writers can publish samples to a data reader. If a sample has not arrived in 10 milliseconds, a deadline missing event occurs and the primary data writer is changed to the one which has the next highest ownership strength.

The results of experimental evaluations are presented to verify performance and failover overhead of our Remus-based solution in conjunction with DDS Ownership QoS. We experimented six cases shown in the Figure 46 to understand latency and failover overhead of running Remus and DDS Ownership QoS for the word count real-time application. The

experimental cases represent the combinatorial fail over cases in an environment selectively exploiting Remus and DDS Ownership QoS.

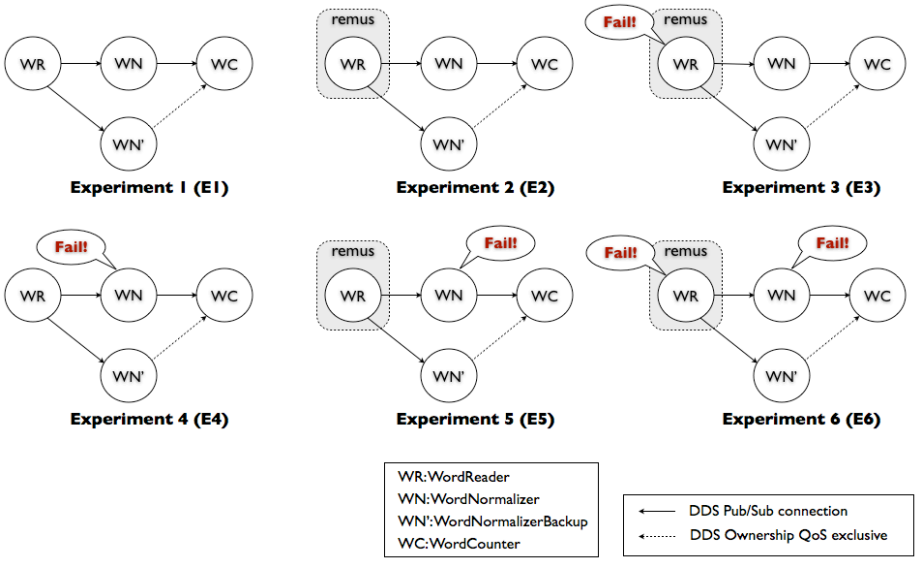

Figure 46: Experiments for the Case Study

Figure 47 depicts the results of Experiment E1 and E2 from Figure 46. Both the experiments have Ownership QoS setup as described above. Experiment E2 additionally has VM1 running the *WordReader* process, which is replicated to VM1' whose placement decision is made by our algorithm. The virtual machine VM1 is replicated using Remus high availability solution with the replication interval set to 40 milliseconds for all the experiments. This interval is also visibly the lowest possible latency for all the experiments, which has ongoing Remus replication. All the experiments depicted in Figure 46 involved a transfer of 8,000 samples from *WordReader* process on VM1 to *WordCount* process running on VM4. In the experiments E1 and E2, *WordNormalizer* processes run on VM2 and VM3 and incur the overhead of DDS Ownership QoS. In addition, experiment E2 has the overhead of Remus replication.

The graph in Figure 47 is a plot of average latency for each of the 80 samples set for a total of 8,000 samples transfer. For experiment E1 with no Remus replication, it

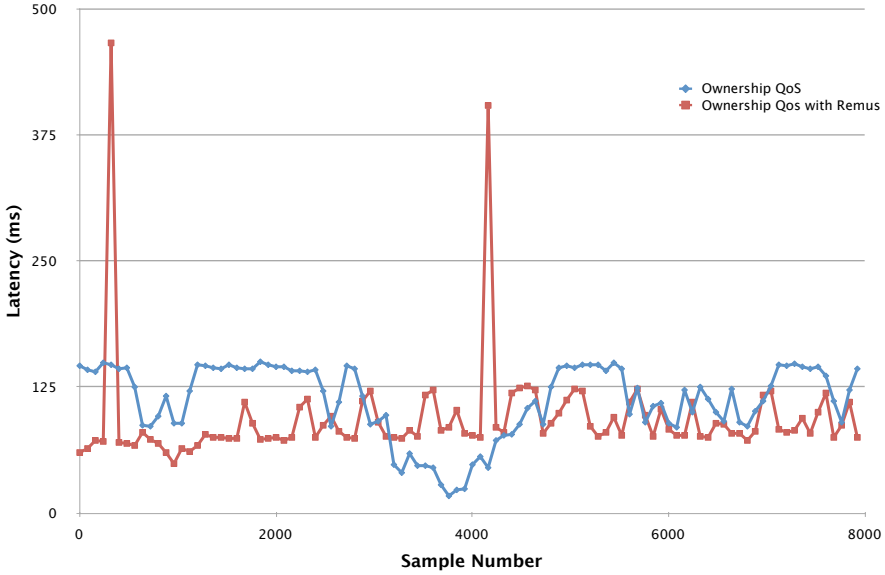

Figure 47: Latency Performance Impact of Remus Replication

was observed that the latency fluctuated within a range depending upon the queue size of *WordCounter* and each of *WordNormalizer* processes. For experiment E2 with Remus replication, the average latency for sample transfer did not have much deviation except for a few jitters. This is because of the fact that Remus replicates at a stable, predefined rate (here 40 ms), however, due to network delays or delay in checkpoint commit, we observed jitters. These jitters can be avoided by setting stricter deadline policies in which case, some samples might get dropped and they might need to be resent. Hence, in case of no failure, there is very little overhead for this soft real-time application.

Figure 48 is the result for experiment E3 where *WordReader* process on VM1 is replicated using Remus and it experienced a failure condition. Before the failure, it can be observed that the latencies were stable with few jitters due to the same reasons explained above. When the failure occurred, it took around 2 seconds for the failover to complete during which a few samples got lost. After the failover, no jitters were observed since Remus replication has not yet started for VM1', but the latency showed more variation as the

system was still stabilizing from the last failure. Thus, the high availability solution works for real-time applications even though a minor perturbation is present during the failover.

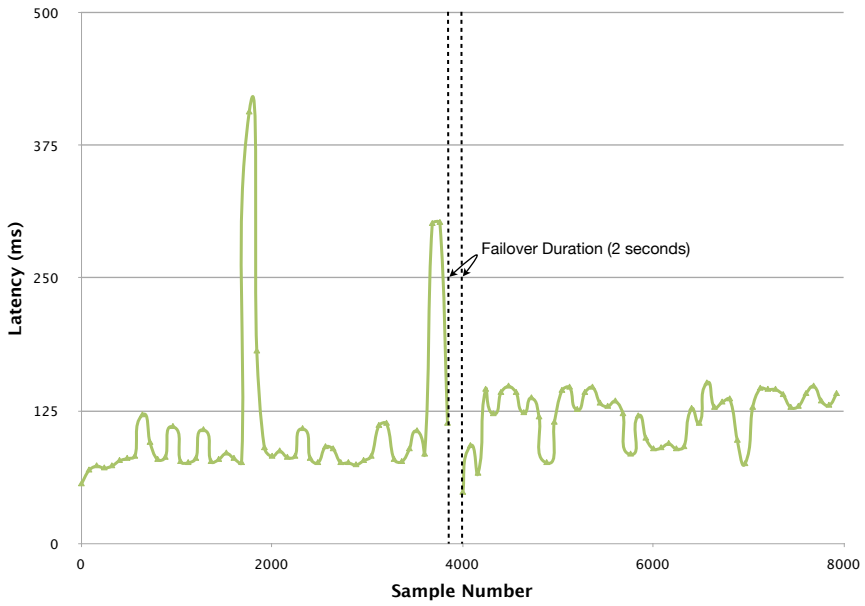

Figure 48: DDS Ownership QoS with Remus Failover

Table 7 represents the missed ratio for different failover experiments performed. In experiments E4 and E5, VM2 failed and the WordNormalizer process failed over to VM3. Since the DDS failover relied on publisher/subscriber mechanism, the number of lost samples is low. The presence of Remus replication process on WordReader process node VM1 did not have any adverse effect on the reliability of the system. However, in case of experiments E3 and E6, where Remus failover took place, the number of lost samples was higher since the failover duration is higher in case of Remus replication than DDS failover. These experiments show that ongoing Remus replication does not affect the performance of DDS failover, even though Remus failover is slower than DDS failover. However, since DDS does not provide any high availability for the source, infrastructure-level high availability provided by our Remus-based solution must be used.

Table 7: Failover Impact on Sample Missed Ratio

	Missed Samples (total of 8000)	Missed Samples Percentage (%)
Experiment 3	221	2.76
Experiment 4	33	0.41
Experiment 5	14	0.18
Experiment 6	549	6.86

IV.6 Concluding Remarks

As real-time applications move to the cloud, it becomes important for cloud infrastructures and middleware to implement algorithms that provide the QoS properties (*e.g.*, timeliness, high-availability, reliability) of these applications. In turn this requires support for algorithms and mechanisms for effective fault-tolerance and assuring application response times while simultaneously utilizing resources optimally. Thus, the desired solutions require a combination of algorithms for managing and deploying replicas of virtual machines on which the real-time applications are deployed in a way that optimally utilizes resources, and algorithms that ensure timeliness and high availability requirements. This chapter presented the architectural details of a middleware framework for a fault-tolerant cloud computing infrastructure that can automatically deploy replicas of VMs according to flexible algorithms defined by users. This chapter also presented performance impact of high availability solutions and deployment decisions by comparing latency of different cases to validate our approach. The following is a summary of the lessons we learned from this research and the empirical evaluations.

- **Remus with efficient placement incurs less overhead and fluctuation on latency.** Experimental results show using Remus incurs overheads on latency because Remus periodically synchronizes a primary VM with a backup machine and these overheads cannot be avoided. However, the problem is that if backup placement is randomly determined, latency values are increased more and fluctuate a lot because of resource

contention. As a result, if the backup placement is optimal, overheads occurred by high availability solutions can be minimized, and especially jitters can be reduced substantially.

- **VM level backup incurs more overheads than process(middleware) level backup, but it must be used for some cases when middleware does not support high availability.** Remus creates backups for all running processes on a VM, and it results in more overheads compared to a technique which realizes high availability at process-level. To experiment and compare the overheads, this chapter presented latency and missed sample ratio of Remus and DDS middleware which supports high availability at process-level. The experimental results show that latency is similar for both Remus and DDS when a failures does not happen. However, when a failure happens and failover process is needed, Remus causes more overheads in terms of the number of missed samples. Nevertheless, if application or middleware does not support high availability solution, VM level backup is required and it incurs acceptable overheads.

- **Timeliness can be improved with both real-time scheduling by hypervisor and resource allocation.** The work presented in this paper addresses just one dimension of a number of challenges that exist in supporting real-time application in the cloud. For example, scheduling of virtual machines (VMs) on the host operating system (OS) and in turn scheduling of applications on the guest OS of the VM in a way that assures application response times is a key challenge that needs to be resolved. Scheduling alone is not sufficient; the resource allocation problem must be addressed wherein physical resources including CPU, memory, disk and network must be allocated to the VMs in a way that will ensure that application QoS properties are satisfied. In doing so, traditional solutions used for hard real-time systems based on over-provisioning are not feasible because the cloud is an inherently shared infrastructure, and operates on the utility computing model.

CHAPTER V

MODEL-DRIVEN GENERATIVE FRAMEWORK FOR AUTOMATED OMG DDS PERFORMANCE TESTING IN THE CLOUD

V.1 Motivation

The OMG DDS is a general-purpose middleware supporting real-time pub/sub semantics for mission-critical applications. Specifically, the OMG DDS supports real-time, topic-based, data-centric, scalable, deterministic and anonymous pub/sub interaction semantics for large-scale distributed applications. To support the quality of service (QoS) requirements of a broad spectrum of application domains, the OMG DDS supports many QoS configuration policies (in the form of configuration parameters) that when used in different combinations determine the delivered end-to-end QoS properties.

An important consideration with DDS QoS policies is that not all QoS policies can be combined with each other since certain combinations tend to be incompatible with each other. Similarly, the parameter values chosen for specific QoS policies may tend to become inconsistent when combined. Both the incompatibility and inconsistency issues pose significant challenges for DDS application developers who must ensure that their deployed applications have compatible and consistent QoS configuration policies. This issue was covered in a previous work [33] that utilized model-driven engineering (MDE) [78] techniques to pinpoint existence of such errors at design-time.

V.1.1 Challenges

Challenge 1: Prediction of end-to-end performance impact with combined QoS policies Addressing these accidental challenges alone is not sufficient, however, towards realizing high confidence DDS-based applications. Every individual QoS policy tends to impact the end-to-end performance and behavior of the application in specific ways. When these

QoS policies are combined in various combinations, it is hard to predict the outcome on QoS of combining these policies. Such a problem is faced not just by application developers but also by the OMG DDS vendors themselves, who must have an in-depth knowledge of how various combinations of configuration parameters interact, and to address issues raised by their customers.

Challenge 2: Generation and deployment of lots of test cases It is not possible to expect an application developer or a vendor to manually write test cases that can experiment every QoS policy and all possible combinations of these QoS policies (along with their values), not to mention that they must also ensure that these combinations are valid. Even if one were to develop these large number of tests, executing them sequentially is time consuming, which impacts both the application developers who aim at getting their applications to market rapidly and vendors who must address customer problems in a timely manner.

V.1.2 Solution Approach

To address the combinatorial testing problem and limitations of sequential testing, this paper presents AUTOMATIC (AUTOmated Middleware Analysis and Testing In the Cloud), which is a framework we have developed that combines MDE techniques with multiple stages of generative capabilities. Specifically, AUTOMATIC provides a domain-specific modeling language that developers use to model their applications and QoS policies of interest. Generative tools synthesize essentially a product line of test cases, each testing different QoS policies for the same pub/sub business logic. A second set of generators synthesize cloud-based deployment logic. Finally, a testing framework automates the testing of the generated test cases in parallel in the cloud. Although a related effort called Expertus [37] uses aspect oriented weaving techniques for code generation and automated testing of applications for performance in the cloud, this effort does not address the QoS configuration combinations and their impact on performance that we address in this paper.

Although our research provides a framework of practical importance for the rapid testing of OMG DDS middleware-based applications, the core ideas behind the work can provide invaluable lessons for developers and engineers wishing to build their own testing and deployment frameworks for applications built using other platforms. In fact, the component-oriented design of our framework makes it possible to reuse parts of the generative capabilities, *e.g.*, the cloud deployment capabilities can be reused without being tightly coupled to DDS-specific test code.

This framework brings the following artifacts and merits.

- It provides a domain-specific modeling language (DSML) that enables developers and engineers to define OMG DDS-based testing scenarios with DDS QoS policies of interest and their acceptable range for the values of these parameters.
- It supports full automation beyond the modeling stage through multi-staging of generative capabilities, which helps to synthesize a family of test code variants that have a common business logic but differ in how the QoS policies are combined, and tested.
- It enables the testing of all the generated scenarios in parallel by exploiting cloud computing's elastic properties, and completely automating the test execution and collection of results.
- It supports a component-oriented, composable generative framework such that individual parts of the tool chain can potentially be used for other application scenarios and purposes.

V.2 Related Work

Expertus [38] is a code generation framework for distributed applications in clouds. The paper described similar vision and approach to ones of our framework. However, Expertus focuses on performance by different application platforms and cloud platforms, but our framework concentrates on application's non-functional configurations. Moreover, The

work exploited aspect-oriented software engineering to weave common and variable parts of code while our work uses model-driven engineering and model interpreters to generate and combine common and variable code elements. The paper did not provide actual performance of distributed applications, but our paper analyzes key performance indicators and characteristics of QoS policies with a demonstration with an example distributed pub/sub application.

DQML [33] is a modeling language to verify compatibility and consistency of DDS QoS configurations. The aim of DQML that is to generate DDS QoS configuration code is identical with our framework, but the ultimate goal of DQML is to check and validate compatibility and consistency of QoS policies while we bring in to focus on performance analysis with cloud platforms by mixed QoS configurations. Our framework also supports validation of QoS compatibility and consistency in a different way from DQML.

V.3 Design and Implementation

Figure 49 describes the overall architecture and workflow of our automated performance testing framework called AUTOMATIC. AUTOMATIC comprises three activity domains: *User*, *Test Automation System*, and *Cloud Infrastructure*. The *Modeling* and *Test Monitoring* functions included in the *User* domain should be conducted by a user who prototypes DDS applications and performs performance testing of the applications. In the *Test Automation System* domain, *Test Planning* and *Test Deployment* functions are carried out by predefined tools in our framework. When the *Test Planning* is completed and ready to be deployed in a testing infrastructure, a test environment is generated for our cloud infrastructure to emulate application testing. In short, users need to define their models of applications and test specifications with a modeling tool as inputs and obtain performance results with a monitoring tool as outputs of this framework.

The rest of this section describes each activity in detail including the performance monitoring capability.

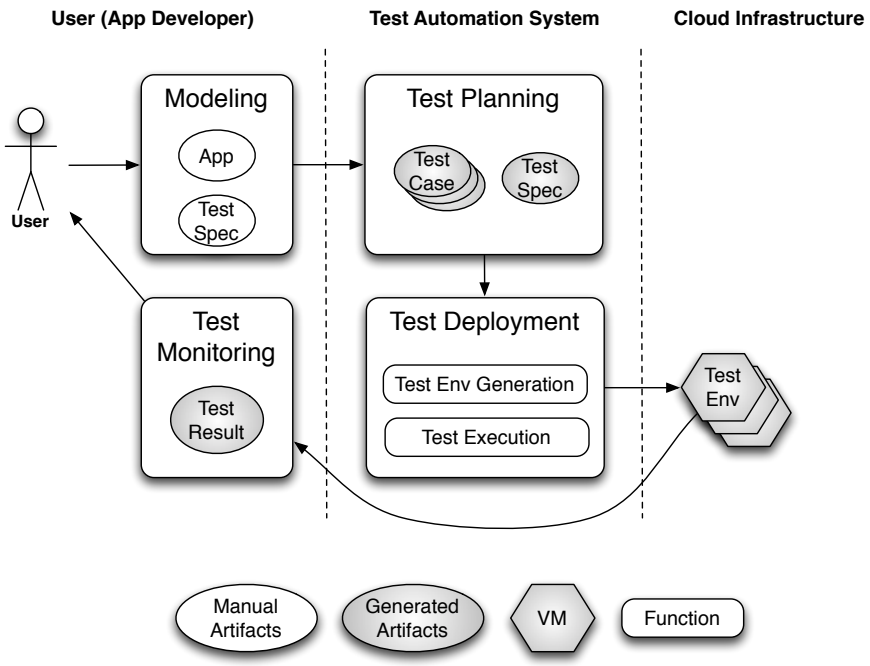

Figure 49: Framework Architecture

V.3.1 Domain-Specific Modeling Language (DSML)

We developed a DSML using the Generic Modeling Environment (GME) (`www.isis.vanderbilt.edu/projects/GME`) that supports modeling a DDS application for emulation and testing its performance for various combinations of DDS QoS policies. GME provides a meta-modeling environment to develop DSMLs for specific domains. Our meta-model includes modeling elements for all OMG DDS entities including Domain, Topic, Publisher, Subscriber, DataWriter, DataReader, QoS, and their connections. In DDS applications, a scope or operating region of an application is determined by the Domain, and applications are isolated by different Domain IDs. DDS applications publish or subscribe via DataWriters and DataReaders through associated Topics, and therefore in the meta-model the Topic and Type elements are contained in the Domain element and Topics and Types in the Domain are accessible by DataWriter and DataReader entities running in the same Domain. Moreover, the Domain contains a Participant element which is a concept to represent a processing unit for publishing or subscribing or both. Lastly, the modeling

capability to configure QoS policies for DDS entities is contained in the Domain element. Data communications between Participants are differentiated and identified by a Topic, so a TopicConnection element is required in the Domain model to be used by DataWriters and DataReaders in Participants.

Figure 50 shows an example application defined with our modeling language. This example application examines the throughput of the application publishing octet sequence data from a Participant containing a DataWriter to a Participant involving a DataReader. Each DataWriter and DataReader are placed under the Participant element and behaves as a communicating port between Participants.

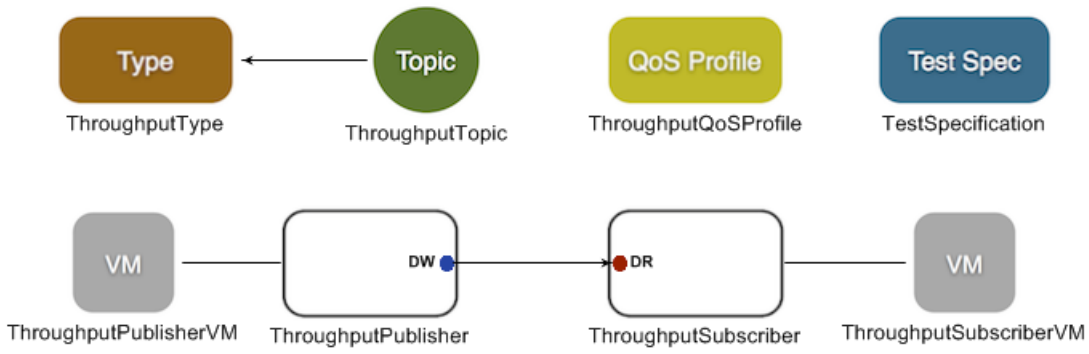

Figure 50: Example Domain-Specific Model of DDS Throughput Testing Application

The DDS Participants are deployed in virtual machines (VMs) for testing and each Participant in the model are connected to a VM element based on the deployment decision by users. In this example, each Participant is deployed in a different VM. The deployment plan (mapping of Participants and VMs) can be flexibly altered by users in the modeling language if users like to test with different deployment plans. Users can emulate their applications by setting analogous hardware specifications to find similar performance results in actual environment.

Communicating DataWriters and DataReaders are connected with directed lines which

indicate communications defined by a Topic. A Topic is shown in the top of this example model. If a name of a line is the same as the name of a Topic, it means DataWriters and DataReaders connected with the link communicate data by the Topic. Each data type of a Topic is determined by a *struct* like data type.

In the QoS Profile element, QoS policies used by DataWriters and DataReaders are contained. For example, Reliability QoS has two kinds of policies to determine the level of reliability: RELIABLE and BEST_EFFORT. History QoS also has two kinds of policies to set the number of history samples in a entity's cache: KEEP_ALL and KEEP_LAST. Some QoS policies need to set as numeric values such as history depth in History QoS. A QoS Profile element can be reused by multiple DataWriters and DataReaders. In our framework, QoS policies defined in a QoS Profile element are variations of generative artifacts, and the number of variations are determined by ranges of configuration parameters set by users.

Finally, the configurable parameters are set in the TestSpec element. In this element, test related information such as running duration of the test, and the number of test cases concurrently running is configured. A deployment tool uses this information to decide the number VMs in a test set and schedule the test operations.

V.3.2 Test Plan Generation

The *Test Planning* function traverses the modeled elements in a model instance via a model interpreter to generate executable applications and related test specification files. To traverse GME model elements, relevant APIs for interpreting GME models are provided and we utilized Unified Data Model (UDM) APIs [52] and the visitor pattern to develop our interpreter. Moreover, LEESA [85] was also adopted in our interpreter implementations to simplify complexity of the part for traversing UDM model elements.

Figure 51 shows an XML-based DDS application tree model transformed by the model interpreter based on Figure 50. Because the aim of our automatic testing framework is to analyze application performance by varying QoS configuration, elements under the QoS

Library are categorized into variable elements and the rest of the elements fall into the common elements category. This approach is conducive to using generative programming to realize a product line of test cases. The QoS Library embodies QoS elements for DataWriters and DataReaders. To demonstrate our framework with a simple example, we varied only the Reliability QoS. In this example, both DataWriter QoS and DataReader QoS have Reliability QoS. RELIABLE or BEST_EFFORT can be selected as a kind of the Reliability QoS.

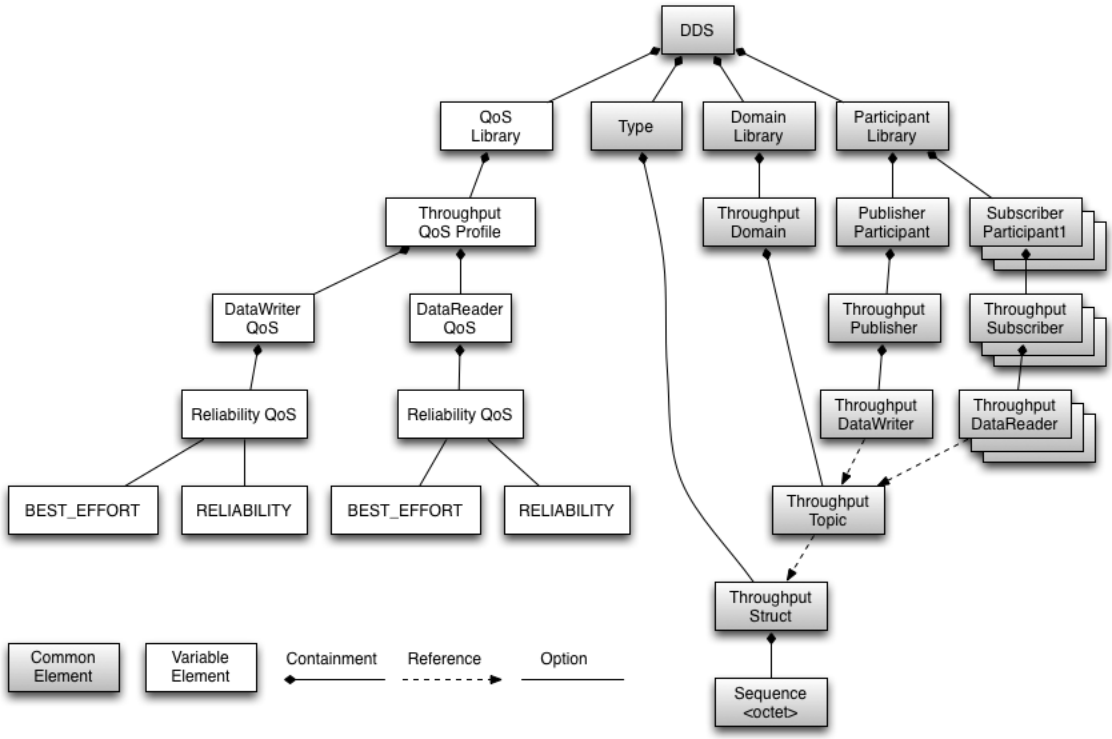

Figure 51: XML-based DDS Application Tree

The following procedure is used to form a tree shown in Figure 52 for all possible combinations of QoS configurations defined in the QoS Library. In the example, four test cases can be generated as each DataWriter and DataReader QoS has Reliability QoS that can choose from BEST_EFFORT and RELIABLE. Once the combination tree for variable

elements is complete, the combination tree is traversed with depth-first search to create trees for variables elements actually used by the applications for testing.

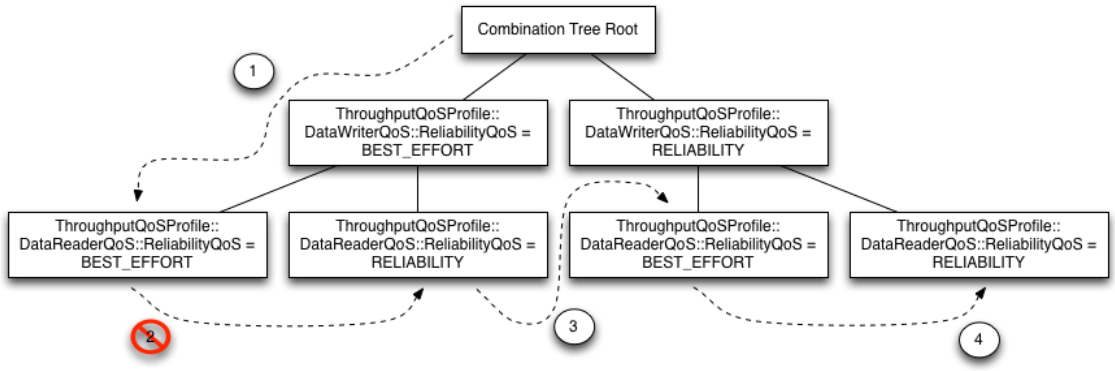

Figure 52: Variable Element Combination Tree

As a final outcome, four trees for variable elements are created as shown in Figure 53. The tree numbered 2 is discarded by the interpreter because the QoS configurations are not compatible. The reason is that if the DataWriter's Reliability QoS is BEST_EFFORT and DataReader's Reliability QoS is RELIABLE, then no communication between them is feasible according to the DDS specification.

There are three ways to check QoS incompatibility and inconsistency. First, it can be done by modeling environment with modeling constraint languages such as Object Constraint Language (OCL). The Distributed QoS Modeling Language (DQML) [33] introduced this approach to configure compatible QoS policies at design-time. Second, we can realize it by checking conditions in a modeling interpreter during traversing modeling elements and their attribute values. Our framework adopted this second approach. Third, QoS incompatibility can be detected by DDS middleware at run-time.

As the final step, the trees for variable elements are combined with the tree for common elements introduced in Figure 51, and the executable applications are generated.

V.3.3 Test Deployment

To deploy the XML-based DDS testing applications in a cloud-based testing infrastructure, specifications related to the deployment are also generated by the model interpreter. The specifications are composed of three parts: *Test Specification*, *VM Specification*, and *Application Specification*. The *Test Specification* describes the environment including a reference to the *VM Specification*, concurrency level, duration for test execution, publication period of publishers. Each test case is defined with an assigned ID and a referring specification file. The referred specification files have information about application's topology and the execution command.

The *VM Specificatio*n example describes required VMs for testing and information of VMs such as VM instance type and image. These specifications are fed into our deployment tool. VM instance types indicate specifications of VM such as the number of virtual CPUs, memory size, and storage capacity. According to the user-selected VM image and VM instance type, the *Test Env Generation* function deploys a proper VM in a cloud infrastructure. When the VM has booted up, a SSH connection is established and a test case application is sent to the VM over the SSH connection by the Test Execution function.

We implemented our deployment tool in Python 2.7 for the *Test Deployment function*. Our private cloud for testing adopted OpenStack as a cloud operating system, and the Python Boto library is exploited to control cloud resources via Amazon AWS APIs. Our tool utilized Paramiko (`http://www.lag.net/paramiko/` to establish SSH communications to the VMs.

The generated XML-specified application that is moved to the deployed VM is subsequently executed on that VM using a tool provided by RTI called the RTI Prototyper (`http://community.rti.com/content/page/download-prototyper`).

RTI Prototyper is a command-line executable application to operate DDS applications defined in XML. By aid of the RTI Prototyper, implementations of our model interpreter could be simplified by only formalized XML files need to be generated for complete DDS

applications. The RTI Prototyper supports configurable parameters to determine DomainParticipant required to run, running time, and publishing data interval. A full execution command with these parameters are automatically recored in Application Specification when the specification is generated by our interpreter, and it is used by the deployment tool for actual execution.

V.3.4 Test Monitoring

We employed another product from RTI called the RTI Monitor to detect DDS applications' performance while it is executing on the VM. The RTI Monitor is a tool to visualize monitoring data of applications. The RTI Monitor helps users to understand their systems easily via graphical interfaces and to verify behaviors of entities as expected. Moreover, it comes to the aid of improving performance throughput provided statistics such as CPU and memory usage, and throughput. The experimental results illustrated in Section V.4 were collected using this tool.

To use the RTI Monitor, DDS applications is demanded to use Monitoring library plug-in. The Monitoring library periodically checks and gets status of DDS entities and all entities related details are sent to the Monitor. The Monitoring library is configurable via DomainParticipant's Property QoS Policy, and our framework automatically produces the configuring QoS parts with a selectable attribute in the modeling environment.

V.4 Technology Validation

Our efforts at validating the claims in AUTOMATIC thus far have focused on a scenario where an application developer seeks to make appropriate tradeoffs trying to balance the conflicting requirements of reliability and timeliness. To that end, the experiment evaluates performance of an example DDS application by combining the Reliability, History and Deadline QoS policies. In this experiment, DDS applications use core libraries of RTI Connext DDS 5.0 (which is an implementation of OMG DDS) and executable scripts provided

with RTI Connext Prototyper 5.0. Our OpenStack-based cloud testbed employs KVM as a virtual machine (VM) hypervisor. Each VM machine type used in this experiment consists of 1 virtual CPU and 512 MB memory.

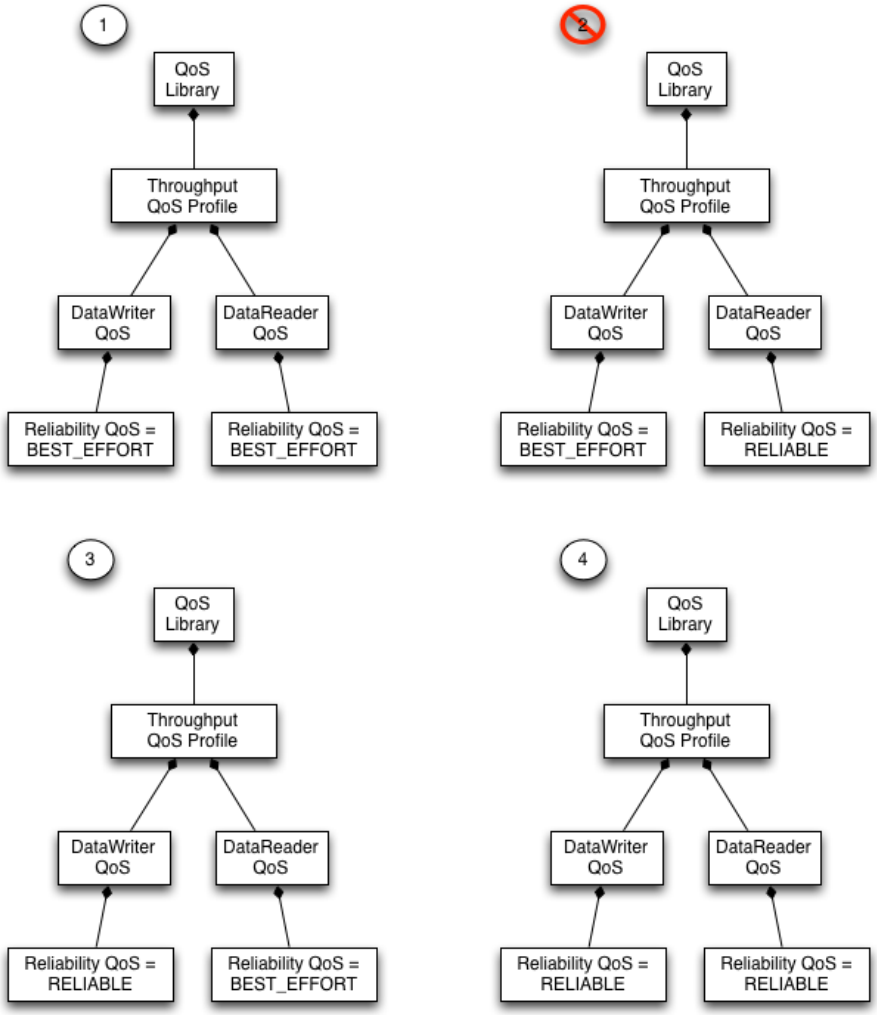

Figure 53: Variable Element Tree

In our example, the publisher periodically publishes a topic containing octet sequence typed data of 64K bytes to the subscriber. We chose a large packet size in the hope of congesting the network. The publishing period is decided by the Deadline QoS setting and was fixed at 1 millisecond. The purpose of this experiment is to understand deadline miss

rate for different Reliability QoS configurations. The HISTORY setting was KEEP_ALL, which means the publisher and subscriber hold on to all the data samples so they can be used for retransmissions when complete reliability is desired. The Reliability QoS setting is varied between RELIABLE (for eventual consistency) versus BEST_EFFORT (where no attempt is made to retry transmissions when samples are lost). The generated test cases are shown in Figure 53.

Figure 54 shows deadline miss counts of DataReader's (an entity on the subscriber side) in the test cases. If a sample is not arrived in a DataReader within 1 millisecond, it is counted as a missed deadline. Each test case runs for 4 minutes and values are monitored every 5 seconds. The X axis indicates time and Y axis presents deadline missed samples for 5 seconds.

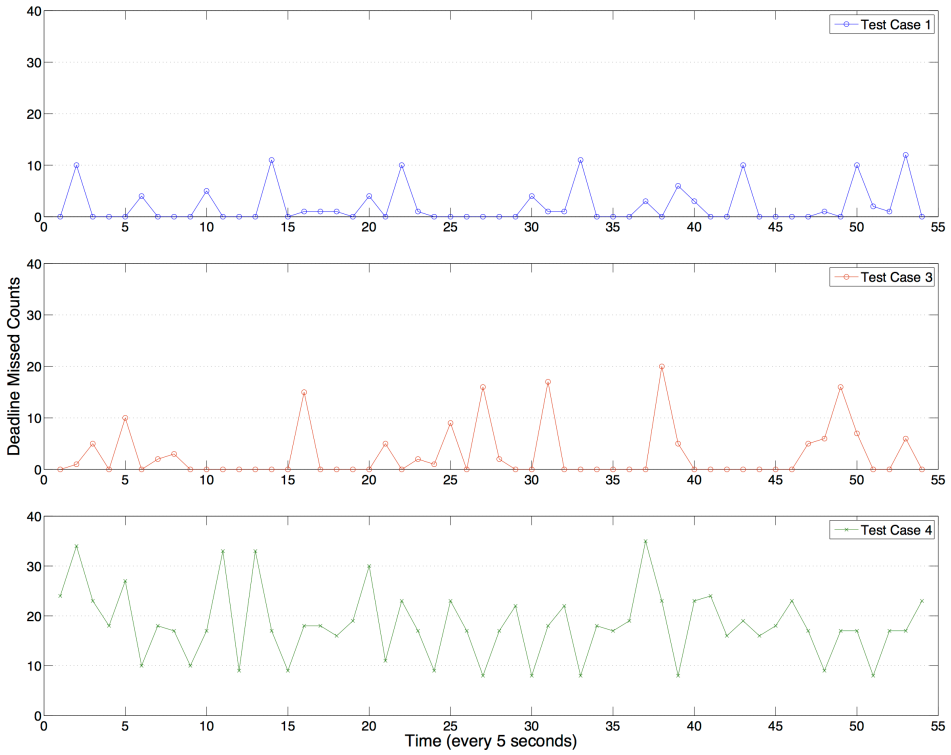

Figure 54: Deadline Miss Counts for Different Reliability QoS Settings

In test case 1, most samples do not miss the deadline and the range of the samples

spans up to 10 as maximum. If Reliability QoS is set to BEST_EFFORT, a DataWriter keeps publishing data regardless of the status of a DataReader and therefore it is beneficial to be used for applications demanding low latency. In a congested network environment, it would possibly lose samples, however, since our test network is not congested, there were no lost samples.

In test case 3, deadline miss counts are monitored from 9 to 35 where they keep occurring during the entire testing period. If the data cache of a DataWriter with the RELIABLE Reliability QoS is filled with unacknowledged samples, the DataWriter's write operation is blocked for a while to control the sending rate to avoid congestion which increases the latency of samples delivered. Accordingly, high latency causes deadline miss counts on the DataReader side. However, samples can reliably arrive at the DataReader due to the middleware supporting the retransmissions.

V.5 Concluding Remarks

Modern middleware, such as the OMG DDS, provide substantial flexibility to applications by virtue of supporting a large number of configuration options. These configuration options when combined in different ways can lead to vastly different performance and behavioral characteristics for the applications. Although some intuition is always available on the potential impact of individual configurations, and some guidelines do emerge after a few years of experience using multiple configurations on real applications (*e.g.*, `community.rti.com/best-practices`), application developers continue to face numerous challenges deciding the right combinations of options they must use for their application for the chosen deployment environments. It is infeasible for developers to manually create and test each possible scenario to understand the impact of the configuration options. To address these challenges in the context of OMG DDS middleware, this paper combines model-driven engineering (MDE) and generative programming techniques

to provide a tool called AUTOMATIC (AUTOmated Middleware Analysis and Testing In the Cloud).

The following is a summary of the lessons we learned from this work

- **MDE help application developers with intuitive abstractions to rapidly describe their testing scenarios**
- **Generative programming is needed since the test cases that combine configuration options can be considered a product line where the DDS application business logic remains common while the configurations can vary.**
- **Deployment and testing in the cloud is chosen as an approach because of its elastic nature where we can automate the parallel execution and collection of test statistics for a large number of generated tests from our tooling.**

Although the presented technology is showcased for the OMG DDS middleware, the principles behind AUTOMATIC are applicable to other middleware. Moreover, our technology has significant practical utility to both application developers and middleware vendors. Current artifacts in AUTOMATIC are available for download from `www.dre.vanderbilt.edu/~kyoungho/AUTOMATIC/`.

CHAPTER VI

CONCLUDING REMARKS

This doctoral dissertation is motivated by the need to address a variety of inherent and accidental challenges manifested in realizing the true potential of Industrial Internet of Things (IIoT). Specifically, we focus on the issues within the plant and enterprise levels of the IIoT conceptual space defined by the Industrial Internet Consortium (IIC) [16]. Our ideas have been implemented and evaluated in the context of the OMG's Data Distribution Service (DDS) data-centric, publish/subscribe (pub/sub) technology.

To address the inherent complexities stemming from scalable discovery of publishers and subscribers, we proposed a scalable DDS discovery protocol called *Content-based Filtering Discovery Protocol (CFDP)*. This protocol was shown to reduce the computing and network resources in the discovery phase to realize scalability in a large scale system. Through this protocol, DDS applications can save on the number of resources used, and achieve fast discovery of communication endpoints.

To realize QoS-enabled data-centric pub/sub communications over WANs, we described *A Cloud-enabled Coordination Service for Internet-scale OMG DDS Applications*. Our solution called *PubSubCoord* includes harnessing ZooKeeper and DDS Routing Service to enable DDS applications to communicate in a WAN in a scalable and automatic way in terms of discovery and pub/sub communications. The *PubSubCoord* architecture provides load balancing and fault-tolerance for brokers located in the cloud as well as flexible pub/-sub overlays based on deadlines dictated by communication requirements.

To support fault tolerant and soft real-time enterprise-level IIoT computing tasks in the cloud, we presented the *A Cloud Middleware Assuring Performance and High Availability of Real-time Applications* middleware solution. Specifically, this middleware integrates a VM failover solution such as Remus to guarantee availability of cloud applications when a

failure happens and provides a way to use pluggable placement algorithms for VM backups to reduce the number of underlying physical machines as well as assure timeliness avoiding contention of shared resources.

Validating various hypotheses in the context of IIoT is hard due to the large number of accidental complexities stemming from the flexibility and configurability of the pub/sub platforms. This research described the *Model-driven Generative Framework for Automated OMG DDS Performance Testing in the Cloud*. This work utilizes MDE tools such as GME to provide intuitive interfaces for users to define test cases and generative techniques to automatically generate configuration and implementation scripts. Additionally, cloud infrastructures are exploited to deploy and execute a number of test cases concurrently by leveraging the elastic properties of the cloud.

A number of follow on research efforts are possible stemming from our research contributions. In particular, we have identified the following extensions for *CFDP* and *PubSubCoord*.

- **CFDP**

  SDP can seamlessly use multicast because only a single multicast address is used for discovery of all participants. For CFDP, however, each content filter used by CFDP will require a separate multicast address. To overcome this limitation, we suggest enhancing CFDP to support multicast thereby reducing the number of discovery messages sent by delegating the overhead to network switches. This approach will group peers with a set of multicast addresses by topic names so that built-in discovery DataWriters will publish data only to assigned multicast channels (groups).

  The current CFDP filters discovery messages based on topic names, which limits its scalability in a system where most data streams are differentiated by a key of a topic, rather than by a topic itself. More work is needed to enhance CFDP to filter discovery messages based on topic names as well as keys. This enhancement should

provide performance benefits for DRE systems that contain numerous endpoints and instances with a single or less number of topics.

- **PubSubCoord**

  Although PubSubCoord supports dynamic joining and leaving of endpoints, the system incurs some delay in executing the coordination actions to converge to a new state of routes. We have not conducted experiments to evaluate this capability, which may shed light on potential performance optimizations.

  Despite the load balancing, it is possible that a routing broker has to maintain connections to many different edge brokers. So fine-grained load balancing mechanisms(*e.g.*, load balancing by keys) are needed that do not adversely impact performance for systems having many endpoints communicating via a small set of topics.

  In our work we described the mechanism by which multi-path overlays can be instantiated for delay-sensitive dissemination paths. Since not every path can be a delay-sensitive path, we need some higher level policy mechanism to decide what characterizes a delay-sensitive path.

  Our experiments were conducted inside an emulated internet-scale network using virtual LANs. This arrangement currently illustrates a large but flat network hierarchy. Although we created scripts to automate many of the tasks in our test harness, more automation is needed to create more complex network hierarchies and evaluate our solution.

The following sections present a summary of research contributions, and a list research publications where these research contributions have been disseminated to the scientific community.

VI.1 Summary of Research Contributions

- **Scalability of discovery protocol for pub/sub middleware**

    1. Design of CFDP which is a scalable and efficient DDS endpoint discovery protocol employing content-based filtering to conserve computing, memory, and network resources used in the DDS discovery process.
    2. Implementation of CFDP prototype on top of a popular DDS middleware implementation.
    3. Empirical results conducted in a testbed to evaluate the performance and resource usage of CFDP compared with SDP.

- **Coordination service pub/sub middleware in WANs**

    1. Design scalable and dynamic discovery and coordination service for DDS systems over WANs called PubSubCoord that enables elastic cloud-based brokers with supporting load balancing and fault-tolerance.
    2. Supporting deadline-aware overlays that is beneficial to pub/sub endpoints requiring strict deadlines by establishing multi-paths.
    3. Empirical results to evaluate the scalability of PubSubCoord in terms of discovery as well as data dissemination.

- **Fault-tolerant and time-sensitive cloud computing infrastructures**

    1. Architecture for fault-tolerant framework that can be used to automatically deploy replicas of virtual machines in data centers in a way that optimizes resources while assuring availability and responsiveness.
    2. Design of a pluggable framework within the fault-tolerant architecture that enables plugging in different placement algorithms for virtual machine replica deployment.

3. Experimental results using a case study that involves a specific replica placement algorithm to evaluate effectiveness of this architecture.

- **Testing performance of different combinations of QoS configurations**

    1. A DSML that enables developers and engineers to define DDS-based testing scenarios with QoS policies of interest and their acceptable range for the values of these parameters.
    2. Full automation beyond the modeling stage through multi-staging of generative capabilities, which helps to synthesize a family of test code variants that have a common business logic but differ in how the QoS policies are combined, and tested.
    3. Enabling the testing of all the generated scenarios in parallel by exploiting cloud computing's elastic properties, and completely automating the test execution and collection of results.

VI.2 Summary of Publications

Journal Publications

1. Kyoungho An, Shashank Shekhar, Faruk Caglar, Aniruddha Gokhale, and Shivakumar Sastry, "A Cloud Middleware for Assuring Performance and High Availability of Soft Real-time Applications", The Elsevier Journal of Systems Architecture (JSA): Embedded Systems Design, 2014.

Book Chapters

1. Kyoungho An, Adam Trewyn, Aniruddha Gokhale and Shivakumar Sastry, "Design and Transformation of Domain-specific Language for Reconfigurable Conveyor Systems", Book chapter in Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, IGI Global publishers, Editor: Marjan Mernik, 2012.

Conference Publications

1. Kyoungho An, Sumant Tambe, Paul Pazandak, Gerardo Pardo-Castellote, Aniruddha Gokhale and Douglas Schmidt, "Content-based Filtering Discovery Protocol (CFDP): Scalable and Efficient OMG DDS Discovery Protocol", 8th ACM International Conference on Distributed Event-Based Systems (DEBS 2014), Mumbai, India, May 26-29, 2014.

2. Kyoungho An, Takayuki Kuroda, Aniruddha Gokhale, Sumant Tambe, and Andrea Sorbini, "Model-driven Generative Framework for Automated DDS Performance Testing in the Cloud", 12th ACM International Conference on Generative Programming: Concepts & Experiences (GPCE 2013), Indianapolis, IN, Oct 27-28, 2013.

3. Kyoungho An, "Resource Management and Fault Tolerance Principles for Supporting Distributed Real-time and Embedded Systems in the Cloud", 9th Middleware

Doctoral Symposium (MDS 2012), co-located with ACM/IFIP/USENIX 13th International Conference on Middleware (Middleware 2012), Montreal, Quebec, Canada, Dec 3-7, 2012.

4. Kyoungho An, Adam Trewyn, Aniruddha Gokhale and Shivakumar Sastry, "Model-driven Performance Analysis of Reconfigurable Conveyor Systems used in Material Handling Applications", Second ACM/IEEE International Conference on Cyber Physical Systems (ICCPS 2011), Chicago, IL, Apr 11-14, 2011.

5. Anushi Shah, Kyoungho An, Aniruddha Gokhale and Jules White, "Maximizing Service Uptime of Smartphone-based Distributed Real-time and Embedded Systems", 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2011), Newport Beach, CA, Mar 28-31, 2011.

Workshop, Work in Progress, and Poster Publications

1. Kyoungho An and Aniruddha Gokhale, "A Cloud-enabled Coordination Service for Internet-scale OMG DDS Applications", Poster paper at the 8th ACM International Conference on Distributed Event-Based Systems (DEBS 2014), Mumbai, India, May 26-29, 2014.

2. Shashank Shekhar, Faruk Caglar, Kyoungho An, Takayuki Kuroda, Aniruddha Gokhale and Swapna Gokhale, "A Model-driven Approach for Price/Performance Tradeoffs in Cloud-based MapReduce Application Deployment", MODELS 2013 workshop on Model-Driven Engineering for High Performance and CLoud computing (MDHPCL 2013), Miami FL, Sep 29, 2013.

3. Kyoungho An and Aniruddha Gokhale, "Model-driven Performance Analysis and Deployment Planning for Real-time Stream Processing", Work-in-Progress (WiP) session at 19th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS 2013), Philadelphia PA, Apr 9-11, 2013.

4. Faruk Caglar, Shashank Shekhar, Kyoungho An and Aniruddha Gokhale, "WiP Abstract: Intelligent Power- and Performance-aware Tradeoffs for Multicore Servers in Cloud Data Centers", Work-in-Progress (WiP) session at 4th ACM/IEEE International Conference on Cyber Physical Systems (ICCPS 2013), Philadelphia PA, Apr 9-11, 2013.

5. Kyoungho An, Faruk Caglar, Shashank Shekhar and Aniruddha Gokhale, "A Framework for Effective Placement of Virtual Machine Replicas for Highly Available Performance-sensitive Cloud-based Applications", RTSS 2012 workshop on Real-time and Distributed Computing in Emerging Applications (REACTION 2012), San Juan, Puerto Rico, Dec 4-7, 2012.

6. Kyoungho An, Subhav Pradhan, Faruk Caglar and Aniruddha Gokhale, "A Publish/Subscribe Middleware for Dependable and Real-time Resource Monitoring in the Cloud", Middleware 2012 workshop on Secure and Dependable Middleware for Cloud Monitoring and Management (SDMCMM 2012), Montreal, Quebec, Canada, Dec 3-7, 2012.

7. Kyoungho An, "Strategies for Reliable, Cloud-based Distributed Real-time and Embedded Systems", Extended abstract for PhD Forum in 31st IEEE International Symposium on Reliable Distributed Systems (SRDS 2012), Irvine, CA, Oct 8-11, 2012.

8. Faruk Caglar, Kyoungho An, Aniruddha Gokhale and Tihamer Levendovszky, "Transitioning to the Cloud? A Model-driven Analysis and Automated Deployment Capability for Cloud Services", MODELS 2012 workshop on Model-Driven Engineering for High Performance and CLoud computing (MDHPCL 2012), Innsbruck, Austria, Sep 30 - Oct 5, 2012.

Technical Reports

1. Shweta Khare, Sumant Tambe, Kyoungho An, Aniruddha Gokhale, Paul Pazandak, "Scalable Reactive Stream Processing Using DDS and Rx: An Industry-Academia Collaborative Research Experience", ISIS Technical Report, no.ISIS-14-103: Institute for Software Integrated Systems, Vanderbilt University, Nashville TN, April, 2014.

2. Kyoungho An, Sumant Tambe, Andrea Sorbini, Sheeladitya Mukherjee, Javier Povedano-Molina, Michael Walker, Nirjhar Vermani, Aniruddha Gokhale, and Paul Pazandak, "Real-time Sensor Data Analysis Processing of a Soccer Game Using OMG DDS Publish/Subscribe Middleware", ISIS Technical Report, no.ISIS-13-102: Institute for Software Integrated Systems, Vanderbilt University, Nashville TN, June, 2013.

Submitted Papers

1. Kyoungho An, Takayuki Kuroda, and Aniruddha Gokhale, A Coordination and Discovery Service for QoS-enabled Data-Centric Publish/Subscribe in Wide Area Networks, 35th IEEE International Conference on Distributed Computing Systems (ICDCS 2015), Columbus, OH, June 29-July 2, 2015.

2. Shweta Khare, Kyoungho An, Aniruddha Gokhale, and Sumant Tambe, Functional Reactive Stream Processing for Data-centric Publish/Subscribe: Experiences using .NET Reactive Extensions with OMG Data Distribution Service, 9th ACM International Conference on Distributed Event-Based Systems (DEBS 2015), Oslo, Norway, June 29-July 3, 2015.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[3] Kevin Ashton. That 'internet of things' thing. *RFiD Journal*, 22(7):97–114, 2009.

[4] Roberto Baldoni, Mariangela Contenti, and Antonino Virgillito. The evolution of publish/subscribe communication systems. In *Future directions in distributed computing*, pages 137–141. Springer, 2003.

[5] Debasis Bandyopadhyay and Jaydip Sen. Internet of things: Applications and challenges in technology and standardization. *Wireless Personal Communications*, 58(1):49–69, 2011.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[7] Paolo Bellavista, Antonio Corradi, and Andrea Reale. Quality of service in wide scale publish-subscribe systems. *IEEE Communications Surveys & Tutorials*, 2014.

[8] A. Beloglazov and R. Buyya. Energy efficient allocation of virtual machines in cloud data centers. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 577–578. Ieee, 2010.

[9] JO Berkey and PY Wang. Two-dimensional finite bin-packing algorithms. *Journal of the operational research society*, pages 423–429, 1987.

[10] Luis M Camarinha-Matos, João Goes, Luís Gomes, and João Martins. Contributing to the internet of things. In *Technological Innovation for the Internet of Things*, pages 3–12. Springer, 2013.

[11] Lewis Carroll. Ieee 1588 precision time protocol (ptp). `http://www.eecis.udel.edu/~mills/ptp.html`, 2012.

[12] Nuno Carvalho, Filipe Araujo, and Luis Rodrigues. Scalable qos-based event routing in publish-subscribe systems. In *Network Computing and Applications, Fourth IEEE International Symposium on*, pages 101–108. IEEE, 2005.

[13] Antonio Carzaniga, David S Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, 2001.

[14] M.K. Chippa, S. M. Whalen, S. Sastry, and F. Douglas. Goal-seeking Framework for Empowering Personalized Wellness Management. In *(POSTER) in Workshop on Medical Cyber Physical Systems, CPSWeek, April*, 2013.

[15] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 14–25. ACM, 2007.

[16] Industrial Internet Consortium. Industrial internet reference architecture version 1.1. http://www.iiconsortium.org, 2015.

[17] A. Corradi, L. Foschini, J. Povedano-Molina, and J.M. Lopez-Soler. DDS-enabled Cloud Management Support for Fast Task Offloading. In *IEEE Symposium on Computers and Communications (ISCC '12)*, pages 67–74, July 2012.

[18] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. USENIX Association, 2008.

[19] Christian Esposito, Domenico Cotroneo, and Aniruddha Gokhale. Reliable publish/-subscribe middleware for time-sensitive internet-scale applications. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 16. ACM, 2009.

[20] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computer Survey*, 35:114–131, June 2003.

[21] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[22] Peter C Evans and Marco Annunziata. Industrial internet: Pushing the boundaries of minds and machines. *General Electric*, page 21, 2012.

[23] J. Fontán, T. Vázquez, L. Gonzalez, RS Montero, and IM Llorente. Opennebula: The open source virtual machine manager for cluster computing. In *Open Source Grid and Cluster Software Conference*, 2008.

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns:*

Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[25] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[26] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the internet of things to the web of things: Resource-oriented architecture and best practices. In *Architecting the Internet of Things*, pages 97–129. Springer, 2011.

[27] Akram Hakiri, Pascal Berthou, Aniruddha Gokhale, Douglas Schmidt, and Thierry Gayraud. Supporting End-to-end Scalability and Real-time Event Dissemination in the OMG Data Distribution Service over Wide Area Networks. *Elsevier Journal of Systems Software (JSS)*, 86(10):2574–2593, October 2013.

[28] Akram Hakiri, Pascal Berthou, Aniruddha Gokhale, Douglas C Schmidt, and Gayraud Thierry. Supporting end-to-end scalability and real-time event dissemination in the omg data distribution service over wide area networks. *Submitted to Elsevier Journal of Systems Software (JSS)*, 2013.

[29] Stephan Haller, Stamatis Karnouskos, and Christoph Schroth. *The internet of things in an enterprise context*. Springer, 2009.

[30] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service. *Sun Microsystems Inc., Santa Clara, CA*, 2002.

[31] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[32] Joe Hoffert, Shanshan Jiang, and Douglas C Schmidt. A Taxonomy of Discovery Services and Gap Analysis for Ultra-large Scale Systems. In *Proceedings of the 45th annual southeast regional conference*, pages 355–361. ACM, 2007.

[33] Joe Hoffert, Douglas Schmidt, and Aniruddha Gokhale. A qos policy configuration modeling language for publish/subscribe middleware platforms. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 140–145. ACM, 2007.

[34] K.-Y. Hou, M. Uysal, A. Merchant, K. G. Shin, and S. Singhal. Hydravm: Low-cost, transparent high availability for virtual machines. Technical report, HP Laboratories, 2011.

[35] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[36] C. Hyser, B. McKee, R. Gardner, and B.J. Watson. Autonomic virtual machine placement in the data center. *Hewlett Packard Laboratories, Tech. Rep. HPL-2007-189*, 2007.

[37] D. Jayasinghe, G. Swint, S. Malkowski, J. Li, Qingyang Wang, Junhee Park, and C. Pu. Expertus: A Generator Approach to Automate Performance Testing in IaaS Clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 115–122, 2012.

[38] Deepal Jayasinghe, Galen Swint, Simon Malkowski, Jack Li, Qingyang Wang, Junhee Park, and Calton Pu. Expertus: A generator approach to automate performance testing in iaas clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 115–122. IEEE, 2012.

[39] Cullen Jennings, Salman Baset, Henning Schulzrinne, Bruce Lowekamp, and Eric Rescorla. Resource Location and Discovery (RELOAD) Base Protocol. *REsource*, 2013.

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

[41] Minkyong Kim, Kyriakos Karenos, Fan Ye, Johnathan Reason, Hui Lei, and Konstantin Shagin. Efficacy of techniques for responsiveness in a wide-area publish/-subscribe system. In *Proceedings of the 11th International Middleware Conference Industrial track*, pages 40–45. ACM, 2010.

[42] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[43] Valerie Lampkin, Weng Tat Leong, Leonardo Olivera, Sweta Rawat, Nagesh Subrahmanyam, Rong Xiang, Gerald Kallas, Neeraj Krishna, Stefan Fassmann, Martin Keen, et al. *Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry*. IBM Redbooks, 2012.

[44] Edward A Lee. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE, 2008.

[45] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubrahmanian, K. Talwar, L. Uyeda, and U. Wieder. Validating heuristics for virtual machines consolidation. *Microsoft Research, MSR-TR-2011-9*, 2011.

[46] Ming Li, Fan Ye, Minkyong Kim, Han Chen, and Hui Lei. A scalable and elastic publish/subscribe service. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1254–1265. IEEE, 2011.

[47] libvirt. libvirt:the virtualization api. http://libvirt.org, 2013.

[48] Steve Lohr. The age of big data. *New York Times*, 11, 2012.

[49] Jose M Lopez-Vega, Gonzalo Camarillo, Javier Povedano-Molina, and Juan M Lopez-Soler. RELOAD extension for data discovery and transfer in data-centric publish–subscribe environments. *Computer Standards & Interfaces*, 36(1):110–121, 2013.

[50] Jose M Lopez-Vega, Javier Povedano-Molina, Gerardo Pardo-Castellote, and Juan M Lopez-Soler. A content-aware bridging service for publish/subscribe environments. *Journal of Systems and Software*, 86(1):108–124, 2013.

[51] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A Survey and Comparison of Peer-to-peer Overlay Network Schemes. *IEEE Communications Surveys and Tutorials*, 7(1-4):72–93, 2005.

[52] Endre Magyari, Arpad Bakay, Andras Lang, Tamas Paka, Attila Vizhanyo, A Agrawal, and Gabor Karsai. Udm: An infrastructure for implementing domain-specific modeling languages. In *The 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA*, volume 3, 2003.

[53] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, pages 237–250. ACM, 2010.

[54] L. Northrop, P. Feiler, R.P. Gabriel, J. Goodenough, R. Linger, R. Kazman, D. Schmidt, K. Sullivan, and K. Wallnau. Ultra-Large-Scale Systems: Software Challenge of the Future. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, June 2006.

[55] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-source Cloud-computing System. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131. IEEE Computer Society, 2009.

[56] OASIS. Mqtt version 3.1.1. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`, 2014.

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

[58] OCI. Open DDS Developer's Guide. `http://download.ociweb.com/OpenDDS/OpenDDS-latest.pdf`, 2013.

[59] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus: an architecture for extensible distributed systems. In *ACM SIGOPS Operating Systems*

Review, volume 27, pages 58–68. ACM, 1994.

[60] OMG. The Data Distribution Service Specification, v1.2. `http://www.omg.org/spec/DDS/1.2`, 2007.

[61] OMG. The Real-Time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification, V2.1. `http://www.omg.org/spec/DDS-RTPS/2.1`, 2010.

[62] openstack. openstack.org. http://www.openstack.org, 2013.

[63] G. Pardo-Castellote. Omg data-distribution service: Architectural overview. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 200–206. IEEE, 2003.

[64] Gerardo Pardo-Castellote, Bert Farabaugh, and Rick Warren. An introduction to dds and data-centric communications. *RTI, Aug*, 2005.

[65] Peter R Pietzuch and Jean M Bacon. Hermes: A distributed event-based middleware architecture. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 611–618. IEEE, 2002.

[66] PrismTech. Messaging technologies for the industrial internet and the internet of things whitepaper. http://www.prismtech.com/sites/default/files/documents/Messaging-Comparison-Whitepaper-July2014.pdf, 2014.

[67] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 51–58. IEEE, 2010.

[68] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

[69] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-scale Peer-to-peer Systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[70] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Networked group communication*, pages 30–43. Springer, 2001.

[71] RTI. RTI Connext DDS Performance Benchmarks - Latency and Throughput. `http://www.rti.com/products/dds/benchmarks-cpp-linux.html`.

[72] RTI. Limited-Bandwidth Plug-ins for DDS. `http://www.rti.com/docs/`

DDS_Over_Low_Bandwidth.pdf, 2011.

[73] RTI. RTI Connext DDS User's Manual. http://community.rti.com/rti-doc/510/ndds.5.1.0/doc/pdf/RTI_CoreLibrariesAndUtilities_UsersManual.pdf, 2013.

[74] RTI. RTI Routing Service User's Manual. http://community.rti.com/rti-doc/510/RTI_Routing_Service_5.1.0/doc/pdf/RTI_Routing_Service_UsersManual.pdf, 2013.

[75] Javier Sanchez-Monedero, Javier Povedano-Molina, Jose M Lopez-Vega, and Juan M Lopez-Soler. Bloom Filter-based Discovery Protocol for DDS Middleware. *Journal of Parallel and Distributed Computing*, 71(10):1305–1317, 2011.

[76] Daniel J. Scales, Mike Nelson, and Ganesh Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *Operating Systems Review*, 44(4):30–39, 2010.

[77] D.C. Schmidt. Middleware for real-time and embedded systems. *Communications of the ACM*, 45(6):43–48, 2002.

[78] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[79] Douglas C Schmidt. Accelerating the Industrial Internet with the OMG Data Distribution Service. http://www.rti.com/whitepapers/OMG_DDS_Industrial_Internet.pdf, 2014.

[80] Thirunavukkarasu Sivaharan, Gordon Blair, and Geoff Coulson. Green: A configurable and re-configurable publish-subscribe middleware for pervasive computing. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 732–749. Springer, 2005.

[81] GE Software. The case for an industrial big data platform. https://www.gesoftware.com/Industrial_Big_Data_Platform.pdf, 2013.

[82] GE Software. The case for an industrial big data platform. https://www.gesoftware.com/sites/default/files/Industrial_Big_Data_Platform.pdf, 2013.

[83] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[84] Teresa M. Takai. Cloud Computing Strategy. Technical report, Department of Defense Office of the Chief Information Officer, July 2012.

[85] Sumant Tambe and Aniruddha Gokhale. Leesa: Embedding strategic and xpath-like object structure traversals in c++. In *Domain-Specific Languages*, pages 100–124. Springer, 2009.

[86] Y. Tamura, K. Sato, S. Kihara, and S. Moriai. Kemari: Virtual machine synchronization for fault tolerance. *In USENIX 2008 Poster Session*, 2008.

[87] Omesh Tickoo, Ravi Iyer, Ramesh Illikkal, and Don Newell. Modeling virtual machine performance: challenges and approaches. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):55–60, 2010.

[88] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89, 2006.

[89] Nanbor Wang, Douglas C Schmidt, Hans van't Hag, and Angelo Corsaro. Toward an Adaptive Data Distribution Service for Dynamic Large-scale Network-centric Operation and Warfare (NCOW) Systems. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1–7. IEEE, 2008.

[90] David S Watson, Mary Ann Piette, Osman Sezgen, Naoya Motegi, and Laurie Ten Hope. Machine to machine (m2m) technology in demand responsive commercial buildings. 2004.

[91] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 39–48, New York, NY, USA, 2011. ACM.

[92] Kaiwen Zhang and Hans-Arno Jacobsen. Sdn-like: The next generation of pub/sub. *arXiv preprint arXiv:1308.0056*, 2013.

[93] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. ACM.