# Making DDS Really Real-Time with OpenFlow

Hyon-Young Choi
hyonchoi@cis.upenn.edu

Andrew L. King
kingand@cis.upenn.edu

Insup Lee
lee@cis.upenn.edu

Department of Computer & Information Science
University of Pennsylvania
Philadelphia, USA

## ABSTRACT

An increasing amount of distributed real-time systems and other critical infrastructure now rely on Data Distribution Service (DDS) middleware for timely dissementation of data between system nodes. While DDS has been designed specifically for use in distributed real-time systems and exposes a number of QoS properties to programmers, DDS fails to lift time fully into the programming abstraction. The reason for this is simple: DDS cannot directly control the underlying network to ensure that messages always arrive to their destination on time.

In this paper we describe a prototype system that uses the OpenFlow SDN protocol to automatically and transparently learn the QoS requirements of DDS participants. Based on the QoS requirements, our system will manipulate the low-level packet forwarding rules in the network to ensure that the QoS requirements are always satisfied.

We use real OpenFlow hardware to evaluate how well our prototype is able to manage network contention and guarantee the requested QoS. Additionally, we evaluate how well the reliability and resilience features of a real DDS implementation is able to compensate for network contention on an unmanaged (*i.e.,* normal) ethernet. To the best of our knowledge, this is the first evaluation of performance DDS under extreme network contention conditions.

## CCS Concepts

•**Computer systems organization** → **Real-time operating systems; Real-time system specification;** *Peer-to-peer architectures; Reconfigurable computing; Maintainability and maintenance;* •**Networks** → *Network experimentation;*

## Keywords

Real-Time, Distributed Systems, Publish-Subscribe, Software Defined Networking, QoS Management



*EMSOFT'16, October 01-07 2016, Pittsburgh, PA, USA*



ISBN 978-1-4503-4485-2/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2968478.2968479

## 1. INTRODUCTION

The development of distributed real-time software systems is a complex and involved process.The distribution of system functions across multiple hosts means programmers must contend with a plethora of issues related to logical concurrency and low-level networking details. In most real-world systems integration contexts, systems developers are also challenged by interoperability, *i.e.,* how to ensure that components produced by different vendors can understand each other and communicate effectively. Additionally, for systems with real-time requirements, programmers and systems engineers must also ensure that the overall system exhibits the correct timing behavior.

Over the past two decades there has been significant effort by both the academic community and industry to address many of the distributed real-time systems development challenges via middleware [15, 4, 3]. The goal is to have the middleware provide a simplified programming abstraction that hides important, yet tedious and complex, implementation details such as data transport, service discovery/routing, serialization/deserialization and many others.

Recently, there has been increasing interest in middleware implementing the Data Distribution Service (DDS) Object Management Group (OMG) standard [11]. There are several high-quality implementations of the standard. Furthermore, a subset of these implementations have been rigorously verified and include certification packagages enabling their use in systems that require DO-178C Level A, IEC 60601, or ISO 26262 certification.

The quality of DDS middleware has lead to many large scale distributed real-time software systems that depend on it such as the Littoral Combat Ship [3], The Grand Coulee Dam [5], and Duke Energy's next generation smart grid [16]. The increasing popularity of DDS cannot be overstated: One major DDS vendor now claims that the total value of system designs that depends on their implementation of DDS exceeds "1 trillion dollars" [7].

DDS provides a "data-centric" publish-subscribe programming abstraction designed for use in real-time systems. Data producers (*i.e.,* publishers) write updates to topics and data consumers (subscribers) specify which topics they want to monitor and receive updates from. DDS lets the programmer attach Quality of Service (QoS) settings to publishers, subscribers, and topics. These settings define how the DDS middleware responds to faults (*e.g.,* connectivity loss) and lets the programmer specify the distributed timing requirements of the system (*e.g.,* the `DEADLINE` parameter, which specifies how often a subscriber should receive updates from

a publisher).

While the DDS standard provides a rich set of real-time centric QoS settings, DDS middleware does not fully lift system timing into the programming abstraction. Indeed, bringing timing fully into the programming abstraction is difficult because the timing behavior of a software system only emerges when you map the software onto a particular platform (*i.e.,* processor, operating system, and networking fabric). In the context of a critical distributed hard real-time system, special real-time networking hardware is often used and the systems integrator must devote significant effort provisioning and configuring that hardware to ensure the required timing behavior. The configuration process usually involves setting priorities or transmission schedules for each time critical flow. Sometimes, if the underlying network will be shared by more than one logically independent application, traffic policers and/or special partitioning features are setup to keep the different applications isolated. These network configurations are usually static and setup offline: If one wants to add a new node to the network (*e.g.,* a sensor or actuator) the system must be brought down, manually reconfigured, and then brought back up. The process of network resource configuration can be complex, costly, and error prone.

Recent advances in Software Defined Networking (SDN) have the potential to enable the enforcement of a distributed system's middleware-specified timing properties automatically and transparently in the network hardware. In theory, a programmer could specify distributed timing constraints, such as end-to-end latency requirements, to the middleware itself. Then the middleware could then leverage SDN protocols such as OpenFlow [10] to automatically and dynamically reconfigure the underlying network to enforce the specified timing constraints.

Using SDN to lift timing behavior into the programming abstraction offers a number of potential benefits. First, it means systems integrators would no longer need to manually provision network resources offline. Second, it allows for the use of general-purpose SDN hardware in real-time applications. Third, it allows for online dynamic network reconfiguration thereby enabling use-cases with real-time requirements but whose complete configuration is not known a priori (*e.g.,* the Industrial Internet of Things (IIoT) [1, 14] and plug & play medical systems [12]).

The aims of this paper are two-fold. First, we propose an SDN controller that automatically enforces requested QoS directly in the network and prevents network overloads from impacting the DDS traffic. A key design feature of our controller is that it does not require any modification to existing DDS implementations and in theory could be deployed in existing systems. This feature is important because it means certified DDS implementations would not need to go through expensive recertification to take advantage of our system. Second, we want to evaluate how a real DDS implementation is able to compensate for less than ideal network conditions such as overloads. Because an increasing amount of critical infrastructure is depending on DDS for correct function, it is important to understand how the middleware behaves under these conditions. As far as we know, up until this point, no such study has been performed or publically released.

This paper is organized as follows: In Section 2 we provide a overview of DDS middleware. Section 3 gives a brief background of OpenFlow. Section 4 describes new proposed QoS parameters and gives an overview of the architecture and function of our SDN controller. Section 5 contains the experimental evaluation. In Section 6 we survey and discuss related work. We conclude and propose areas for future work in Section 7.

# 2. DDS BACKGROUND

DDS provides a *data-centric* publish-subscribe programming abstraction [11]. The DDS abstraction exposes a number of features programmers can use to disseminate data across a distributed system at different levels of granularity. DDS also provides a rich set of QoS settings that can be used to both control and specify the non-functional properties of the system. A complete description of the DDS programming abstraction is beyond the scope of this paper. Instead, in this section we give a functional overview of the DDS programming abstraction. We describe, at a high-level, the different abstraction entities, their roles, and how they are typically implemented. We give special attention to the different QoS settings that can either affect or specify desired system timing.

## 2.1 Data Centric Programming Abstraction

Figure 2 illustrates the basic DDS programming abstraction. Any software that wants to produce and share data uses a *DataWriter* object. The *DataWriter* is associated with a *Topic*. If another program (which could be on a different host) wants that data, it uses a *DataReader* that is *subscribed* to that topic. We say that DDS is data-centric because *Topics* have structured data types and the data typing used in DDS lets us view *Topics* in a way that is intuitively similar to a table in a relational DBMS: *DataWriter* objects are a portal programmers can use to update the data stored in the table, and *DataReader* objects provide a view of the data in the table. Indeed, the DDS specification lets programmers use SQL-like queries with *DataReaders* to extract the specific data they desire.

Figure 1 shows how the top-level programming abstraction relates to intermediate abstractions and the underlying implementation. At the top level are the *DataWriters*, *DataReaders*, and *Topics* which embody the datacentric portion of the abstraction. In DDS, *DataWriters* are bound to Publishers and *DataReaders* are bound to Subscribers. The publishers and subscribers sit on top of modules that implement the *Real-Time Publish Subscribe Protocol* or RTPS. The RTPS module is split into two sub-modules, a platform independent module and a platform specific module. The Platform Independant Module (PIM) defines the types of messages and state-machines that are used by the RTPS. The Platform Specific Module (PSM) maps the logical protocol elements onto a specific networking implementation (*e.g.,* IP/UDP, CAN Bus, etc).

### 2.1.1 Quality of Service Parameters

DDS supports over two dozen QoS policies that can be specified for the different entities. The QoS policies affect how the middleware disseminates data from publishers to subscribers. Furthermore, the middleware automatically checks the consistency of the QoS settings between entities involved in a publish/subscribe relationship and notifies the application of any violation. The QoS policies supported by DDS affect various non-functional properties of the publish-

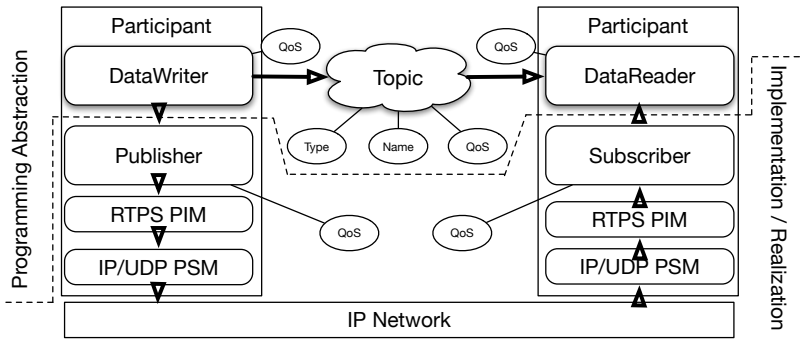



Figure 1: DDS programming abstraction entities and implementation.

subscribr system including data availability, delivery and timing. We will not cover every aspect of DDS QoS here. Instead we will only focus on the parameters that typically affect timing (either directly or indirectly).

- `DEADLINE`**(number)** - Defines the maximum separation between consecutive updates to a topic. *DataWriters* must not wait longer than the `DEADLINE` between consecutive writes. *DataReaders* will generate an exception if the deadline is exceeded between updates (*i.e.,* if a *DataWriter* is too slow or the underlying network delays delivery of an update).

- `LATENCY_BUDGET`**(number)** - Hint to the middleware specifying tolerable update delivery delay. The latency budget is used by the underlying middleware to decide whether it can batch updates together in order to optimize network bandwidth.

- `BEST_EFFORT`**(boolean)** - If true, the middleware won't attempt to resend lost updates. Publishers will simply transmit topic updates as fast as possible onto the network.

- `RELIABLE`**(boolean)** - Specifies whether a reliable transport must be used to deliver updates. Messages lost due to network contention or faults will eventually be recovered by the subscriber.

- `HISTORY_DEPTH`**(number)** - Defines how large the update cache must be in a *DataReader* or *DataWriter*. For *DataWriters* the history depth indicates how many updates it will keep available for recovery in `RELIABLE` sessions. For *DataReaders* it is simply the number of previous updates the DataReader can make available to the application.

- `DESTINATION_ORDER` - Specifies whether the subscriber *DataReader* must present topic updates to the application in the order they were written to the *DataReader*.

- **User QoS** - DDS also allows users to specify their own QoS parameters that can be processed by their application. User QoS lets users attach arbitrary key/value pairs to *DataReader/DataWriters*. These key/values are exchanged during discovery but will not affect the behavior of the middleware. Instead, they can be recovered and used by the application for whatever purpose as intended by the programmer.

## 2.2 Operation

The operation of DDS can be divided into three main phases: 1) *Discovery* - where the different DDS participants find each other on the network, 2) *Matching* - where the discovered participants determine if they should engage in a publish-subscribe relatonship, and 3) *Data Distribution* - where data is disseminated from the publishers to subscribers that have matched. The particulars of how DDS operates depends on the type of networking technology being used and the Platform Specific Module (PSM) implemention of the RTPS for that network. Here we give a high-level overview of how these different phases are implemented in the IP/UDP PSM that is used when DDS is operated on typical IP networks.

### *2.2.1 Discovery*

The discovery phase is itself divided into two sub-phases: Participant discovery and endpoint discovery. The aim of participant discovery is to discover each node on the network that is running DDS. Once the DDS participants have been discovered, each participant will attempt to discover the end-points (*i.e., DataWriter*s & *DataReader*s) hosted on the other participants and what topics those end-points are publishing or subscribing to.

The IP/UDP PSM implements participant discovery using the Simple Participant Discovery Protocol (SPDP). The SPDP works by IP multicasting: Each participant periodically transmits a discovery message to a pre-defined multicast address. The discovery message contains the information needed by other participants to uniquely identify the originator of the message. Participants also listen for multicasts to that same address. When a participant detects another participant, the information contained in the discovery message is stored in a local "participant database".

After a participant discovers another participant it will

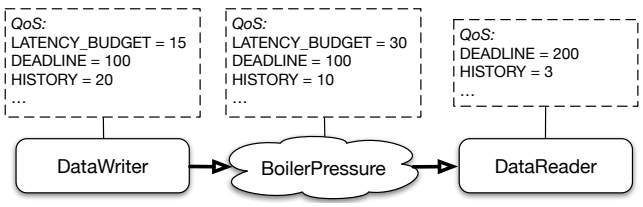



Figure 2: Publish-subscribe relationship with consistent QoS settings.

initiate end-point discovery using the Simple End-point Discovery Protocol (SEDP). SEDP is a reliable (*i.e.,* it will resend lost messages) point-to-point protocol that allows one participant to learn what the other participant's *DataReaders*/*DataWriters* are, the topics/data-types associated with those *DataReaders*/*DataWriters*, and their respective QoS settings. Additionally, participants learn a "locator list" matching topics to IP addresses. Typically, each topic will be associated with its own multicast address / IP port combination to facilitate efficient dissemintation of topic updates over the network. The participants will store the information learned about remote end-points in a local end-point cache.

### 2.2.2 Matching

After the end-points have been discovered the matching process will begin. Subscribers will examine their local end-point cache to see if they have discovered any end-points that are publishing to the topics they are subscribing to. If the subcriber find an end-point that is publishing to a topic they subscribe to, the subscriber will compare its own QoS requirements to the QoS offered by the publishers. If the QoS settings are consistent (*i.e.,* the publisher's QoS guarantee is at least as strong as the subscriber's requirements) then the match is successful and the subscriber will start receiving topic updates on behalf of its *DataReaders*. Since multicast locators are typically used, the subscriber will start listening on the appropriate address for updates.

### 2.2.3 Data Distribution

At a high-level, IP/UDP data distribution is fairly simple and works by having the publishers serialize the update data, encapsulate that data with DDS specific information (*e.g.,* sequence number) and then transmit the encapsulated data as a UDP packet to the multicast address associated with the topic in question. The subscribers subscribed to the topic will be listening for packets on that multicast address. When a subscriber receives a new packet they will deserialize the update and present it to the application via the associated *DataReader*. While basic operation is fairly simple, the details of how the RTPS and IP/UDP PSM handles data distribution changes based on the QoS settings used.

- `BEST_EFFORT` = *true* - If `BEST_EFFORT` QoS is active then the publisher will "fire and forget" topic updates. Updates lost in transit will not be recovered.
- `RELIABLE` = *true* - The IP/UDP PSM will use a reliable multicast procotol to recover lost update messages. Subscribers will keep track of each received update's sequence number. NACK messages are sent to the publisher for each sequence number missed and the publisher will retransmit the updates (that is has cached) for the missing sequence numbers.
- `HISTORY_DEPTH` = $n$. When reliable transport is enabled, the publisher can only resend lost updates that are present in its local cache. If the publisher no longer has the update specified by a NACK it will notify the subscriber that the data is no longer available.
- `DESTINATION_ORDER` = *true*. If a *DataReader* is set to enforce ordering it will not present the most recent update to the application until all previous updates have been presented. If intermediate updates have been lost due to network faults or contention, the *DataReader* will wait to present recent updates until either the missing updates have been recovered and presented, or the publisher has indicated it no longer has the missing updates in its update cache.

## 3. OPENFLOW BACKGROUND

OpenFlow [10] is a protocol designed to enable Software Defined Networking (SDN). SDN separates the *data plane* (where data packets are switched, routed and otherwise moved) and the *control plane*, which decides how to configure the data plane based on high-level routing or switching policies. Traditional networking equipment contains both the control and data planes. For example, a typical COTS ethernet switch implements the data-plane in hardware using a specialized packet-switching ASIC while the firmware running on the switch's general purpose CPU will configure the switching ASIC according to whatever policy the switch manufacturer has defined.

SDN protocols like OpenFlow enable the reconfiguration of the switch hardware ASIC from a server process running on a remote machine (called the OpenFlow controller). This capability makes it posible to write "controller applications" using a language like Java or C++ that learn the topology of the network (*i.e.,* learn what ports on what switches different hosts are connected to) and then effect complex routing, forwarding and QoS strategies.

We now describe the operation of an OpenFlow network. An OpenFlow network consists of OpenFlow switches and OpenFlow controllers (*e.g.,* Floodlight, Ryu, *etc.*). An OpenFlow hardware switch is a Layer 2/3 Ethernet switch that maintains a table of *flow* entries and actions.

The flow table associates each flow with a set of *actions* that indicate to the switch how to handle packets matching the flow. The OpenFlow standard defines a number of different possible actions a switch may implement including *enqueue* (place the packet on a specific egress queue), *limit* (apply a rate-limiter) and *drop* (simply drop the packet).

When a switch receives a packet on one of its interfaces, it tries to find a matching flow entry in its flow-table. If a matching entry is found, the associated action set is applied. If the packet does not match an existing entry the switch performs a *Packet-in*. When the switch performs a *Packet-in*, it sends a message to the OpenFlow controller (a piece software running on a server in the network). The message contains the packet that failed to match as well as meta-information about the packet such as on which switch the match failure occurred, the physical port of the switch that the packet arrived on, and other statistics. The controller can use the meta-information and the information in the failed packet itself to decide on a course of action. The controller can then either modify one or more switch's flow tables with a new rules and/or send the packet back to a switch with instructions on how to deal with just that packet (called *Packet-out*).

## 4. OVERVIEW

Our controller works by using the OpenFlow protocol to transparently insert itself into the DDS discovery process. It uses the OpenFlow protocol to dynamically learn where DDS nodes exist on the network and what each nodes specified QoS is. If two nodes want to engage in a publish-

subscribe relationship, the controller extracts the specified end-to-end QoS by intercepting and deserializing the discovery messages. The controller will then attempt to derive a network configuration and resource reservation that satisfies the specified QoS. If the new configuration will guarantee the QoS the controller will let the discovery process proceed and then commit the new configuration to the network so DDS data traffic can flow from publisher to subscriber.

## 4.1 Proposed QoS Additions

While our system does not require any modifications to the DDS middleware itself, it does need information that can't be easily derived from the existing standardized QoS parameters. Here we propose two new QoS parameters that will enable us to support automatic timing guarantees. While our current system requires that the application developer supply these parameters to the middleware via the User QoS feature, we hope that these (or equivalent) parameters will eventually be considered for inclusion in the DDS standard.

The first QoS we propose is called `MINIMUM_SEPARATION`. `MINIMUM_SEPARATION` applies to *Publisher*s and defines the smallest allowed interval between the network transmissions related to two consecutive updates to a topic. In this paper we use $minsep_p^t$ to denote the `MINIMUM_SEPARATION` applied to publisher $p$ for topic $t$. Formally, we define $minsep_p^t$ as follows:

*Definition 1.* **Publisher minimum separation.** Let $t_i$ and $t_{i+1}$ be any two moments when the publisher $p$ starts transmitting the data for topic $t$ update $i$ and $i+1$ respectively. Then it is always the case that $minsep_p^t \leq t_{i+1} - t_i$.

`MINIMUM_SEPARATION` is essential for our ability to do schedulability analysis: It precisely defines how often a publisher can transmit a data on the network. When combined with the maximum data-size for an update to a given data-type, `MINIMUM_SEPARATION` lets us derive how much load the publisher can induce on the network which is required for schedulability analysis and the ability to guarantee QoS. The second proposed Qos parameter concerns end-to-end latency. While the DDS standard already has the `LATENCY_BUDGET` QoS, its semantics is purposely not well defined. Furthermore, `LATENCY_BUDGET` is intended to apply to publishers as a hint how to batch topic updates for network transmission. We propose a new latency QoS in order to prevent confusion with DDS' `LATENCY_BUDGET`. Our new proposed parameter for end-to-end latency, called `E2E_LATENCY`, defines an upper bound on the amount of time it can take an update to traverse the network from publisher to subscriber. Unlike DDS' `LATENCY_BUDGET`, `E2E_LATENCY` is intended to be specified by *Subscribers*. Furthermore, the semantics `E2E_LATENCY` require the underlying network infrastructure to guarantee that the specified latency *always* be satisfied. In this paper we use $l_s^t$ to denote the `E2E_LATENCY` specified by subscriber $s$ for topic $t$:

*Definition 2.* **End-to-end latency.** Let $t_i^p$ be the moment publisher $p$ transmits the first bit of update $i$ on the network and $t_i^s$ be the moment subscriber $s$ receives the last bit of update $i$. Then for any $i$, it is always the case that $t_i^s - t_i^p \leq l_s^t$.

## 4.2 Operation & Architecture

Figure 3a shows the functional software architecture of our SDN controller and Figure 3b shows how DDS discovery & data traffic flows in our solution. The controller is implemented in Java as a "controller" application running on top of the Floodlight controller. Floodlight offers basic functionality including:

- *TopologyService* - Maintains the topology of the network. The topology service periodically causes the switches under the control of the controller to send Link Layer Discovery Packets (LLDP) out of their ports. Changes in the physical topology of the network (*e.g.,* due to cable connect/disconnect) will be detected by the topology service, which will then notify the other Floodlight modules and controller applications.

- *Forwarding/Routing* - These two modules will learn the MAC address of all network connected devices and will push flow rules to ensure that non-DDS traffic can transit the network as if it was a normal (non-SDN) ethernet. We've modified these modules from the ones available in the normal Floodlight distribution to ensure that non-DDS traffic will always be queued at a lower priority than the DDS traffic.

- *DeviceManager* - Maintains database of the hosts connected to the network including their MAC/IP addresses and what port of what switch the host is plugged into.

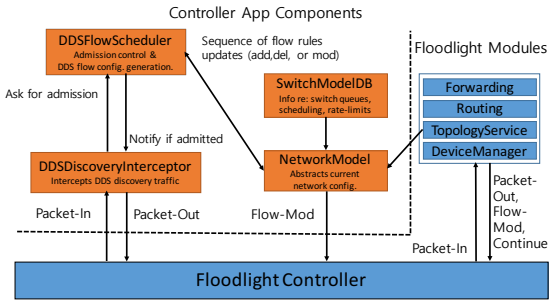
In addition to the above, our controller runs four more modules specifically designed guarantee the QoS of DDS traffic:

- *DDSDiscoveryInterceptor* - Handles *Packet-In* events for all DDS discovery (*i.e.,* SPDP or SEDP) traffic. This module eavesdrops on the discovery procotol and learns what network devices are DDS participants, whether two participants have end-points that want to engage in a publish-subscribe relationship, and what the requested QoS for that relationship is.

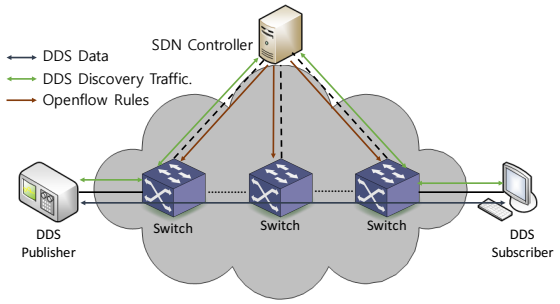
- *DDSFlowScheduler* - Does admission control for publish-subscribe relationships. Generates a network configuration (*i.e.,* flow rules) that potentially guarantees the QoS of all publish-susbcribe flows. Uses schedulability analysis to determine if the new network configuration is feasible. If so uses the *NetworkModel* to safely update the flow rules in the switches.

- *NetworkModel* - Stores dynamic information about the state of the network and resource reservations for DDS clients. It tracks which nodes are DDS participants, which participants are engaged in a publish-subscribe relationship, the QoS of those relationships, and the flow rules installed into each switch to support those relationships.

- *SwitchModelDB* - Stores static information about the performance characteristics of each switch in the network. This information is used by the flow scheduler to generate new network configurations and to perform feasibility testing of those configurations. The information includes, for each model of switch used in the network, multiplexing latency, the number of




(a) Software architecture of the SDN controller.




(b) Routing of DDS data, DDS discovery, and OpenFlow flow-modification messages.

Figure 3: SDN controller design & operation.

egress queues per port (and the depth of the queues), the intra- and inter-queue packet scheduling discipline used (*e.g.,* FIFO, WFQ, FP, *etc.*), and the precision of the ingress rate-limiters.

Using these modules the controller application works as follows. When DDS clients start participant discovery via the SPDP they will multicast discovery packets. We ensure that the flowtable of each switch has no entry that can match against SPDP traffic. This causes a switch that receives the SPDP traffic to forward it to the controller via *Packet-In*. On the controller side, *Packet-In*'s related to SPDP or SEDP traffic is forwarded to the *DDSInterceptor* module. The *DDSInterceptor* deserializes the SDPD messages and learns the IP address and network location (*i.e.,* the switch and port) of the DDS participant. This information is then cached in the *NetworkModel*. The interceptor then *Packet-out*s the SPDP packet to the other DDS participants.

Next, once the participants have discovered each other, they will initiate end-point discovery with the SEDP. Again, since the SEDP traffic will not match any rule in any switch's flow table the SEDP traffic will be forwarded to the controller via *Packet-In*. The controller uses the SEDP traffic to learn which DDS participants want to engage in a publish-subscribe relationship and the requested QoS settings for that relationship. If both `MINIMUM_SEPARATION` (publisher) and `E2E_LATENCY` (subscriber) has been specified then the interceptor will initiate admission control with the *FlowScheduler*.

The *FlowScheduler* will attempt to derive a network configuration that guarantees the requested QoS. The mathematical details of how such a scheduler could work are beyond the scope of this paper (See [8] or [9] for two different approaches. We follow the approach of [8]). Instead we will give a high-level description. First the *FlowScheduler* uses the publisher's `MINIMUM_SEPARATION` and the topic's data type to infer a worst-case arrival pattern of bits that the flow of updates from the publisher can induce onto the network. Then, the *FlowScheduler* uses the information stored in the *NetworkModel* to find a path for the flow across the network that satisfies the latency requirement. The *FlowScheduler* may choose to prioritize the new flow differently at each hop to ensure that each real-time DDS flow (both new and old) will always satisfy its QoS. After it has generated a new configuration, the *FlowScheduler* will test the configuration for schedulability.

If any flow admitted to the system can miss its specified QoS the admission process for the new flow fails and the *FlowScheduler* will instruct the *DDSInterceptor* to block the matching process between the publisher and subscriber in question. If the new configuration guarantees the flow's QoS the flow scheduler will give an ordered list of "flow mod" commands to the *NetworkModel*. The *NetworkModel* will cache the new flow configuration and then commit it to the network in the order specified (The ordering generated by the *FlowScheduler* ensures that QoS of existing flows will not be violated during the reconfiguration process). After the new configuration has been comitted to the network the *FlowScheduler* will instruct the *DDSInterceptor* to let the SEDP proceed so the publisher and subcriber can match. Since flow rules matching the new publish-subscribe flow are now present in the switches, all publish subscribe traffic between the new end-points can flow freely and will be processed at line rate.

# 5. EXPERIMENTAL EVALUATION

Our experiments have two goals. First, we want to understand how a real DDS implementation compensates for network overloads. In particular, we want to evaluate how network overload conditions affect the end-to-end timing of the publish-subscribe relationships and then how well the DDS reliability and resilience features (*e.g.,* `RELIABLE`, `HISTORY_DEPTH`, and `DESITINATION_ORDER` QoS) help the application recover from message losses incurred during the overload. The second goal is to evaluate how well our SDN controller is able to protect DDS traffic from interference from network overloads. We will compare the timing performance of the system when our controller is enabled to when it is not and when the network is and isn't overloaded.

## 5.1 Setup

Figure 4 shows the experimental setup. We built a network comprised of a single Pica8 P-3297 Gigabit top of rack switch, three Linux computers (one for the publisher, one for the subscriber, and one to run the SDN controller) and a Linux desktop to act as a traffic generator. In order to ensure that we can overload the queuing capacity of the switch port connected to the subscriber host the traffic generator is connected to the switch with two 1Gbps links.

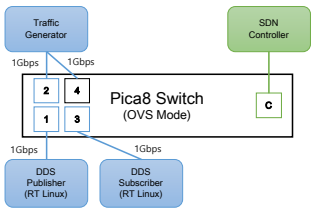



Figure 4: Experimental setup.

The publisher/subscriber computers were desktop machines with Intel Core i7-3770 (publisher) or i7-4770 (subscriber) processors, 8GB (publisher) or 16GB (subscriber) of RAM and configured with the fully preemptible (*i.e.,* `RT_PREEMPT`) real-time Linux kernel version 4.4.4-rt11.

We wrote a custom application in real-time (RTSJ) Java (running on IBM's WebSphereRT real-time JVM) that publishes an update to topic *TestTopic* every $5ms$. Likewise, we wrote a subscriber application that uses DDS to subscribe to *TestTopic*. We chose Real-Time Java because it offers a good combination of programmer productivity, predictability, and lets us avoid certain annoying issues that can manifest when unmanaged programming languages are used (*e.g.,* heap fragmentation). Both the subscriber and publisher have `DEADLINE` QoS set to $5ms$. The publisher's `MIN_SEPARATION` is also set to set to $5ms$ and the `E2E_LATENCY` of the subscriber is set to $1ms$. The DDS implementation used is Real-Time Innovation Inc.'s Connext DDS [7]

Each time the subscriber got an update it would record when it got the update, the sequence number of the update, and a count of the cummulative updates. The combination of real-time Java, IBM's WebSphereRT, and real-time Linux afforded us fairly precise control over when the publisher actually published an update to a topic, and when the subscriber activated in response to receiving an update: Observed dispatch jitters (*i.e.,* variation in when scheduled events, such as topic writes, happen in the Java application) were always $\leq 100\mu s$. Latency through the network card, kernel networking stack, JVM and middleware were always less than $1ms$ total.

The traffic generator host was equipped pktgen packet generator program. When activated, it generated a flood of UDP packets towards a the subscriber's IP address in a specific pattern: The generator would cycle flooding for 1 second, then stop for one second. The goal of this pattern is to help us see the effects of *transient* overloads. Using both network links the traffic generator could consistently flood the network at 2,000Mb/s, more than enough to overwhelm the capacity of the subscriber's 1000Mb/s connection to the switch. We ran our test publish-subcribe application a number of times with a variety of QoS settings and with/without our SDN controller. For each run we ran our publish-subscribe system for 25 seconds. Over the course of these runs we captured two categories of data-sets. The first category is *cummulative updates received vs time*. The other is *Normalized Latency vs. Seq. Number*.

### 5.1.1 Cummulative Updates vs. Time

The cummulative updates vs. time datasets give the number of updates to *TestTopic* the subscriber's *DataReader* received and processed by time $t$. We ran 11 categories of this experiment, 6 with `BEST_EFFORT` QoS and 5 with `RELIABLE` QoS. In all categories `DESTINATION_ORDER` QoS is active:

- `BEST_EFFORT` without contention or SDN, $DataSize = 100$ bytes: This variations is to establish a baseline of "ideal" behavior. As `DEADLINE` $= 5ms$ we expect the subscriber to receive updates at a uniform rate of 200 per second because there is no network contention.
- `BEST_EFFORT` without SDN but with periodic contention, $DataSize = n$ bytes with $n \in \{100, 500, 1000, 1400\}$: Captures the performance degradation due to network contention.
- `BEST_EFFORT` with SDN, periodic contention and $DataSize = 1400$. Captures the ability of the SDN controller to effectively manage the contention and preserve the specified timing behavior of the publish subscribe system.
- `RELIABLE` without SDN but with periodic contention, `HISTORY_DEPTH` $= n$ bytes with $n \in \{1, 50, 100, \infty\}$: Captures the ability of the reliable transport to compensate for network packet drops.
- `RELIABLE` with contention and SDN, `HISTORY_DEPTH` $= 1$: This variations captures the performance of the system when the SDN control is enforcing QoS and the reliable multicast transport is being used.

Additionally, for each category subject to traffic loads from the traffic generator we ran 15 sub-variations where we varied the size of the generated packets from 100 - 1500 bytes in steps of 100 bytes while keeping the overall traffic rate constant (in effect varying the packets per second).

### 5.1.2 Normalized Latency vs. Seq. Number

The normalized latency vs. seq. number datasets give when a particular update was received *vs.* when it was expected:

*Definition 3.* Expected Update Time - Let $i$ denote an update. Let $t_0$ be the time the first update to the topic is received. Then the expected update time for update $i$, $e(i)$ is calculated with the following equation:

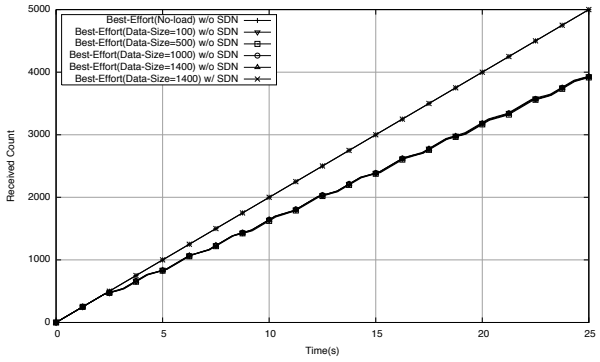
$$e(i) = (i \times 5ms) + t_0 \tag{1}$$

*Definition 4.* Normalized Latency - Let $t_i$ be the time update $i$ is actually received. Then the normalized latency of an update $i$, $nl(i)$ is the difference of the actual update time from the expected time:

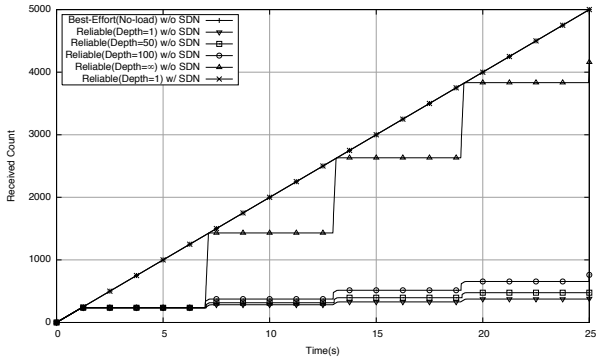
$$nl(i) = t_i - e(i) \tag{2}$$

Normalized latency helps us understand the effects of network contention on how the application perceives the global data space with respect to time. As with the cummulative updates vs. time data sets we capture data for a number of QoS variations including `BEST_EFFORT` & `RELIABLE` transports, different `HISTORY_DEPTH` and with/without SDN support.

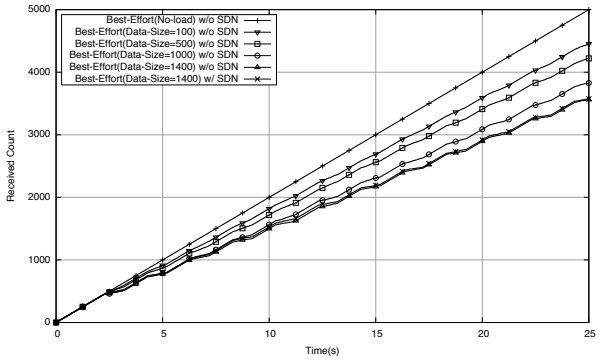
Observe that, even in the ideal case, we don't expect the normalized latency to be 0 due to processing and network jitters. Also, when SDN support is enabled, we expect small variations in the normalized latency because `E2E_LATENCY` $= 1ms$ (update $i$ could have an actual latency close to 0 but update $i + 1$ could have an actual latency of $1ms$).



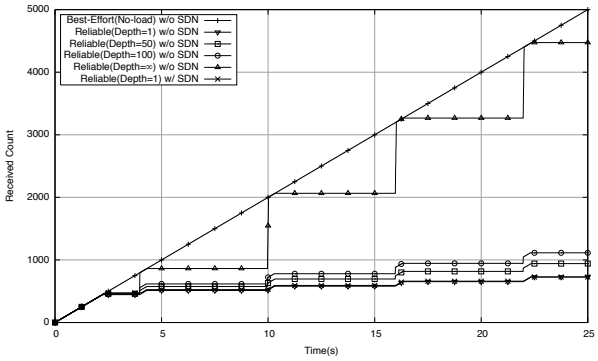

(a) `BEST_EFFORT` with 1400-byte background traffic packets.




(b) `RELIABLE` with 1400-byte background traffic packets.




(c) `BEST_EFFORT` with 100-byte background traffic packets.




(d) `RELIABLE` with 100-byte background traffic packets.

Figure 5: Cummulative updates vs. time.

## 5.2 Results & Discussion

Figure 5 shows the cummulative updates vs. time dataset. The results for background traffic packet sizes $\geq$ 200 is the same so we only show the results for packet sizes 1400 (Figures 5a & 5b) and 100 (Figure 5c & 5d). For both `BEST_EFFORT` (Figure 5a) and `RELIABLE` (Figure 5b) QoS the effects of periodic network contention are readily apparent when SDN support is disabled. However, the effects manifest differently depending on whether `BEST_EFFORT` or `RELIABLE` is used. When there is no contention the behavior of the system very closely matches the idealized expected behavior: the slope of the line indicates that the subscriber is uniformly receiving updates every $\sim 5ms$. When `BEST_EFFORT` QoS is used periodic networking contention causes both updates to be dropped in the network and the overall update rate to be depressed (causing `DEADLINE` QoS violations in the *DataReader*).

The periodic pattern of the contention is clearly visible in Figure 5a: For the lines representing the data sets without SDN support, their slope is depressed for the 1 second durations the traffic generator is active and the slope returns to the idealized rate once the traffic generator is off. The difference between the line "idealized" line (representing the run where there is no load) and the other lines is number of updates that were lost due to network overloads. As we can see, the the data-size of the update does not affect the impact of the network contention. When the SDN controller is managing the traffic flows the traffic generator causes no affect on the rate the subscriber receives updates. Indeed, the line for "Best-Effort(Data-Size=1400) w/ SDN" exactly tracks the line for when there is no contention.

A system running with `RELIABLE` QoS reacts differently to network contention (see Figure 5b). Instead of only experiencing an update rate slowdown when the contention is active (*i.e.,* for 1 second durations) like with `BEST_EFFORT` QoS, updates are prevented from being processed by the *DataReader* for over 8 seconds with `RELIABLE` QoS. Why is this? There are two factors at play (confirmed via code inspections and discussions with the middleware vendor). First, because `DESTINATION_ORDER` = *true* the middleware won't manifest updates to *DataReader* until all previous updates have been manifested: If one update is dropped, all subsequent updates that are received are held back from the application by the middleware until the subscriber receives the missing update in effect. Second, since the publisher must retransmit the missed updates, network contention is actually increased, making it more likely for further updates to drop.

Though `RELIABLE` QoS can exaggerate the duration that updates are missed, if the publisher's `HISTORY_DEPTH` is large enough the middleware is able to exploit network idle times to recover the missing data. We see this behavior with the line for "Reliable(Depth=$\infty$) w/o SDN": No updates are processed by the *DataReader* once the traffic generator activates at $t = 1$ second until $t = 8$ seconds, at which point all the missing updates have been recovered and the received update count jumps to where it should be if no updates had been lost. Again, like with `BEST_EFFORT` QoS, the SDN controller manages the traffic flows to ensure idealized behavior when `RELIABLE` QoS is used.

However, for both RELIABLE and BEST_EFFORT QoS with 100 byte background traffic packet sizes (Figures 5c & 5d) our SDN controller is apparently not able to enforce the specified QoS. After an investigation we discovered that it was not the network dropping packets but the subscriber host itself: The host's NIC and kernel could not keep up with the packet rate induced by the small packets as confirmed by the rx_no_buffer_count and rx_missed_errors kernel driver statistics.

Figure 6 shows the normalized latency of each update for runs of the system with a variety of different QoS settings. Figure 6a shows the "idealized" behavior: BEST_EFFORT QoS with no interference from the traffic generator. The normalized latency in the idealized case remains close to 0 for the entire run. The normalized latency is actually usually slightly less than 0. How can this be? Recall that the expected arrival time is computed relative to the time the first update is received. The publisher starts out sending the first few updates with a frequency slightly greater then the specified MIN_SPEARATION (verified by inspecting the sending timestamps). It is unclear why this behavior occurs but we hypothesize speed increases are due to optimizations resulting from JIT compilation of the Java application or cache effects in the CPU.

Figures 6b, 6c, 6d, 6e, and 6f all show the normalized latency under a variety of QoS configurations while the system is subjected to periodic network contention and no SDN support. In each case, the periodic interference pattern is visible in the normalized latency. For BEST_EFFORT QoS (Figure 6 (b)) updates are delayed or dropped while the traffic generator is active. Interestingly, the normalized latency of the *received* updates plateaus. Why is this? We believe that the plateau results from the size of the egress queue on the switch: Updates that would take longer due to queue contention are dropped because the queue is already at capacity.

For RELIABLE QoS (Figures 6c, 6d, 6e, and 6f) we again see how the network contention, reliable multicast, and DESTINATION_ORDER causes a cascade of delays due to dropped updates. This effect is most striking when the HISTORY_DEPTH $= \infty$ (Figure 6f): The network contention causes an update to be missed early in each of the traffic generator's active periods which causes all subsequent messages to be delayed. However, once the dropped update is recovered subsequent updates flow steadily.

When our SDN controller is used to manage the traffic flows, the normalized latency for both BEST_EFFORT and RELIABLE runs closely matches the idealized run where there is no background traffic.

# 6. RELATED WORK

The most closely related work is by King *et al.* [8]. King *et al.* proposed a custom publish-subscribe middleware specifically designed to work with OpenFlow networks to enforce end-to-end QoS. While [8] provides the results of a preliminary evaluation demonstrating the potential of using SDN to enforce QoS in publish-subscribe networks no comparison was made to an "industrial strength" middleware with advanced reliability and resilience features like DDS: The custom middleware used in [8] only supports unreliable UDP multicasting to transport data from publisher to subcriber. While this limitation does not cast doubt on the evaluation presented in [8] it does mean that King *et al.*'s results might not have applied to more realistic middleware implementations. The experimental evaluation in this paper should be considered confirmation, using industrial strength middleware, of ideas originally proposed in [8].

Also related is the work by Bertaux *et al.* in [2]. In [2], Bertaux *et al.* describes an OpenFlow SDN architecture intended to capture DDS QoS and optimize the network at Layers 2 & 3 to achieve the specified QoS. Unfortunately, [2] does not provide many details of their system nor do they provide any sort of experimental evaluation.

Recently there have been a number of efforts to augment DDS with the ability to optimize its QoS parameters on the fly [6]. These approaches work by adding a "QoS monitor" to the middleware. The QoS monitor reacts to changes in the actual runtime performance of the system and attempts to globally optimize performance by tweaking the local performance of each DDS participant (*e.g.,* causing the publishers to slow down or speed up how often they transmit updates).

[13] performed a number of different performance evaluations of DDS middleware implementations. Unlike our evaluation the experiments in [13] do not specifically how well the middleware performs under faulty or overload conditions. Instead, the goal was to evaluated the average case behavior of the implementation and to test its scalability (*i.e.,* the number of DDS participants that could be effectively managed).

# 7. CONCLUSION & FUTURE WORK

In this paper we evaluated how a real DDS middleware implementation performs under network overload conditions. Because an increasing amount of critical infrastructure is depending on DDS for correct function, it is important to understand how the middleware compensates for less than ideal conditions. As far as we know, this is the first evaluation of its kind for DDS middleware.

We also presented solution that uses software defined networking to fully "lift" timing properties into the DDS programming abstraction: Our SDN controller ensures that the requested QoS is guaranteed. Our solution has a number of advantages, including the fact that it can work with commodity off the shelf networking hardware and does not require any modification to the DDS middleware implementation. Our experiments indicate that our SDN controller is able to effectively mitigate the overload conditions that would normally negatively impact the performance of a DDS publish-subscribe system.

Our results are promising but preliminary. For example, our experiments only involved a network with one switch and one real-time flow. Real publish-subscribe applications can span many networking switching elements and involve many thousands of publish-susbcribe flows, each with their own QoS requirements. One very important direction for future work is larger scale experimental evaluations, both in terms of the number of network switching elements and the number of flows with real-time requirements.

Lastly, while this paper focused on using SDN to enforce the timing properties of publish-subscribe systems one can imagine that SDN could also be used to enforce other non-functional properties such as security. For example, many security-sensitive systems require that classified data only flow between classified nodes. SDN could automatically enforce dataflow properties in the network itself by only generating flow rules that satisfy the security requirements. Investigating how publish-subscribe middleware and SDN could

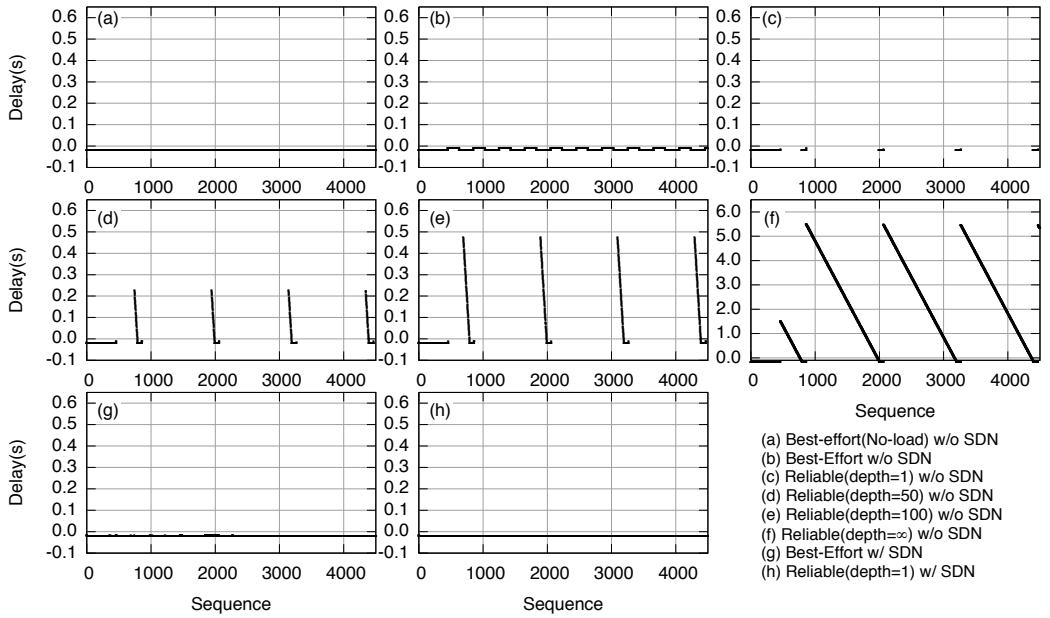



Figure 6: Normalized update latency vs. sequence number. 1400 byte background traffic packets.

be combined to create a robust security architecture could be interesting future work.

# 8. REFERENCES


[1] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[2] L. Bertaux, A. Hakiri, S. Medjiah, P. Berthou, and S. Abdellatif. A dds/sdn based communication system for efficient support of dynamic distributed real-time applications. In *Distributed Simulation and Real Time Applications (DS-RT), 2014 IEEE/ACM 18th International Symposium on*, pages 77–84. IEEE, 2014.

[3] DoD. The Data Distribution Service: Reducing Costs Through Agile Integration. Technical report, U.S. Department of Defense, 2011.

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[5] M. Gill. Rti data distribution service selected for upgrade of grand coulee dam‘s control system. *Rueters*, 2009.

[6] J. Hoffert, A. Gokhale, and D. C. Schmidt. Timely autonomic adaptation of publish/subscribe middleware in dynamic environments. *Innovations and Approaches for Resilient and Adaptive Systems*, page 172, 2012.

[7] R.-T. Innovations. Rti connext dds professional, 2014.

[8] A. L. King, S. Chen, and I. Lee. The middleware assurance substrate: Enabling strong real-time guarantees in open systems with openflow. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, pages 133–140. IEEE, 2014.

[9] J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media, 2001.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[11] 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.

[12] J. Plourde, D. Arney, and J. M. Goldman. Openice: An open, interoperable platform for medical cyber-physical systems. In *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*, pages 221–221. IEEE, 2014.

[13] S. Sierla, J. Peltola, and K. Koskinen. Evaluation of a real-time distribution service.

[14] J. A. Stankovic. Research directions for the internet of things. *Internet of Things Journal, IEEE*, 1(1):3–9, 2014.

[15] S. Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[16] A. Vukojevic, S. Laval, and J. Handley. An integrated utility microgrid test site ecosystem optimized by an open interoperable distributed intelligence platform. In *Innovative Smart Grid Technologies Conference (ISGT), 2015 IEEE Power & Energy Society*, pages 1–5. IEEE, 2015.