6-2014

The MIDdleware Assurance Substrate: Enabling Strong Real-Time Guarantees in Open Systems With OpenFlow

Andrew L. King
University of Pennsylvania, kingand@cis.upenn.edu

Sanjian Chen
University of Pennsylvania, schen@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Recommended Citation

Andrew L. King, Sanjian Chen, and Insup Lee, "The MIDdleware Assurance Substrate: Enabling Strong Real-Time Guarantees in Open Systems With OpenFlow", *17th IEEE Computer Society Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)* , 133-140. June 2014. http://dx.doi.org/10.1109/ISORC.2014.49

17th IEEE Computer Society symposium on object/component/service-oriented realtime distributed computing (ISORC 2014), Reno, Nevada, USA. June 8-12, 2014

The MIDdleware Assurance Substrate: Enabling Strong Real-Time Guarantees in Open Systems With OpenFlow

Abstract

Middleware designed for use in Distributed Real-Time and Embedded (DRE) systems enable cost and development time reductions by providing simple communications abstractions and hiding operating system-level networking API details from developers. While current middleware technologies can hide many low-level details, designers must provide a static configuration for the system's underlying network in order to achieve required performance characteristics. This has not been a problem for many types of DRE systems where the configuration of the system is relatively fixed from the factory (e.g., aircraft or naval vessels). However for truly open systems (i.e., systems where end users can add or subtract components at runtime) the standard static network configuration approach cannot guarantee that required performance will be met because network resource demands are not fully known a priori. Open systems with stringent performance requirements need middleware that can dynamically manage the underlying network configuration automatically in response to changing demands. Fortunately, recent trends in networking have resulted in a wide variety of networking equipment that expose a standardized low-level interface to their configuration via the OpenFlow protocol. In this paper we discuss how OpenFlow can be leveraged by DRE middleware to automatically provide performance guarantees. In order to make the discussion concrete, we describe the architecture of our prototype middleware MIDAS as well as the details of one example network resource management strategy. We demonstrate the feasibility of our approach via performance assesment of a simple DRE application using our MIDAS and commerically available OpenFlow hardware.

Disciplines

Computer Engineering | Computer Sciences

Comments

17th IEEE Computer Society symposium on object/component/service-oriented realtime distributed computing (ISORC 2014), Reno, Nevada, USA. June 8-12, 2014

The MIDdleware Assurance Substrate: Enabling Strong Real-Time Guarantees in Open Systems with OpenFlow

Andrew L. King, Sanjian Chen, and Insup Lee
Computer and Information Science Dept.
University of Pennsylvania
Philadelphia, PA
Email: {kingand, schen, lee}@cis.upenn.edu

***Abstract*—Middleware designed for use in Distributed Real-Time and Embedded (DRE) systems enable cost and development time reductions by providing simple communications abstractions and hiding operating system-level networking API details from developers. While current middleware technologies can hide many low-level details, designers must provide a static configuration for the system's underlying network in order to achieve required performance characteristics. This has not been a problem for many types of DRE systems where the configuration of the system is relatively fixed from the factory (e.g., aircraft or naval vessels). However for truly open systems (i.e., systems where end users can add or substract components at runtime) the standard static network configuration approach cannot guarantee that required performance will be met because network resource demands are not fully known a priori. Open systems with stringent performance requirements need middleware that can dynamically manage the underlying network configuration automatically in response to changing demands. Fortunately, recent trends in networking have resulted in a wide variety of networking equipment that expose a standardized low-level interface to their configuration via the OpenFlow protocol. In this paper we discuss how OpenFlow can be leveraged by DRE middleware to automatically provide performance guarantees. In order to make the discussion concrete, we describe the architecture of our prototype middleware MIDAS as well as the details of one example network resource management strategy. We demonstrate the feasibility of our approach via performance assesment of a simple DRE application using our MIDAS and commerically available OpenFlow hardware.**

I. Introduction

Various middleware technologies conforming to standards such as Real-Time CORBA [15] and the Data Distribution Service (DDS) [12] have enabled cost and development time reductions for Distributed Real-Time and Embedded (DRE) Systems. These cost and development time reductions are achieved because the middleware provides a simple, high-level abstraction that the developers of individual components can program to in order to interact with other components in the system. Additionally, the prime contractor can mandate a specific data-models and component interfaces realized via specific middleware to ensure that problems are less likely to occur during system integration.

This research was supported by ARO W911NF-11-1-0403, ONR N000141310802, and the MKE (Ministry of Knowledge Economy), Korea, under the Global Collaborative R&D program supervised by the KIAT (M002300089).

While middleware for DRE systems are engineered with features that enable deterministic behavior, the use of a DRE middleware itself is not enough to achieve end-to-end timing guarantees. Systems designers must not only tweak the configuration of the middleware *software*, but they also must configure the network to ensure that resources will always be appropriatly allocated and that faults (*i.e.,* nodes trying to saturate the network known as "babbling idiots") are properly contained. Traditionally, these configurations have been *static* meaning they are not changed once the system is fully assembled. Thus middleware for DRE has not traditionally exerted control of the network itself. This separation of concerns has been reasonable for two reasons. First, most safety-critical DRE are delivered to the customer fully integrated and assemebled. This means that the prime-contractor has *a priori* knowledege to configure the network appropriately. Second, most networking technologies used in safety-critical DRE such as TTA or TT-Ethernet only support static configurations and must be configured offline.

Recent trends in Cyber-Physical Systems are revealing a new class of DRE. In this new class of DRE, the system is dynamic: Users can add or subtract system components based on current needs or even significantly alter the network topology. A pedagogical example of this new type of DRE are plug and play medical systems. In a plug and play medical system a caregiver working in an ICU may couple physiological sensors, actuators (*e.g.,* an infusion pump) and control algorithms running on a server over the hospital network to provide a specific type of therapy. In some sense this DRE is instantiated 'on-demand' over a shared resource (the network). In this context the traditional separation between middleware and network configuration will not work; It would be burdonsome for hospital IT staff to reconfigure the network and middleware each time the patient or therapy changes.

Fortunately, the recent trend towards Software Defined Networking (*i.e.,* OpenFlow [11]) in the COTS networking space has made a middleware that automatically provides end-to-end timing guarantees in a dynamic environment a real possibility. In this paper we describe the MIDdleware Assurance Substrate (MIDAS), a publish-subscribe middleware that controls the low level switching configuration of the network via the OpenFlow protocol. We say that MIDAS provides strong real-time guarantees because the required QoS of any admitted subscriber is guaranteed for the duration of the admission assuming there is no failure of the underlying hardware. The purpose of this paper is not to provide a definitive solution in this space, but instead to show one way such a middleware should be designed and to get preliminary results on the effectiveness of using OpenFlow-enabled networking equipment in a real-time environment.

This paper is organized as follows. In Section II we define the timing parameters our middleware supports. Section III contains a brief overview of OpenFlow and how it is leveraged by MIDAS. Section IV gives the software architecture of MIDAS and gives its major components. In Section V we describe on of the strategies MIDAS can use to perform admission control and generate network configurations. Section VI describes preliminary experiments using MIDAS with real OpenFlow hardware and an application taken from the medical domain. In Section VII we describe related work. Finally we conclude in Section VIII

II. Quality of Service Parameters

The MIDAS real-time message bus provides a publish-subscribe abstraction for communication in DRE systems. We focus on the publish-subcribe paradigm for two reasons. First, publish-subscribe has proven popular for the development of DRE systems because of the loose coupling it enables between publishers (nodes that emit data *e.g.,* sensors) and subscribers (nodes that consume data *e.g.,* actuators). Second, publish-susbcribe performance mostly concerns the network. Any effort to improve publish-subscribe middleware determinism will be largely complementary to existing work on improving other DRE middleware approaches which focus more on CPU scheduling. While current publish-subscribe middleware for DRE systems such as DDS expose quality of service settings to developers the standard does not require any enforcement mechanism. Any guarantee of timing behavior must be achieved by statically configuring network hardware rate-limiter mechanisms, tuning various middleware parameters and assigning priorities to time critical messages. In this section we will draw on classical real-time scheduling theory to devise a simple set of three QoS parameters that would be appropriate for a publish-subscribe middleware that is able to provide end-to-end timing guarantees in the form of maximum end-to-end latency. Before we describe these parameters we first define latency:

Definition 1 (Network Latency)**.** *Let $\mathcal{P}_\mathcal{T}$ be a publisher publishing to topic $\mathcal{T}$ and $S_\mathcal{T}$ be a subscriber to $\mathcal{T}$. Let t_1 be the moment $\mathcal{P}_\mathcal{T}$ starts transmitting message m on the network and let t_2 be the smallest time t after m has been fully received by $\mathcal{S}_\mathcal{T}$. The end-to-end network latency of m is $\mathcal{L}\left(\mathcal{S}_\mathcal{T}, m\right) = t_2 - t_1$.*

Definition 2 (Guaranteed Maximum End-to-End Latency)**.** *Let $\mathcal{P}_\mathcal{T}$ be a publisher publishing to topic T and $\mathcal{S}_\mathcal{T}$ be a subscriber to $\mathcal{T}$. If $\mathcal{S}_\mathcal{T}$ requests a guaranteed maximum latency, denoted $\mathcal{L}_{max}\left(\mathcal{S}_\mathcal{T}\right)$, then $\forall_m \mathcal{L}\left(\mathcal{S}_\mathcal{T}, m\right) \leq \mathcal{L}_{max}\left(\mathcal{S}_\mathcal{T}\right)$.*

In MIDAS, developers can specify maximum end-to-end latency (Definition 2) using the parameter *maxLatency*, minimum (resp. maximum) separation between consecutive updates as *minSep* (resp. *maxSep*). All three parameters are used at runtime by the middleware to check if a subscriber should receive data from a publisher. While *maxSep* and *maxLatency* closely align in function with the DDS parameters *DEADLINE* and *LATENCY_BUDGET*[1] *minSep* has no DDS analog. The primary function of *minSep* is to let the middleware know how much load the publisher can induce on the system. Figure 1 illustrates how MIDAS will match QoS between publishers, subscribers, and the underlying system. Subscriber A is admitted to the system because its required $maxSep$ (20ms) is greater than or equal to the publisher's (15ms). Additionally the middleware has determined it can guarantee the requested maximum latency. Subscriber B is not admitted because it requires a $maxSep$ of 14ms which is smaller than the publisher's. Finally, Subscriber C is not admitted to the system only because the underlying middleware has determined that it cannot guarantee C's requested maximum end-to-end latency.

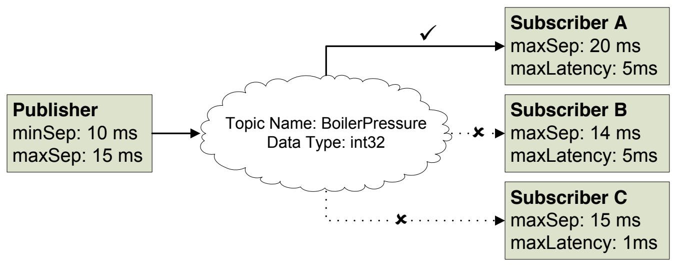

Fig. 1: Real-Time Publisher and Subscribers with QoS.

III. Approach Overview

We achieve real-time guarantees on open networks built from COTS equipment by handing complete control over the network to the middleware via OpenFlow. Tight integration of the middleware with OpenFlow provides several benefits. First, it gives the middleware complete control over how data packets on the network are forwarded, prioritized, and rate-limited. Second, many COTS switches can be made OpenFlow capable with a firmware update. This means that existing network deployments can be made OpenFlow capable. Third, in many OpenFlow switches all OpenFlow rule processing occurs at line rate. This means that the middleware can affect configuration changes in the network without any appreciable loss of network peformance.

We now describe the operation of an OpenFlow network. An OpenFlow network consists of two types of entities: OpenFlow switches and OpenFlow controllers. 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 an *action* set which tells the switch how to handle a packet matching the flow. Table I shows an example flow table. The table has two flow entries which match against input port, Ethernet address, IP address and UDP port number. There are two actions associated with each flow. While the OpenFlow specification describes a number of different actions our prototype utilizes the enqueue and meter actions. The meter action requires the switch to apply traffic policing to the flow. The enqueue action

[1]In DDS *LATENCY_BUDGET* is only used as a hint for optimzation purposes. It does not confer any notion of a guarantee.

Input Port	Datalink				IP		UDP		Action
	VLAN ID	Src	Dst	Type	Src	Dst	Src Port	Dst Port	
3	0	89ab	89ac	IP	192.168.1.1	192.168.1.2	100	101	meter=1,enqueue=4:7
4	0	89ac	89ab	IP	192.168.1.2	192.168.1.1	101	100	meter=2,enqueue=3:2

TABLE I: Example OpenFlow flow table

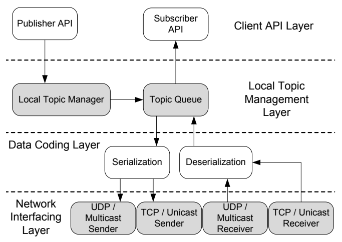

Fig. 2: Client Library

requires the switch to place the packet on an egress queue associated with a specific port during forwarding.

When an OpenFlow switch receives a packet on one of its interfaces, it compares that packet to its flow table. If the packet matches a flow table entry, it applies the action set associated with that flow entry. If the packet does not match an existing entry the switch performs what the OpenFlow protocol calls a *packet-in*. When the switch performs a packet-in, it forwards the packet to the OpenFlow controller (a piece software running on a server in the network). The controller analyzes the packet and can execute any arbitrary algorithm to generate a new flow rule. The controller can then update the switch's flow table with the new rule. Packet-in allows the OpenFlow controller to 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 with algorithms implemented in a normal high level programming language like Java or C++.

We now provide an overview of how the RTMB provides real-time guarantees on OpenFlow enabled COTS networks. The RTMB implements a Global Resource Manager (GRM) which contains a specialized OpenFlow controller. When a publish-subscribe client comes online it first connects to the GRM. This allows the GRM to learn where on the network the client is located (*i.e.,* the switch and port it is connected to). Then, when a client requests a subscription with a specified QoS, the GRM will peform *admission control*. First, the scheduling algorithm in the GRM will generate a new network configuration based on the new QoS request. The new configuration is then analyzed by a *schedulability test* which determines if any QoS constraints could be violated with that configuration (see Section V for an example scheduling and schedulability algorithm). If a violation is possible, the client is notified and their request is not granted. If QoS is guaranteed in the new configuration, the GRM commits the network configuration to the network using OpenFlow and then admits the client. Note that this system architecture allows us to handle non publish-subscribe best effort traffic (*e.g.,* web-browsing) on the same network transparently; the GRM will automatically map best effort traffic to the lowest priority queues on each switch.

IV. Middleware Design

Now we describe the various software components in the RTMB. The RTMB adopts a brokerless architecture and the functionality of middleware is separated into two software stacks implemented in Java. The client library provides the publish-subscribe abstraction to clients that wish to be publishers or subscribers. The Global Resource Manager (GRM) runs on a server connected to the network and is responsible for managing active topics, publishers, subscribers and the underlying network configuration. Both the client library and GRM have features specifically designed to enable automatic QoS guarantees.

A. Client Library

The architecture of the client library is illustrated in Figure 2. If the application is a publisher, messages flow from the application to a local topic queue by way of the local topic manager. This allows the client library to perform a *zero-copy* transfer of data between publishers and subscriber that are running on the same host. Each local topic queue always has a special subscriber: the data-coding layer. The data-coding is responsible for serializing messages prior to transmission on the network. After a message has been serialized, a *sender* object transmits it onto the network. The type of sender used depends on what transport protocol was negotiated with the GRM. Symetrically, the *receivers* receive messages from the network, pass those messages to the data coding layer where they are deserialized and then placed on the appropriate topic queue. Subscribers are invoked when the topic queue associated with their topic becomes non-empty.

The Client Library has one important feature used to support automatic QoS guarantees: it statically infers the maximum serialized message sizes from message types. When a publisher comes online it specifies the type of message it will publish. The API passes this information to the topic management layer, which in turn asks the data coding layer for message size bounds on that type. In our prototype, the data coding layer uses Java reflection to determine the structure of the type and infer the maximum number of bytes used to represent a message of that type on the network.[2] Maximum message size information is used by the GRM when it performs the schedulability analysis of a network configuration.

B. Global Resource Manager

The GRM (Figure 3) is responsible for orchestrating all activity on the network to ensure that data is correctly propagated between publishers and subscibers. To accomplish this, the GRM must maintain configuration information about the network and implement the appropriate scheduling and network reconfiguration algorithms. Because we are concerned with providing guaranteed timing, the GRM must keep record of how switches in the network are interconnected, where clients are plugged into the network, the performance characteristics of each switch, and which multicast addresses are associated with what topics.

[2]Our prototype currently only supports non-inductive types (*e.g.,* record types) that can be easily analyzed for size-bounds.

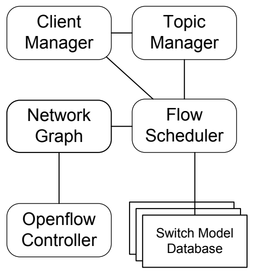

Fig. 3: Global Resource Manager Architecture

These various responsibilites are decomposed along module boundaries. Several of these modules' functions do not need to be extensively elaborated: the client manager is a server process that handles client's requests (*e.g.,* to start publishing on a topic); and the topic manager maintains a record of active topics and the network addresses associated with each topic, the OpenFlow controller implements the OpenFlow protocol and exposes a simple API to the flow scheduler to reconfigure the network.

The flow scheduler implements the admission control, scheduling and network reconfiguration algorithms used to ensure QoS constraints are not violated (see Section V).

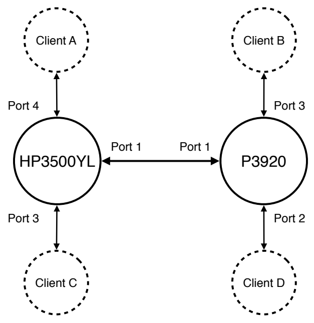

Fig. 4: Example Network Graph

We now elaborate the network graph and the switch model data in more detail. The switch model database is a repository of performance and timing characteristics for different models of OpenFlow switch. This information is vitally important to the GRM; it needs to know how each switch in its network behaves. The information in the switch model repository is created before the middleware is deployed on a network. In our protoype each switch model is represented by an XML file that is read by the GRM when the GRM starts up. Each switch model contains the model name of the switch, the number of ports on the switch, the number of egress queues associated with each port, the bandwidth capacity, and the number and precision of the hardware rate-limiters.

The network graph maintains both static and dynamic network configuration information. The static information is specified at deployment time; it defines what switches are on the network (the model, etc.) and how they interconnect. The dynamic information is either learned via OpenFlow (*e.g.,* what ports on which switch are specific client connected) or set by the flow scheduler.

Figure 4 illustrates a simple network graph. The network consists of two switches. These switches are connected via an uplink cable on each of their port 1. Each switch is connected to two clients (denoted by dotted circles).

V. Flow Scheduling

A valid flow scheduler must perform three tasks: First, when a subscriber comes online and requests a subscription to an existing topic the flow scheduler must generate a candidate network configuration using OpenFlow configuration primitives. Second, the scheduler must analyze the new configuration and determine if it guarantees the timing constraints of all admitted flows plus the new one. Third, if the new configuration is acceptable the scheduler must reconfigure the network carefully so no constraints of existing flows are violated during the reconfiguration. All three of these activities are non-trivial: Distributed scheduling is known to be NP-Hard if an optimal schedule is desired and there are no known exact schedulability tests for the general network setting [6]. In light of these difficulties, we do not describe an optimal approach. Instead, we focus on a strategy that is both fast (*i.e.,* polynomial-time in the size of the network), exploits the types of configuration primitives by OpenFlow, and allows for the safe transition from one valid configuration to another. Analysis and improvement of the approach in terms of network utilization and schedulability is left for future work. We start be describing how the strategy would apply to a network consisting of a single switch, then extend it to the multi-switch case.

A. Single Switch Scheduling

The flow scheduler generates a candidate network configuration in several phases. First, for each publish-subscribe relationship the flow scheduler queries the OpenFlow controller to determine what switch port each publisher and is connected to and the network address associated with a given topic. Then, for each publisher $\mathcal{P}$ publishing to $\mathcal{T}$, the flow scheduler configures a rate-limiter. The rate-limiter is configured with a maximum burst size B and and maximum rate R, and is set to apply to all packets that enter the switch on the port connected to $\mathcal{P}$ destined to the network address associated with $\mathcal{T}$. If a publisher $\mathcal{P}$ specifies a minimum separation between each message of $minSep$, and maximum message size of M, then the burst size and rate are set as follows:

$$B = M, \quad R = \frac{M}{minSep}$$

This allows $\mathcal{P}$ to burst its entire message onto the network while ensuring that $\mathcal{P}$ cannot overload the network if $\mathcal{P}$ becomes a babbling idiot.

Before we can describe how the flow scheduler prioritizes flows we need to explain how to calculate upper bounds on the worst case latency of message. Latency in a switched network has a number of sources. The first is due to the bandwidth of the network link. The second is due to the physical wire that connects a network node to a switch: an electrical signal takes time to propagate along a wire (in most networks the latency effects of the wires are small because they are relatively short). The third is the multiplexing latency of the switch. Switch multiplexing latency is the time it takes a switch to move a bit entering the switch on one port to the output queue of another. On modern non-blocking switches this is usually on the order of several microseconds. Finally, there is queuing latency, which is the amount of time a message spends waiting in an egress queue. In a modern switched Ethernet all these latencies are fixed (*i.e.,* do not change due to network load) except for queuing latency. Messages placed from different flows placed on queues associated with the same switch port are in contention for shared "forwarding resources." We now formally define the fixed latency, queuing latency, and end-to-end latency for a single switch.

Definition 3 (Wire Latency)**.** *The function $w(N_1, N_2)$ denotes the signal propagation latency between network stations N_1 and N_2. A network station can be either a switch, or a publisher/subscriber.*

Definition 4 (Fixed Latency)**.** *Let $f = (\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T})$. Let the maximum message size of a message publish to topic $\mathcal{T}$ be M. Then the fixed portion of the end-to-end latency, denoted $\mathcal{L}_F(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T})$, between $\mathcal{P}_\mathcal{T}$ and $\mathcal{S}_\mathcal{T}$ is:*

$$\mathcal{L}_F(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T}) = \frac{M}{C} + w(\mathcal{P}_\mathcal{T}, s) + w(\mathcal{S}_\mathcal{T}, s) + s^{mux} \quad (1)$$

where C is the network bandwidth and s^{mux} is the multiplexing latency of switch s

Definition 5 (Queing Latency)**.** *Let $(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T})$ be the flow from $\mathcal{P}_\mathcal{T}$ to $\mathcal{S}_\mathcal{T}$, and let $s(i)$ be the i^{th} port on switch s which the flow is routed out of, then the queuing latency of the flow $(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T})$ with priority p at switch/port $s(i)$ is $\mathcal{Q}(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T}, s(i), p)$.*

Definition 6 (End-to-End Latency)**.** *The end-to-end latency is the sum of the fixed and queuing latency:*

$$\mathcal{L}_{e2e}(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T}) = \mathcal{L}_F(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T}) + \mathcal{Q}(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T}, s(i), p) \quad (2)$$

How can we calculate $\mathcal{Q}(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T}, s(i), p)$? We adapt an approximate technique for calculating the response time of a task under fixed priority scheduling on a uniprocessor. In [5], Bini *et al.* provide a linear equation for calculating an upperbound worst case response time of a task. Assuming P_i is the minimum separation between consecutive arrivals of task T_i, E_i is the worst case execution time and $hp(i)$ is the set of tasks assigned priority higher than T_i, then the response time R_i is bounded from above by:

$$R_i^{ub} = \frac{E_i + \sum_{j \in hp(i)} E_j \left(1 - \frac{E_j}{P_j}\right)}{1 - \sum_{j \in hp(i)} \frac{E_j}{P_j}} \quad (3)$$

This equation is useful in our application because the per-task workload approximations Bini *et al.* used to derive the response time bound also approximate the traffic pattern of a flow conforming to a rate-limiter. We then transform Equation 4 into a worst-case bound on latency due to queuing by substituting message sizes divided by bandwidth for execution cost, $minSep$ for the periods, and subtracting the overall message transmission cost for our flow[3] Additionally, let $hp(p, s(i))$ be the set of flows with priority higher than p at port i on switch s:

$$\mathcal{Q}(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T}, s(i), p)^{ub} = \frac{\frac{M_\mathcal{T}}{C} + \sum_{j \in hp(p, s(i))} \frac{M_j}{C}\left(1 - \frac{\frac{B_j}{C}}{minSep_j}\right)}{1 - \sum_{j \in hp(p,s(i))} \frac{\frac{B_j}{C}}{minSep_j}} - \frac{M_\mathcal{T}}{C} \quad (4)$$

We can now use the upper-bound on worst-case switch latency to determine how to prioritize each flow. Common techniques for priority assignment in real-time systems include the Rate Monotonic (RM) and Deadline Monotonic orderings (DM) [3]. Unfortunately, both RM and DM theory require that each flow is assigned a unique priority. This is not possible on real networking hardware: most Ethernet switches only provide 8 priority queues per port for egress traffic. To overcome this limitation, we use Audsley's Optimal Priority Assignment (OPA) algorithm [2]. OPA has two desireable properties: It is optimal locally (if a flow set will meet its latency requirement at a single switch under any fixed-priority configuration it will also under OPA) and it minimizes the number or priority levels required to schedule the flow set. Because each port of the switch is independent in terms of its egress queueing, we only need to differentiate the priorities of flows exiting the switch on the same port.

We now describe a version of Audsley's OPA adapted to assign priorities to flows in an OpenFlow switch. Our modified OPA takes as input a set of flows (denoted $\mathcal{F}$) forwarded out of the same port. OPA starts by attempting to assign flows to the lowest priority level. If a flow f can exceeed its latency bounds at a given priority level, OPA moves on and will attempt to assign that flow a higher priority later. Conservatively, a flow $(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T})$ can miss its latency bounds if $\mathcal{Q}(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T}, s(i), p)^{ub} + \mathcal{L}_F(\mathcal{P}_\mathcal{T}, \mathcal{S}_\mathcal{T}) > \mathcal{L}_{max}(\mathcal{S}_\mathcal{T})$. If OPA exits before assigning a priority to every flow, then the flow set is not schedulable with *any* fixed priority assignment. If the number of priority levels required to schedule $\mathcal{F}$ is greater than the number of priorities provided by the switch, then the flow scheduler deems the flow set unschedulable.

Before admitting a new subscriber the flow scheduler must reconfigure the network to accomodate the new flow without causing existing flows to violate their latency requirements. Existing flows must be migrated to their new priorities in a specific order to avoid *priority inversions*. To safely accomplish the reconfiguration the flow scheduler maps existing priorities according to their priority assignment in the new configuration: flows with lower priority are reprioritized first.

B. Extension to Multi-switch

The prototype flow scheduler supports real-time guarantees on networks consisting of multiple switches by tranforming the

[3]This is because the response time for a task also takes into account the execution time for that task. We only want a upper bound on the interference.

distributed scheduling problem into a sequence of local (*i.e.,* single switch) scheduling problems. Before we proceed we modify Equations 5 and 2 to describe the sources of latency for a flow that is forwarded through a sequence of switches. As in the single switch case there are fixed and queuing sources of latency:

Definition 7 (Multiswitch Fixed Latency)**.** *Let ρ be a path of length m through the network from $\mathcal{P}_{\mathcal{T}}$ to $\mathcal{S}_{\mathcal{T}}$. Let N_k be the k^{th} network node (switch or publisher/subscriber) on ρ.Let the maximum message size of a message publish to topic $\mathcal{T}$ be M. Then the fixed portion of the end-to-end latency, denoted $\mathcal{L}_F^{\rho}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right)$, between $\mathcal{P}_{\mathcal{T}}$ and $\mathcal{S}_{\mathcal{T}}$ is:*

$$\mathcal{L}_F^{\rho}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) = \frac{M}{C} + \sum_{1<k\leq m} w\left(N_{k-1}, N_k\right) + \sum_{1<k<m} s_k^{mux} \tag{5}$$

Definition 8 (Multiswitch Queuing Latency Latency)**.** *Let ρ be a path through the network with length m from $\mathcal{P}_{\mathcal{T}}$ to $\mathcal{S}_{\mathcal{T}}$. Then the queuing latency due to all the switches on ρ is the sum of all the queuing latencies of the switches along the path:*

$$\mathcal{Q}^{\rho}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) = \sum_{1<k<m} \mathcal{Q}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}, s_k\left(i\right), p_k\right) \tag{6}$$

Definition 9 (Total Multiswitch End-to-End Latency)**.** *Let ρ be a path through the network crossing m switches. Then the total end-to-end latency due to both fixed and queuing delays along ρ is:*

$$\mathcal{L}_{e2e}^{\rho}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) = \mathcal{Q}^{\rho}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) + \mathcal{L}_F^{\rho}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) \tag{7}$$

Given these equations for end-to-end latency for flows crossing multiple switches we describe how MIDAS generates and applies network configurations for multi-switch networks. As mentioned earlier in this section distributed scheduling is in general quite difficult. Further complicating matters is that MIDAS must be able to reconfigure the the entire network without causing any QoS constraint violations for existing flows. This is challenging because the reconfiguration of an upstream switch will impact the worst case load on downstream switches. Imagine for example a simple network consisting of two switches s_1 and s_2. Now imagine some flow f forwarded along the path s_1, s_2. Say that the minimum separation between bursts of f at s_1 is 20ms and the worst case queuing latency at s_1 is 3ms. This means that the minimum separtion that could be observed by s_2 is 17ms (the case where the first burst of f is delayed the maximum amount and then the second burst is not delayed at all). Now assume a new flow f' is admitted to the network and it is prioritized higher than f on s_1. This will increase the worst-case queuing latency of f (*e.g.,* to 10ms) at s_1 and further contract the worst-case burst separation observed by s_2 (down to 10ms).

We avoid having to calculate network-wide side effects each time a new subscriber is admitted by transforming the distributed scheduling problem into a sequence of local scheduling problems: When a subscriber $\mathcal{S}_{\mathcal{T}}$ requests a subscription to $\mathcal{T}$ with a latency constraint $\mathcal{L}_{max}\left(\mathcal{S}_{\mathcal{T}}\right)$ we first calculate the shortest unweighted path ρ between $\mathcal{P}_{\mathcal{T}}$ and $\mathcal{S}_{\mathcal{T}}$. Next, we uniformly allot a portion $\mathcal{L}_{max}\left(\mathcal{S}_{\mathcal{T}}\right)$ to each switch: for each switch s_k in ρ we calculate $\mathcal{L}_{max}\left(\mathcal{S}_{T}\right)_{s_k}$ where:

$$\mathcal{L}_{max}\left(\mathcal{S}_{T}\right)_{s_k} = \frac{\mathcal{L}_{max}\left(\mathcal{S}_{\mathcal{T}}\right) - \mathcal{L}_F^{\rho}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right)}{|\rho|}$$

That is, we split the allowed queuing latency up evenly between all the switches along ρ. We now recursively calculate the worst case minimum separation observed at each switch on the path. Let $minSep_k$ be the minimum worst case separation of bursts at switch s_k then:

$$minSep_{k+1} = minSep_k - \mathcal{L}_{max}\left(\mathcal{S}_{T}\right)_{s_k}$$

Finally, we apply the single switch schedulability, priority assignment and network reconfiguration algorithms using each $\mathcal{L}_{max}\left(\mathcal{S}_{T}\right)_{s_k}$ and $minSep_k$ for the appropriate switch. Because we fixed the allotted switch queuing latency when the flow as admitted, the $minSep_k$ values will never change.

VI. Experimental Evaluation

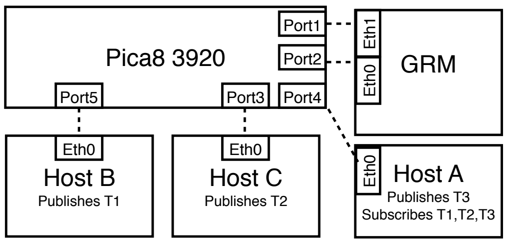

Fig. 5: Network Layout

We evalauted two aspects of MIDAS. First we wanted to see if the network scheduling used in the MIDAS improved the timing performance relative to that of a standard switch. Second, we wanted to see how robust the MIDAS timing guarantees are. In order to evaluate these two aspects we deployed the MIDAS on our OpenFlow test bench (Figure 5).

Our OpenFlow test-bench consists of 4 computers and an OpenFlow capable switch, a Pica8 P3290 [1]. Each of the 4 computers were plugged into the switches' *data-plane* ports (*i.e.,* OpenFlow managed ports). The GRM was also plugged into the *control-plane* port which carries OpenFlow management traffic. Measuring end-to-end timing in a distributed network accurately is challenging due to clock synchronization issues. We avoid these synchronization issues by exploiting OpenFlow to let us run publisher's and subscribers on the same hosts: we add an OpenFlow rule that causes the switch to intercept packets from certain flows, rewrite the packet headers, and then retransmit the packet back out the port it arrived on. This allows us to 'fool' the client; it can publish to $\mathcal{T}_x$ and subscribe to $\mathcal{T}_y$ but in reality it the messages being published to $\mathcal{T}_x$ are being sent back modified so they look as if they are from $\mathcal{T}_y$. This allows us to compare the timestamps of messages using the same system clock while still subjecting the

Topic	$minSep$	Max. Latency	Message Size (Bytes)	Bandwidth
$\mathcal{T}_1$	3ms	2ms	192192	512.512mbit/s
$\mathcal{T}_2$	3ms	3ms	96000	256.000mbit/s
$\mathcal{T}_3$	11ms	8ms	64000	46.545mbit/s
TOTAL:				815.057mbit/s

TABLE II: Experimental Publish-Subcribe Set

message to the same queuing, multiplexing and wire latencies it would experience if it was being sent to another host.

All timing measurements we done on Host A. Host A was running real-time Linux with IBM's RTSJ-compliant Real-Time JVM. The RTMB client library on Host A was scheduled with the highest system priority using RTSJ Java's `NoHeapRealtimeThreads` to ensure that they would not be interfered with by the Java garbage collector or other processes on the system. All timing measurements were made by calling Java's `System.currentTimeMillis()`. Prior to running our experimental scenarios we lower-bounded the amount of latency added by the Linux TCP/IP stack and the JVM by sending a message to the loopback interface. This latency was consistently 1ms, which means that observed latencies as recorded by the software are usually 1ms more than the actual network latency.

For each experiment we used the same 3 publishers each publishing to a different topic ($\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}_3$) with a single host subscribing to each topic. Table II lists each topic, relevent QoS ($minSep$ of the publisher and max latency from the subscriber), and the bandwidth required by each. The publish-subscribe set is designed to be representative of a demanding plug and play medical system: $\mathcal{T}_1$ represents a high framerate/resolution video stream, $\mathcal{T}_2$ & $\mathcal{T}_3$ represent high resolution data coming from ECG and EEG machines. For each experiment we captured all messages received within a 10 second window and recorded their latencies. During following discussion, we use $\mathcal{S}_x$ to denote the subscriber to topic x and $\mathcal{P}_x$ as the publisher to x.

A. *Scenario 1: Comparison to Best-Effort*

Here we compare the performance of the middleware in two network settings. In the first setting, we configure the Pica8 to behave like a normal L2/L3 switch (it uses a fair-queueing strategy to forward ethernet frames in this mode). We call this the 'best-effort' setting. In the second setting we place the network under control of the MIDAS using the strategy of Section V. We ran three experiments where we observed the end-to-end latencies of messages published to each topic. In the best-effort setting $\mathcal{S}_{\mathcal{T}_3}$ still met its latency constraints. The same was not true for $\mathcal{S}_{\mathcal{T}_2}$ or $\mathcal{S}_{\mathcal{T}_1}$. Due to space, we report the data concerning $\mathcal{S}_{\mathcal{T}_2}$:

Figure 6 shows the latency of each message over the observation window for $\mathcal{S}_{\mathcal{T}_2}$. Figure 7 shows the same for the MIDAS managed setting. Each point on each graph represents the end-to-end latency of a single message sent to $\mathcal{T}_2$ and received by the subscriber. The x-axis is the moment (in milliseconds) that the message was transmitted. The y-axis is the latency of that message. Even accounting for jitters in the operating system and JVM the end-to-end deadline of $\mathcal{S}_{\mathcal{T}_2}$ is repeatedly violated on the best-effort system (observe the

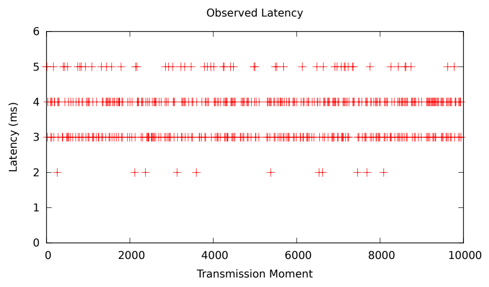

Fig. 6: Best Effort

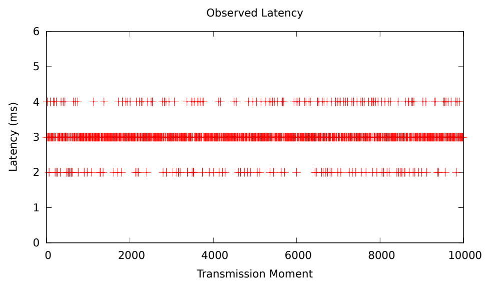

Fig. 7: MIDAS

number of samples in the 5ms row of Figure 6). Additionally, $\mathcal{S}_{\mathcal{T}_2}$ never received 48% of the messages that were sent. This is because the messages are quite large and the egress queues can be overrun in the best effort setting. Any loss of a single ethernet frame will result in the loss of the whole message. All messages arrived in the MIDAS-managed setting. While it would be possible to reduce the message drop rate in the best effort setting by causing the senders to retransmit on failure, doing so would increase the effective latency of the message. Taking into account the 1 ms latency added by the JVM and TCP/IP stack, no messages violated the latency requirement when the MIDAS was managing the network configuration (All samples in Figure 7 are 4ms or less).

B. *Scenario 2: Fault Containment*

In this scenario we modify the publishers to $\mathcal{T}_1$ and $\mathcal{T}_2$ so they simulate babbling idiots (*i.e.,* set their $minSep$ to 0) and we record the latencies of messages flowing to $\mathcal{T}_3$. In this experiment the publishers were able to saturate a 1 gigabit per second Ethernet link each. We modified $\mathcal{P}_{\mathcal{T}_1}$ and $\mathcal{P}_{\mathcal{T}_2}$ because MIDAS will configure their respective flows with the highest priority which means they have the most opportunity to starve the other flows if they misbehave. This represents a worst case scenario for our approach. When run on the best effort network (*i.e.,* with no flow prioritization) $\mathcal{P}_{\mathcal{T}_1}$ and $\mathcal{P}_{\mathcal{T}_2}$ were able to starve enough of the network forwarding capacity from the flow associated with $\mathcal{P}_{\mathcal{T}_3}$ to cause *all* messages to be dropped. Figure 8 contains the observed latencies when MIDAS was managing the network. Again, each point in the graph represents a single message. The y-axis is latency of that message, and its x-value is the moment the message was transmitted. Under MIDAS, no messages were dropped and all messages arrived earlier than their required latency bounds, $8ms$.

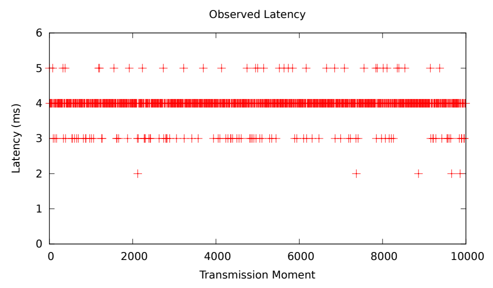

Fig. 8: Latency bounds for $\mathcal{S}_{\mathcal{T}_3}$ when $\mathcal{P}_{\mathcal{T}_1}$ and $\mathcal{P}_{\mathcal{T}_2}$ are malfunctioning

VII. Related Work

To our knowledge the RTMB is the first example of a publish-subscribe middleware that uses OpenFlow to provide real-time guarantees on COTS Ethernet networks. Most research into middleware for distributed real-time systems can be divided into two categories. The first category involves research into how various CPU scheduling and determinism features can be used in middleware to effectively support the predictable execution of distributed tasks in a distributed environment. Examples of such work include TAO [15] and the many middleware other real-time Corba [14] middleware works such as FC-ORB[16], QuO [17] and [8].

The other, less extensively studied, category includes middleware which tries to achieve deterministic network behavior by coordinating the activity of the application nodes. Examples of such work are [10], FTT-Ethernet [13] and the Synchronous Scheduling Service for RMI [4]. These approaches all offer some notion of guarantee but they are not robust because they depend on the cooperation of each node on the network: if a node does not cooperate (either due to a fault or malicious activity) then that node can distrupt the whole network.

There have also been a number of projects where OpenFlow has been used to provide some type of QoS guarantee. However, these projects have not focused on real-time systems aspects. Instead, their application focused on data center centric QoS (like minimum guaranteed bandwidth) [9] or for multimedia systems [7].

VIII. Conclusion

The work in this paper represents a first step towards DRE middleware that can automatically provide strong real-time guarantees in a dynamic environment. We believe support for providing these guarantees in a dynamic environment will be critical for the success of newly emerging types of Cyber-Physical Systems. We described MIDAS, a prototype publish-subscribe middleware which uses OpenFlow to manage the underlying network and provide guarantees for real-time QoS in order to illuminate some of the architectural and technical issues that apply to middleware designed for this environment.

Our initial evaluations showed that our prototype does enable more deterministic timing behavior. Even with a relatively high network load of 815 megabits per second all publish-subscribe network flows satisfied their millisecond-level timing requirements while on the normal Ethernet network latency constraints were violated and almost half the messages were dropped. The evaluations also showed that MIDAS' ability to provide guarantees is robust: when we reconfigured two publishers as babbling idiots MIDAS prevented the remaining flow from deviating from its specified QoS constraints.

We believe the results in the paper encourage further research into OpenFlow and how it can be used to benefit DRE middleware. Such research topics include better scheduling and reconfiguration algorithms designed for use with the OpenFlow primitives, other types of QoS (beyond timing) that could benefit from full network control, and ways to detect and adapt to network faults dynamically.

References

[1] Pica8 3290 Product Literature (2013), http://www.hp.com/rnd/products/switches/HP_ProCurve_Switch_5400zl_3500yl_Series/specs.htm

[2] Audsley, N., Dd, Y.: Optimal priority assignment and feasibility of static priority tasks with arbitrary start times (1991)

[3] Audsley, N.C.: Deadline monotonic scheduling (1990)

[4] Basanta-Val, P., Almeida, L., Garcia-Valls, M., Estevez-Ayres, I.: Towards a synchronous scheduling service on top of a unicast distributed real-time java. In: Real Time and Embedded Technology and Applications Symposium, 2007. RTAS'07. 13th IEEE. pp. 123–132. IEEE (2007)

[5] Bini, E., Nguyen, T.H.C., Richard, P., Baruah, S.: A response-time bound in fixed-priority scheduling with arbitrary deadlines. Computers, IEEE Transactions on 58(2), 279 –286 (feb 2009)

[6] Bouillard, A., Jouhet, L., Thierry, É.: Tight performance bounds in the worst-case analysis of feed-forward networks. In: INFOCOM, 2010 Proceedings IEEE. pp. 1–9. IEEE (2010)

[7] Egilmez, H.E., Dane, S.T., Bagci, K.T., Tekalp, A.M.: Openqos: An openflow controller design for multimedia delivery with end-to-end quality of service over software-defined networks. In: Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific. pp. 1–8. IEEE (2012)

[8] Eide, E., Stack, T., Regehr, J., Lepreau, J.: Dynamic cpu management for real-time, middleware-based systems. In: Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE. pp. 286–295. IEEE (2004)

[9] Kim, W., Sharma, P., Lee, J., Banerjee, S., Tourrilhes, J., Lee, S.J., Yalagandula, P.: Automated and scalable qos control for network convergence. Proc. INM/WREN 10, 1–1 (2010)

[10] Marau, R., Almeida, L., Sousa, M., Pedreiras, P.: A middleware to support dynamic reconfiguration of real-time networks. In: Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on. pp. 1–10 (Sept)

[11] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. 38(2), 69–74 (Mar 2008), http://doi.acm.org/10.1145/1355734.1355746

[12] Pardo-Castellote, G.: Omg data-distribution service: architectural overview. In: Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on. pp. 200–206 (May)

[13] Pedreiras, P., Gai, P., Almeida, L., Buttazzo, G.C.: Ftt-ethernet: a flexible real-time communication protocol that supports dynamic qos management on ethernet-based systems. Industrial Informatics, IEEE Transactions on 1(3), 162–172 (2005)

[14] Schmidt, D., Kuhns, F.: An overview of the real-time corba specification. Computer 33(6), 56–63 (2000)

[15] Schmidt, D.C., Levine, D.L., Mungee, S.: The design of the tao real-time object request broker. Comput. Commun. 21(4), 294–324 (Apr 1998), http://dx.doi.org/10.1016/S0140-3664(97)00165-5

[16] Wang, X., Chen, Y., Lu, C., Koutsoukos, X.: Fc-orb: A robust distributed real-time embedded middleware with end-to-end utilization control. Journal of Systems and Software 80(7), 938–950 (2007)

[17] Zinky, J.A., Bakken, D.E., Schantz, R.E.: Architectural support for quality of service for corba objects. Theory and Practice of Object Systems 3(1), 55–73 (1997)