Wide Area Network-scale Discovery and Data Dissemination in Data-centric Publish/Subscribe Systems *

Kyoungho An† and Aniruddha Gokhale
EECS Dept, Vanderbilt Univ
Nashville, TN, USA
{kyoungho.an,a.gokhale}@vanderbilt.edu

Sumant Tambe
Real-Time Innovations
Sunnyvale, CA, USA
sumant@rti.com

Takayuki Kuroda
Knowledge Discovery
Research Lab, NEC Corp
Kawasaki, Kanagawa, Japan
t-kuroda@ax.jp.nec.com

ABSTRACT

Distributed systems found in application domains, such as smart transportation and smart grids, inherently require dissemination of large amount of data over wide area networks (WAN). A large portion of this data is analyzed and used to manage the overall health and safety of these distributed systems. The data-centric, publish/subscribe (pub/sub) paradigm is an attractive choice to address these needs because it provides scalable and loosely coupled data communications. However, existing data-centric pub/sub mechanisms supporting quality of service (QoS) tend to operate effectively only within local area networks. Likewise broker-based solutions that operate at WAN-scale seldom provide mechanisms to coordinate among themselves for discovery and dissemination of information, and cannot handle both the heterogeneity of pub/sub endpoints as well as the significant churn in endpoints that is common in WAN-scale systems. To address these limitations, this paper presents *PubSubCoord*, which is a cloud-based coordination and discovery service for WAN-scale pub/sub systems. *PubSubCoord* realizes a WAN-scale, adaptive, and low-latency endpoint discovery and data dissemination architecture by (a) balancing the load using elastic cloud resources, (b) clustering brokers by topics for affinity, and (c) minimizing the number of data delivery hops in the pub/sub overlay. *PubSubCoord* builds on ZooKeeper's coordination primitives to support dynamic discovery of brokers and pub/sub endpoints located in isolated networks. Empirical results evaluating the performance of *PubSubCoord* are presented for (1) scalability of data dissemination and coordination, and (2) deadline-aware overlays employing configurable QoS to provide low-latency data delivery for topics demanding strict service requirements.

Categories and Subject Descriptors

C.2.4 [**Computer Systems**]: Distributed Systems—*publish/subscribe*; D.2.11 [**Software Engineering**]: Architectures—*middleware*

Keywords

Publish/subscribe, Distributed Systems, WAN

1. INTRODUCTION

Distributed systems found in application domains, such as transportation, advanced and agile manufacturing, smart energy grids, and other industrial use cases of Internet of Things, are increasingly assuming wide area network (WAN) scale, e.g., Industrial Internet Reference Architecture. [1] These systems are characterized by a large collection of heterogeneous entities, some of which are sensors that sense various parameters across the span of the system and disseminate these information for processing so that appropriate control operations are actuated over different spatio-temporal scales to manage different properties of the system. The key to the success of these systems relies on how effective is the system in collecting and delivering data across large number of heterogeneous entities at internet scale in a manner that satisfies different quality of service (QoS) properties (e.g., timeliness, reliability, and security).

The publish/subscribe (pub/sub) communication paradigm is a promising solution since it provides a scalable and decoupled data delivery mechanism between communicating peers. Many industrial and academic pub/sub solutions exist [4, 7, 9, 12, 18, 22, 27]. Some of these even support the desired QoS properties, such as availability [9, 22], configurable reliability [18], durability [12], and timeliness [6, 15]. However, these solutions tend to support only one QoS property at a time and in most cases, the support for configurability and dynamic adaptation is lacking. More importantly, dynamic discovery of heterogeneous endpoints, which is a key requirement, is often missing in these solutions.

One approach to supporting WAN-based pub/sub relies on broker-based solutions [1, 10, 14, 17] because this approach can solve practical issues when a system is deployed

*This work is supported in part by NSF CAREER CNS 0845789. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF.

†Author is currently with Real-Time Innovations

[1] http://www.iiconsortium.org/IIRA.htm

ISIS Technical Report ISIS-15-120 Nashville, TN, USA 2015.

in network environments that use network address translation (NAT) or firewall or do not support multicast. However, as industrial systems progressively integrate other subsystems located in multiple disparate networks, the number of deployed brokers becomes very large. Consequently, the amount of effort to manage these dispersed brokers becomes unwieldy. Moreover, forming an efficient overlay network of brokers that offers both scalability and low latency becomes even harder.

To address these key requirements, we present *PubSubCoord,* which is a cloud-based coordination service for geographically distributed pub/sub brokers to transparently connect heterogeneous endpoints and realize internet-scale data-centric pub/sub systems. Specifically, this paper addresses the following challenges in the context of scalable, reliable, and dynamic pub/sub systems:

- **Scalability and Availability:** To address the scalability and availability needs of data dissemination across WAN-scale systems despite NAT/firewall issues and failures, PubSubCoord uses the separation of concerns principle to decouple local area-based brokers called *edge brokers* that handle local pub/sub issues from cloud-based brokers called *routing brokers* that handle routing between edge brokers (See Section 2.2).

- **Dynamic Discovery and Dissemination:** To support dynamic discovery and data routing between brokers, PubsubCoord provides efficient coordination among the brokers by building pub/sub-specific event notifications using the basic primitives provided by ZooKeeper coordination service [13]. This solution helps to synchronize the dissemination paths over the dynamic overlay network of brokers and heterogeneous endpoints (See Section 2.3).

- **Overload and Fault Management:** To manage topic and dissemination overload, PubSubCoord uses cloud-based elasticity to balance the load. Load balancing and broker failures are handled by an elected leader (See Section 2.4).

- **Performance Optimizations:** For those dissemination paths that need both low latency and reliability assurances, PubSubCoord trades off resource usage in favor of deadline-aware overlays that build multiple, redundant paths between brokers (See Section 2.4.3).

Our PubSubCoord design can easily be adopted by industrial systems because its design is based on proven software engineering design patterns. We have favored maximal reuse of proven industrial-strength solutions wherever possible instead of reinventing the wheel. A key guiding principle for us was to ensure a *non-invasive and extensible design* which preserves the endpoint discovery and data dissemination model of the underlying pub/sub messaging system by tunneling discovery and dissemination messages across the hierarchy of brokers. Using this approach, it is possible to support multiple concrete pub/sub technologies without breaking their individual semantics. We present extensive empirical test data to validate our claims. The experimental results are demonstrated concretely in the context of endpoints that use the OMG Data Distribution Service (DDS) [20] as the underlying pub/sub messaging system. The source code and experimental apparatus of PubSubCoord are made available in open source.[2]

The remainder of this paper is organized as follows: Section 2 describes the design and implementation of PubSubCoord; Section 3 shows experimental results validating our claims; Section 4 compares PubSubCoord with related work; and Section 5 presents concluding remarks and alludes to future work.

2. DESIGN OF PUBSUBCOORD

This section presents the PubSubCoord architecture and the rationale for the design decisions. We present each solution explaining the context, the design patterns and frameworks used, and how the consequences from applying the patterns are resolved. We then provide details on the runtime interactions between the elements of the architecture. We then allude to some implementation details.

2.1 PubSubCoord Architecture: The Big Picture

Figure 1 shows the layered PubSubCoord architecture depicting three layers: a coordination layer, a pub/sub broker overlay layer, and the physical network layer. The pub/sub broker overlay comprises a broker hierarchy based on a separation of concerns representing the logical network of brokers and endpoints in a system. An *edge broker* is directly connected to individual endpoints in a local area network (LAN) (*i.e.*, which represents an isolated network) to serve as a bridge to other endpoints placed in different networks. A *routing broker* serves as a mediator to route data between edge brokers according to assigned and matched topics that are present in the global data space. The coordination layer comprises an ensemble of ZooKeeper servers used for coordination between the brokers.

The data dissemination in PubSubCoord is explained using an example from Figure 1. $Pi\{T\}$ denotes a publisher i that publishes topic T (similarly for a subscriber $Sj\{T\}$). In the example, since publisher $P1$ and subscriber $S1$ are the only endpoints interested in topic A, they communicate within their local network A only using the techniques provided by the underlying pub/sub technology. On the other hand, $P2$, $P4$, and $S2$ are interested in topic B but are deployed in different isolated networks. So their communications are routed through a routing broker that is responsible for topic B. The network transport protocol between brokers is configurable, but TCP is used as a default transport to ensure reliable communication over WANs. As seen from this example, a maximum of 2 hops on the overlay network are incurred by data flowing from one isolated network to another (*e.g.*, network A to B).

2.2 Addressing Scalability and Availability Requirements

Context: Traditional WAN-based pub/sub systems tend to form an overlay network of brokers to which endpoints can be connected. The brokers exchange subscriptions they receive from subscribers which are used to build routing paths from publishers. The main challenge in this approach stems from having to build routing states among brokers to route data efficiently according to matching subscribers. Secondly,

[2]URL for download: `www.dre.vanderbilt.edu/~kyoungho/pubsubcoord`.

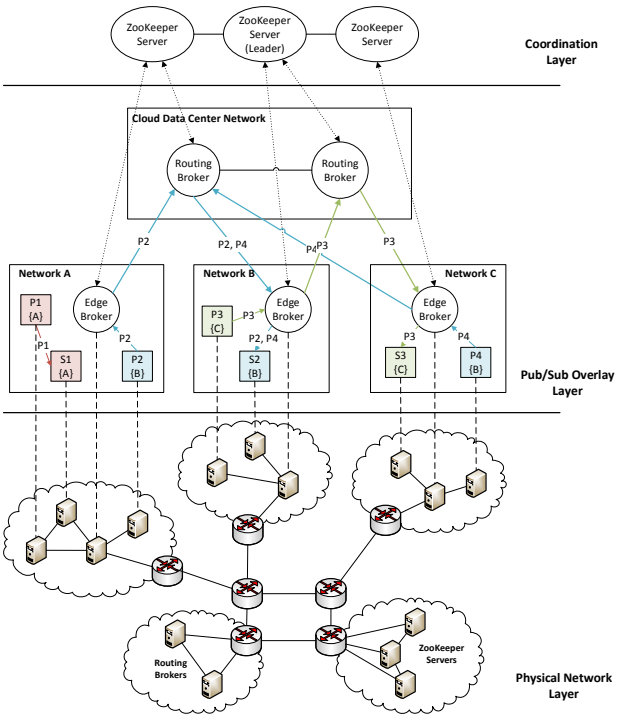


Figure 1: Layering and Separation of Concerns in the PubSubCoord Architecture

in traditional broker-based pub/sub systems, if some broker were to fail, it halts not only the pub/sub service for endpoints connected to this broker but also service for endpoints connected to other brokers that use this failed broker as an intermediate routing broker.

Design: To resolve these duo of challenges, PubSubCoord's broker overlay layer is structured as a two-tier architecture of brokers with strict separation of responsibilities: at the first tier are *edge brokers* that manage pub/sub issues within an isolated network, and at the second tier are *routing brokers*, which route traffic among different edge brokers. Our solution clusters edge brokers by matching topics and routes data through routing brokers. Each routing broker is responsible for handling only a certain number of topics so as to balance the load, minimize the overall amount of data exchanged and number of connections between edge brokers.

Consequences and resolution: The immediate consequence of such a design decision is having to decide how many routing brokers to maintain, how many topics to be handled by each routing broker, and how to organize them in the system. Having only one routing broker would be problematic since it cannot scale to handle the substantial routing load stemming from the dissemination of different topic data among the large number of endpoints. Having multiple routing brokers and organizing them in multiple levels of hierarchy similar to DNS would not be acceptable either since it would complicate the management of topics and recovery from failures because the routing state and topic management would get distributed across multiple levels. Secondly, multiple levels as in DNS introduces multiple routing hops, which will impact latency of distribution. For that reason, we maintain a flat tier of routing brokers.

The number of routing brokers and topics managed by each routing broker is determined by the end-to-end performance requirements of the pub/sub flows. Thus, a solution that can elastically scale the number of routing brokers and balance the number of topics handled by each broker is needed. For that reason, the routing broker tier is placed as a cluster in the cloud where resources can be elastically scaled up/down depending on the demand. This capability allows us to dynamically adapt to system load and scale to existing demand.

2.3 Addressing Dynamic Endpoint Discovery and Dissemination Requirements

Context: In the WAN-style systems of interest to us, matching publishers and subscribers are most likely distributed in separate isolated networks, and may join or leave dynamically. Thus, publishers and subscribers must be able to dynamically discover each other and a dissemination route needs to be established between the communicating entities.

So far we described the design rationale for a 2-tier broker architecture, which helps resolve issues stemming from having to maintain substantial pub/sub routing states but did not show how endpoints in isolated networks are discovered and how the routes are established dynamically based on endpoint discovery.

Design: To address this need, PubSubCoord provides a coordination layer (top layer shown in Figure 1) comprising an ensemble of ZooKeeper [13] servers, which help brokers discover each other and build broker overlay networks using the PubSubCoord coordination logic. ZooKeeper is an industrial-strength solution that provides generic primitives for developing domain-specific coordination capabilities for distributed applications.

The ZooKeeper service consists of an ensemble of servers that use replication to accomplish high availability with high performance and relaxed consistency. Many coordination recipes using ZooKeeper exist (e.g., leader election, group membership, and sharing configuration metadata) that are needed by distributed applications. PubSubCoord builds upon these recipes to develop coordination logic for broker interaction. ZooKeeper also provides the *watch* mechanism to notify a client of ZooKeeper of a change to a *znode* (*i.e.*, a ZooKeeper data object containing its path and data content). This feature is exploited for dynamic changes in the system.

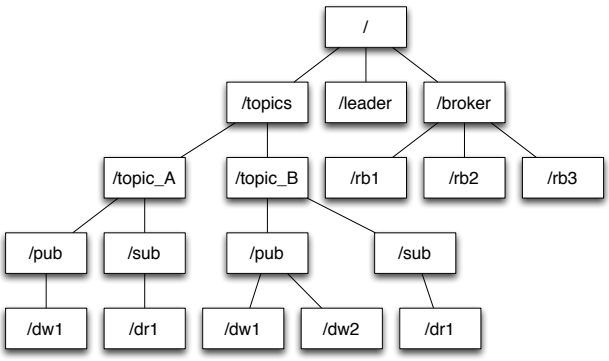


Figure 2: Tunneling used in PubSubCoord for ZNode Tree

Consequences and resolution: The data model of Zoo-

Keeper is structured like a file system in the form of znodes. The original intent of this hierarchical namespace is to manage group membership. We repurposed it to manage pub/-sub endpoints by applying the Tunnel pattern to introduce pub/sub semantics into the znodes. Figure 2 shows the znode data tree structure of PubSubCoord with pub/sub semantics. The root znode contains three child znodes: *topics*, *leader*, and *broker*. All unique topics defined in the pub/sub system are rooted under the *topics* znode. The child znodes for every unique topic represent the endpoints, i.e., publishers and subscribers, associated with it. The *leader* znode is used to elect a leader among the routing brokers. The *broker* znode has child znodes for each routing broker where its location information (*i.e.*, IP address and port number of a routing broker) is stored. The leader uses this information to associate a selected routing broker's location to a topic znode after the topic assignment.

Dynamic updates to different parts of this tree are achieved through broker interactions that exploit the watch mechanism. Specifically, dynamic changes in the system (*e.g.*, broker or pub/sub join/leave, topic creation/deletion) are handled using the Observer pattern where brokers connect to the ZooKeeper service as its clients and create, update, and delete znodes stored in the servers. They use the watch mechanism to set watches on interesting znodes to receive notifications. PubSubCoord exploits the *ephemeral* mode feature of ZooKeeper, where a specific znode in the tree is automatically deleted from the tree when the client session handling this znode is lost. Details on all the interactions that take place in this context are provided in Section 2.5 where we show how brokers discover each other and routes are established.

2.4 Addressing Overload and Fault Tolerance Requirements

Context: Performance of the WAN-scale pub/sub system in our architecture can be impacted by at least two factors: load on a routing broker and network congestion on the two hop route between edge brokers over the broker overlay.[3] Load on a routing broker depends on the number of topics it manages and correspondingly the number of edge brokers it interconnects through itself. Addressing these two sources of performance bottleneck are important for PubSubCoord. In the case of faults, although many kinds of faults are possible, failures of routing brokers and the coordination logic are most critical. Hence, we focus only on tolerating failures in routing brokers and the coordination logic.

2.4.1 Routing Broker Overload Management

Design: We address the routing broker overload problem by supporting load balancing within the routing broker tier. Load balancing is handled by a leader routing broker, which is elected among the routing brokers. To elect a leader in a consistent and safe manner, PubSubCoord uses ZooKeeper's *leader* znode for routing brokers to write themselves on the znode so as to be elected as a leader (*i.e.*, voting process). The routing broker that gets to write first becomes a leader since the znode is locked thereafter (*i.e.*, no one can write on the znode unless the leader fails). The rest of the routing brokers become workers. Worker routing brokers relay pub-/sub data between edge brokers. The leader routing broker can also serve as a worker routing broker.

Consequences and resolution: A leader routing broker must manage the cluster of routing brokers and assign topics to workers in a way that balances the load. It does this by selecting the least loaded worker, which currently is decided based on the number of adopted topics by that worker. However, the use of the Strategy pattern enables us to plug in other load balancing schemes (*e.g.*, least loaded based on CPU utilization or the number of connections).

2.4.2 Broker Fault Tolerance

Design: PubSubCoord offers tolerance to routing broker failures, which can be of two kinds: worker failure and leader failure. When the worker fails, the leader reassigns topics handled by that failed broker to another worker routing broker to avoid service cessation. If the load is too high, the cloud will elastically scale the number of routing brokers. If a leader fails, the routing brokers vote for another leader again. On (re)assignment or failure of routing broker, PubSubCoord leverages ZooKeeper's watch mechanism to notify the appropriate edge brokers to update their paths to the right routing broker.

Consequences and resolution: Since the ZooKeeper server itself may fail, to provide a scalable and fault-tolerant service at the coordination layer, multiple ZooKeeper servers can exist as an ensemble, and a leader of the ensemble synchronizes data between distributed servers to provide consistent coordination events to clients (*i.e.*, brokers in our solution) and avoid single points of failure.

Note that we do not offer a solution to edge broker failure. We treat this failure as making an isolated network unreachable and hence not part of the pub/sub system. If and when it reappears, the edge broker will follow the protocol for informing PubSubCoord of its existence as described next in the runtime interactions.

2.4.3 Deadline-aware Overlay Optimizations

A second cause of performance bottlenecks stems from the congested two-hop route connecting edge brokers via a routing broker. To overcome this problem, PubSubCoord also supports an optimization to both improve reliability and latency by providing an additional one hop path over the overlay that directly connects communicating edge brokers. Note that doing this for every edge broker is infeasible due to the very large connection state that every edge broker must manage. Figure 3 illustrates the concept. These optimizations can be leveraged by pub/sub flows that require stringent assurances on reliable and deadline-driven data delivery.

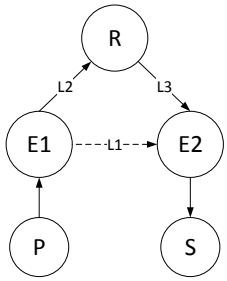


Figure 3: Multi-path Deadline-aware Overlay Concept

[3]We have not addressed security issues in this paper.

To achieve this feature, PubSubCoord requires hints from the underlying pub/sub messaging system. As an example, OMG DDS uses *deadline* QoS as a contract between pub/-sub flows, which is used to express the maximum duration of a sample to be updated. For those event streams requiring strict deadlines, multi-path overlay networks build an alternative, additional path directly between edge brokers thereby reducing the number of hops to just one.

The patterns-based framework design of PubSubCoord enables strategizing it with the underlying technology-specific optimizations. This technology-specific logic should also provide the threshold on when to activate these optimizations.

2.5 Broker Interactions and Operation

So far we presented the architectural elements of PubSubCoord. We now present details on the runtime interactions among these elements that realizes the various capabilities of PubSubCoord. We describe how the brokers interact and the actual process of updating their internal states used in routing the streamed pub/sub data.

2.5.1 Routing Broker Responsibilities

Figure 4 presents the sequence diagram showing the runtime interactions of the routing brokers. The algorithm executed by the routing broker is captured in Algorithm 1. This algorithm is predominantly event-driven, i.e., it is made up of callback functions that are invoked when some condition is satisfied. These callback functions are invoked by ZooKeeper due to the different watch conditions. The *Routing Service* shown in the figure and used in the algorithm are the capabilities at the edge broker that bridge the isolated network to the outside world.

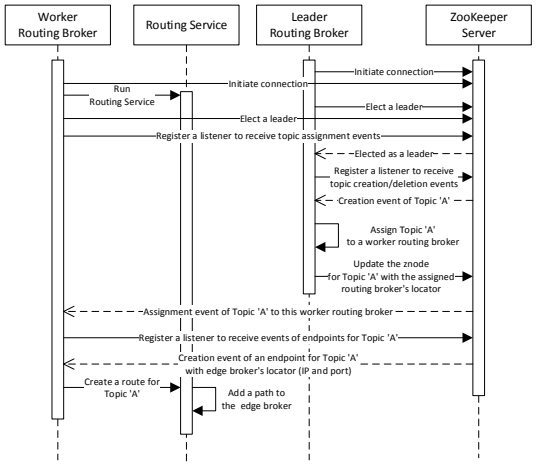


Figure 4: Routing Broker Sequence Diagram

The algorithm and sequence of steps for the routing broker operate as follows: Each routing broker initially connects to the ZooKeeper leader as a client of the ZooKeeper service. The cluster of routing brokers subsequently elect a leader among themselves using the process described in Section 2.3 that uses the *leader* znode. Once a leader is elected, it registers a listener (*i.e.*, event detector that is notified when the registered znode changes) on the *topics* znode (shown in Figure 2) to receive topic relevant events (*e.g.*, creation or deletion of topics).

Algorithm 1 Routing Broker Callback Functions

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

The following callback functions are implemented by the routing brokers:

- BROKER NODE LISTENER – This function is invoked when a znode for a worker routing broker is updated with an assigned topic by a leader routing broker and activated by a ZooKeeper client process.
- ENDPOINT LISTENER – This function is invoked when children pub/sub endpoints of a znode for an assigned topic are created, deleted, or updated. It is activated by a ZooKeeper client process.

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

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

Consider a concrete example. Using Figure 4, when *TopicA* is created, the leader routing broker assigns the topic to the least loaded worker, which currently is decided based on the number of adopted topics by that worker. However, other

strategies can also be used in the load balancing decisions (*e.g.*, least loaded based on CPU utilization or the number of connections). Next, the leader updates a locator of the assigned worker broker on the corresponding znode that is created for *TopicA*, *i.e.*, a child of *topics* znode – see the left-most node in row three of Figure 2. This locator information will then be used by edge brokers interested in *TopicA*.

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

2.5.2 Edge Broker Responsibilities

Figure 5 shows the corresponding sequence diagram for edge brokers, and Algorithm 2 describes the logic of the callback functions implemented by the edge broker.

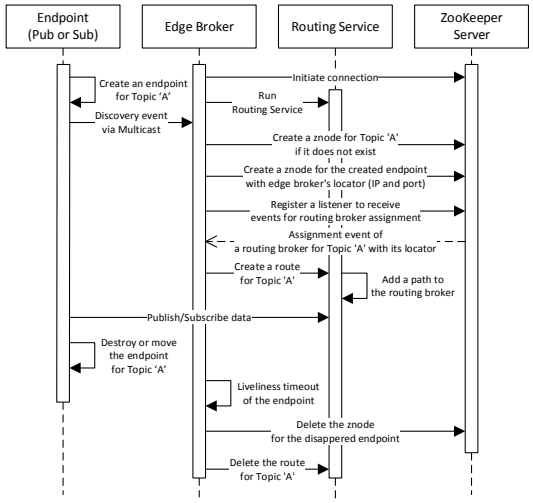


Figure 5: Edge Broker Sequence Diagram

Recall that an edge broker interfaces the underlying concrete pub/sub technology with the generic capabilities provided by PubSubCoord by tunneling technology-specific metadata through the generic data structures and communication mechanisms of PubSubCoord. To better explain the operation of the edge broker, we have used the OMG DDS as our concrete pub/sub technology.[4].

The OMG DDS specification defines a distributed pub-/sub communications standard [20]. At the core of DDS is a data-centric architecture (*i.e.*, subscriptions are defined by topics, keyed data types, data contents, and QoS policies) for connecting anonymous data publishers with data subscribers in a logical global data space. A DDS data publisher produces typed data streams identified by names called *topics*. The coupling between a publisher and subscriber is expressed in terms of topic name, its data type schema, and QoS attributes of publishers and subscribers. To ease the management, each publisher is made up of one or more DataWriters and each subscriber is made up of one or more DataReaders. Each DataWriter and DataReader can be associated with only one topic and perform the action of writing and reading, respectively.

[4]Our empirical evaluations also use OMG DDS as the concrete pub/sub technology as described in Section 3

Algorithm 2 Edge Broker Callback Functions

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

The algorithm implements the following callback functions for edge brokers which are invoked under the following conditions:

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

The edge broker operates as follows: Like routing brokers, edge brokers initially connect to the ZooKeeper servers as clients of the ZooKeeper service. In the context of OMG DDS, edge brokers make use of *built-in entities* (*i.e.*, special pub/sub entities for discovering peers and endpoints in a network supported by OMG DDS) to discover endpoints in local networks. For example, when a pub or sub endpoint interested in *TopicA* is created within some isolated network, the built-in entities receive discovery events via multicast, and then edge brokers create znodes for the created endpoints.

PubSubCoord does not need to know the semantics of these technology-specific mechanisms; all technology-specific

information is masked within the generic znode data structures. When an endpoint is created with a new topic, an edge broker informs ZooKeeper of the new topic which inserts it into its znode tree and informs the leader routing broker of the new topic. The routing broker leader then selects one of the existing worker routing brokers to handle that topic as explained before. This selection is made based on the load on each worker routing broker.

Edge brokers register a listener on a topic znode (*e.g.*, *topic_A* in Figure 2) in which the created endpoint is interested in to obtain the locator of the routing broker that is in charge of that particular topic. Once a locator of a routing broker is obtained, an edge broker initiates a data dissemination path to the routing broker through the routing capabilities that are assumed to be collocated with the edge broker.

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

In the TOPIC NODE LISTENER callback function, each topic znode stores the locator of the worker routing broker that is responsible for the topic. The locator of a routing broker is added to the routing capability of the edge broker to establish a communication path between the edge broker and a worker routing broker.

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

To support mobility or termination of endpoints, PubSubCoord relies on some cooperation from the underlying pub/sub technology. For example, in the context of OMG DDS, if the created endpoints move to different networks or are deleted, a timeout event occurs by virtue of using the *liveliness* QoS policy (*i.e.*, a DDS QoS policy used to detect disconnected endpoints where the timeout values are configurable) and accordingly the znodes (which operate in the ephemeral mode) for those endpoints are deleted from the coordination servers and the route maintained at the edge broker is also terminated.

2.6 Implementation Details

Recall that PubSubCoord is meant to work with any underlying pub/sub technology with a goal of making it WAN-enabled. Any concrete realization of PubSubCoord will require some concrete underlying pub/sub technology. We have demonstrated the feasibility of PubSubCoord in the context of OMG Data Distribution Service (DDS). We have used Curator,[5] which is a high-level API that simplifies using ZooKeeper, and provides useful recipes such as leader election and caches of znodes. We use the cache recipe to locally reserve data objects that are accessed multiple times for fast data access and reduce the load on ZooKeeper servers.

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

3. EXPERIMENTAL VALIDATION OF PUBSUBCOORD

We now present results of experiments we conducted to validate our claims. We focus on PubSubCoord performance and scalability, and overhead of the coordination layer.

3.1 Overview of Testbed Configuration and Default Settings

Our testbed is a private cloud managed by OpenStack comprising 60 physical machines each with 12 cores and 32 GB of memory. We emulated a WAN environment in our testbed using Neutron,[6] which is an OpenStack project for networking as a service that allows users to create virtual networks by using an Open vSwitch plugin.[7] To that end, we created a total of 120 virtual networks. 380 virtual machines (VMs) are placed across these virtual networks. Each VM is configured with one virtual CPU and 2 GB of memory. We use RTI Connext 5.1 [8] as the implementation of the OMG DDS standard technology, and its Routing Service is integrated with brokers for our test applications.

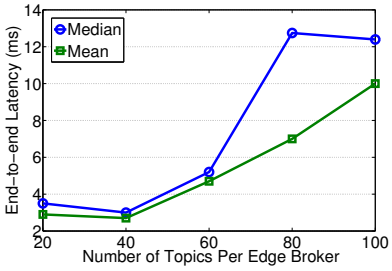
We used all the 380 VMs for our scalability experiments discussed below. Each broker executes inside its own VM for which we used a total of 160 VMs: 120 VMs for hosting edge brokers (implying 120 isolated networks) and 40 VMs for routing brokers (implying a cluster of 40 routing brokers). Of the remaining 220 VMs, 20 VMs are used for hosting publishers and 200 VMs for subscribers. Each of these VMs runs either 25 publisher or 50 subscriber test applications. We locate 50 publishers or 100 subscribers within each isolated network (*i.e.*, 2 VMs inside each network). This approach keeps the experimental apparatus simple. The entire number of publishers and subscribers is 1,000 and 10,000, respectively. Subscribers in each network are interested in 100 topics out of the total 1,000 topics in a system. Publishers push data every 50 milliseconds, and the size of a data sample is 64 bytes; any other size is acceptable, however, we did not test with simultaneous different packet sizes.

3.2 OMG DDS QoS Policy Settings Used

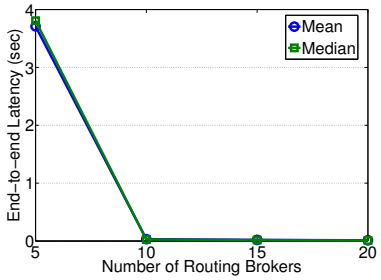
Since OMG DDS supports a number of different QoS policies that can be mixed and matched, we used some of these that are important for the systems of interest to us. Each QoS policy has offered and requested semantics (*i.e.*, offered by publishers and requested by subscribers) and are used in conjunction with data types of topics to match pairs of endpoints, *i.e.*, the DataReader and DataWriter. We describe the purpose of only those policies that we have used in our experiments and show how they are used.

The ***reliability*** QoS controls the reliability of data flows between DataWriters and DataReaders at the transport level. It can be of two kinds: BEST_EFFORT and RELIABLE. The ***durability*** QoS specifies whether or not the DDS middleware stores and delivers previously published data samples to endpoints that join the network later. The reliability and persistency can be affected by the ***history*** QoS policy,

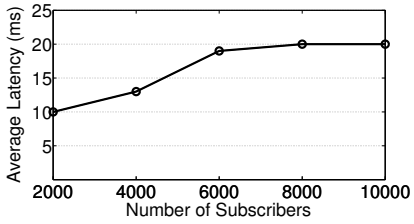
[6] `https://wiki.openstack.org/wiki/Neutron`
[7] `http://www.openvswitch.org`
[8] `https://www.rti.com/products/dds/`



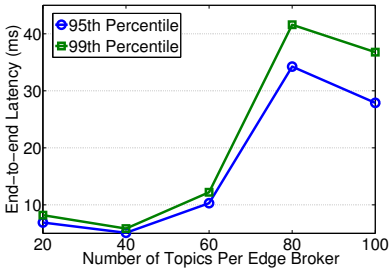
(a) Mean and Median End-to-end Latency of Pub/Sub by Different Number of Topics Per Network



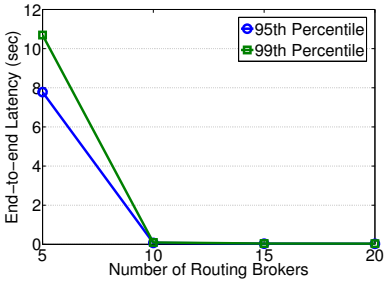
(b) Mean and Median End-to-end Latency of Pub/Sub with Load Balance in Routing Brokers



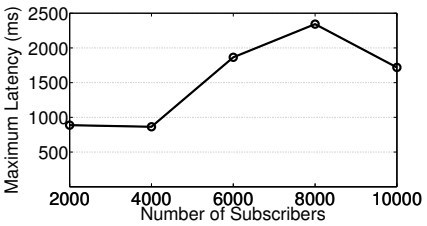
(c) Average Latency of Coordination Service by Different Number of Joining Subscribers



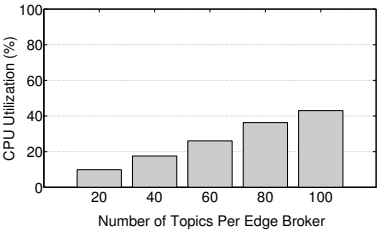
(d) 95th and 99th Percentile End-to-end Latency of Pub/Sub by Different Number of Topics Per Network



(e) 95th and 99th Percentile End-to-end Latency of Pub/Sub with Load Balance in Routing Brokers

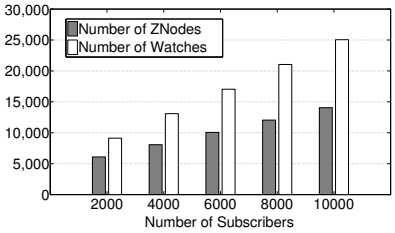


(f) Maximum Latency of Coordination Service by Different Number of Joining Subscribers



(g) CPU Utilization by Different Number of Topics Per Network

(h) CPU Utilization with Load Balance in Routing Brokers



(i) Number of ZNodes and Watches by Different Number of Joining Subscribers

Figure 6: Results of Performance and Scalability Experiments

which specifies how much data must be stored in in-memory cache allocated by the middleware. The ***lifespan*** QoS helps to control memory usage and lifecycle of data by setting the expiration time of the data on DataWriters, so that the middleware can delete expired data from the cache. The ***deadline*** QoS policy specifies the deadline between two successive updates for each data sample. The middleware will notify the application via callbacks if a DataReader or a DataWriter breaks the deadline contract. Note that DDS makes no effort to meet the deadline; it only notifies if the deadline is missed.

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

3.3 Testing Methodology

To evaluate our solution, the system performance is measured in terms of the end-to-end latency from publishers to subscribers, while scalability of data dissemination is measured in terms of CPU usage on brokers. CPU usage is shown along with latency to understand how different configuration settings, *i.e.*, number of topics per network and number of routing brokers, affect dissemination scalability.

Moreover, we measure latency of coordination requests and the number of data objects and notifications on ZooKeeper servers to show coordination scalability. To measure end-to-end latency from publishers to subscribers, we calculate time differences with timestamps of events on publishers and subscribers. For clock synchronization, we exploit the Precise Time Protocol (PTP) [5] that guarantees fine-grained time synchronization for distributed machines, and achieves clock accuracy in the sub-microsecond range on a local network. Due to space limitations, we have not presented results on system behavior and performance due to dynamic churn in the different aspects of the system.

3.4 Performance and Scalability Results

For end-to-end latency of measurements, we collect latency values of 5,000 samples in total for each subscriber and use values only after 1,000 samples since the latency values of the initial samples are not consistent due to coordination and discovery process overhead until the system stabilizes (*e.g.*, time for discovery of brokers and creating routes). Figure 6 summarizes all the results that are explained below.

3.4.1 Evaluating the Broker Overlay Layer

Since the edge brokers are responsible for delivering data incoming from other edge brokers via worker routing brokers to subscribers in a local, isolated network, the computation overhead on edge brokers grows linearly as the number of adopted topics increases. Figure 6a, 6d, and 6g show results with different number of topics per edge broker, increasing the number of topics from 20 to 100 out of 1,000 topics in a system. The CPU utilization linearly increases according to the number of adopted topics, and latency values grow as well. From these results, we can infer that if the number of incoming streams increases due to more number of topics per network, it adversely affects latency values even though the CPUs of edge brokers are not saturated.

Our solution supports load balancing in the group of routing brokers and makes it possible to flexibly scale systems with the number of topics. Figure 6b, 6e, and 6h present latency and CPU usage across different number of routing brokers. When the number of routing brokers is small, in this case 5, the CPU of the routing brokers become saturated and latency values are adversely impacted. However, after increasing the number of routing brokers to 10, latency values improve. The results in Figure 6h also validate that CPU usage linearly decreases by increasing the number of routing brokers.

3.4.2 Evaluating the Coordination Layer

We evaluate the scalability of a ZooKeeper-based centralized coordination service by increasing the number of simultaneous joining subscribers. Figures 6c and 6f show latency, *i.e.*, the amount of time it takes for the server to respond to a client request. Figure 6i presents the number of used znodes and watches. We use *mntr*, a ZooKeeper command for monitoring service,[9] to retrieve the experimental values presented in our results. We increase the number of subscribers from 2,000 to 10,000 in steps of 2,000. The average latency increases from 10 milliseconds to 20 milliseconds and the number of znodes and watches linearly increase approximately 2,000 and 4,000, respectively, as the number of subscribers increase. The reason why the number of watches are twice compared to the number of znodes is that ZooKeeper needs to notify brokers for both publishers and subscribers if they have matching pub/sub endpoints.

3.5 Evaluating Performance Optimizations for Deadline-aware Overlays

We also conducted experiments to validate our deadline-aware overlays showing latency and overhead by comparing the performance parameters for multi-path and single-path overlays. A topology used for these experiments is shown in Figure 3. To emulate variable delays in the network and packet losses, which are common in WANs, we use Dummynet [23]. These parameters are varied depending on geographic locations of brokers, which is a factor influencing the need for deadline-aware overlays.

To ensure realistic network delays and losses, for the multi-path overlay experiments, we use delay and loss data provided by Verizon, which shows latency and packet delivery statistics for communication between different countries across the globe.[10] We categorize delay and loss data into two groups (*i.e.*, A with 30ms delay and no packet loss, and B with 250 msec delay and 1% packet loss in Table 1) and experimented 8 possible combinations with the given links (*i.e.*, L1, L2, and L3 as shown in Figure 3), and test cases described in Table 1.

Table 1: Deadline-aware Overlays Experiment Cases

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

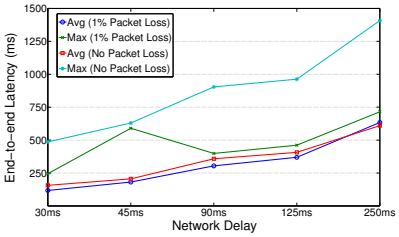
A = 30ms delay, no packet loss
B = 250ms delay, 1% packet loss

Figure 7a and 7b show average and maximum latency of single-path overlays with different network delays and packet loss and multi-path overlays with 8 test cases, respectively. From Case 1 to Case 5, the multi-path overlays perform better than any cases of single-path in terms of latency. All cases of multi-path overlays outperform a case with 125 milliseconds delay and 1% packet loss in single-path overlays. In spite of that, a multi-path overlay builds a duplicate path from an edge broker other than from a routing broker, so it causes extra overhead compared to a single-path overlay due to additional computations and extra network transfer at the edge broker. We measure network transfer overhead for 10,000 samples from a publisher to a subscriber to compare single-path and multi-path by using *tcpdump*[11] and the results are presented in Figure 7c. These results validate that deadline-aware overlay improves latency, but incurs some overhead.

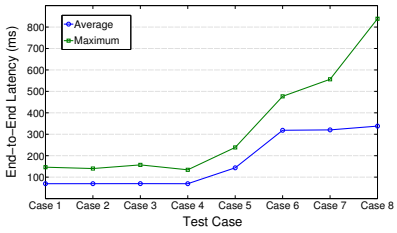
[9] `http://zookeeper.apache.org/doc/trunk/zookeeperAdmin.html`

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

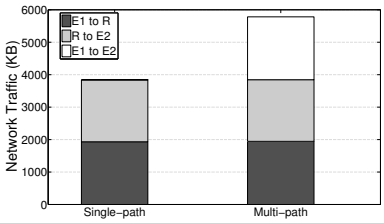
[11] `http://www.tcpdump.org`



(a) End-to-end Latency of Pub/Sub with Single-path Overlays



(b) End-to-end Latency of Pub/Sub Multi-path Overlays



(c) Overhead Comparison

Figure 7: Deadline-aware Experiments

3.6 Discussion and Key Insights

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

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

- **Centralized coordination services like ZooKeeper can serve as a pub/sub control plane for large-scale systems.** Our solution employs a centralized service for coordinating pub/sub brokers for its consistency and simplicity, and our experimental results show that the average latency of the coordination service is 20 milliseconds for 10,000 subscribers joining simultaneously. Moreover, the number of data nodes and notifications linearly increase by organizing its data tree in a hierarchical way. Our experiments use a standalone server for coordination, but multiple servers as an ensemble can be used for scalability and fault-tolerance and ZooKeeper guarantees consistency of data between multiple servers. The ensemble of servers is more scalable for read operations, but not for write operations that require synchronizing data between servers. In future, we plan to carry out experiments with increasing the number of coordination servers to understand its scalability for pub/sub broker coordination.

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

- **End-to-end QoS management is required for efficiency.** Although most of the QoS policies in our solution are supported by hop-by-hop enforcement between brokers, QoS policies for persistence, reliability, and ordering used in our experiments assure end-to-end QoS. However, this may be inefficient for some cases. For example, the *durability* QoS ensures sending previously published data to late joining subscribers. To support end-to-end data persistence with hop-by-hop QoS enforcement, each broker needs to keep history data in memory that will not be freed until it is acknowledged. This is beneficial for some late joining subscribers that require history data with low-latency. However, keeping duplicate history data on each broker consumes memory resources. We suggest end-to-end acknowledgment mechanisms as a solution to address these inefficiencies.

4. RELATED WORK

Prior research on pub/sub systems can be classified into topic-based, attribute-based, and content-based depending on the subscription model. The topic-based model, such as Scribe [24], TIB/RV [19], and SpiderCast [8], groups subscription events in topics. In the attribute-based model, events are defined by specific types, and therefore this model helps developers to define data models in a robust way by type-checking. The content-based model [7, 22] allows subscribers to express their interests by specifying conditions

on the data content of events, and the system filters out and delivers events based on the conditions.

The Object Management Group (OMG)'s Data Distribution Service (DDS) [20] standard for data-centric pub/sub holds substantial promise for CPS because of its support for configurable QoS policies, dynamic discovery, and asynchronous and anonymous decoupling of data endpoints (*i.e.*, publishers and subscribers) in time and space. However, DDS is limited in its support for WAN-based CPS. For instance, DDS uses multicast as a default transport to dynamically discover peers in a system. If the endpoints are located in isolated networks that do not support multicast, then these endpoints cannot be discovered by each other. Secondly, even if these endpoints were discoverable, because of network firewalls and network address translation (NAT), peers may not be able to deliver messages to the destination endpoints. PubSubCoord addresses these limitations and makes it possible for OMG DDS to be used in WAN-scale CPS without any modifications to the OMG DDS semantics.

Pub/sub systems tend to form overlay networks to support application-level multicast rather than using IP-based multicast owing to the fact that IP multicast is not supported in WANs and the limited number of IP-based multicast addresses would not fit the potential number of logical channels for fine-grained subscription models [2]. Overlay architectures for pub/sub systems can be categorized into broker-based overlay [7, 22, 19], structured peer-to-peer [24], and unstructured peer-to-peer. GREEN [25] supports configurable overlay architectures for different network environments. PubSubCoord adopts a hybrid approach that constructs unstructured peer-to-peer overlays in LANs by dynamically discovering peers via multicast, and broker-based overlays in WANs.

The BlueDove [16] pub/sub system achieves scalability and elasticity by harnessing cloud resources. It is a two-tier architecture to reduce the number of delivery hops and for simplicity. BlueDove is designed for enterprise systems deployed in the cloud and does not consider the restrictions of physical locations of pub/sub endpoints. Like BlueDove, PubSubCoord also uses a hierarchical broker solution and exploits the cloud for scalability. However, in our system, pub/sub endpoints located in different networks dynamically discover each other with the help of edge brokers, and therefore we consider physical restrictions of pub/sub endpoints. Moreover, our approach is decoupled from any specific pub-/sub technology.

Bellavista et al. [3] study QoS-aware pub/sub systems over WANs and compare multiple existing pub/sub systems supporting QoS including DDS. In [15], the authors evaluate a pub/sub system for wide-area networks named Harmony and techniques for responsive and highly available messaging. The Harmony system delivers messages through broker overlays placed in different physical networks, and pub/sub endpoints communicate via local brokers located in the same network. Although this effort describes a WAN-scale pub/-sub solution with QoS support, it centers on selective routing strategies to balance responsiveness and resource usage using multi-hop broker networks. In contrast, PubSubCoord uses only a two-hop overlay architecture connecting the edge and routing brokers.

IndiQoS [6] also proposes a pub/sub system with QoS support to reduce end-to-end latency by exploiting network-level reservation mechanisms, where message brokers are structured using distributed hash table (DHT). Similar to IndiQoS, we pursue low-latency and high availability but our solution can also support other QoS policies such as configurable transport reliability, data persistence, ordering, and resource management by controlling the depth of history data and subscribing rate. In other words, we strive to provide the capabilities of the underlying pub/sub technology at the WAN level and additionally also provide WAN-based optimizations. We do not use a DHT solution for brokers and so a comparison along these lines will require additional research, which is part of our future work.

Recent research including ours [10, 17] has broadened the scope of OMG DDS to WANs by bringing in routing engines to disseminate data from a local network to others. Our solution emphasizes reuse and hence leverages such routing engines and additionally solves the discovery and coordination problem between routing engines that otherwise requires significant manual efforts for large-scale systems. Finally, [28] suggests separation of control and data plane in next generation pub/sub systems, which is motivated by software-defined networking (SDN). We have not explored the benefits of SDN for PubSubCoord, however, our other ongoing efforts have demonstrated preliminary ideas [11, 21], which form additional dimensions of future work.

Literature on stream processing is not reviewed since we feel the comparisons are not accurate due to the differences in the computation and communication semantics.

5. CONCLUDING REMARKS

Emerging WAN-scale distributed systems found in domains, such as transportation and smart grid, must disseminate large volumes of data between a large number of heterogeneous entities that are geographically distributed, and require a variety of QoS properties for data dissemination from the publishers of information to the subscribers. Many disparate solutions that handle individual aspects of the problem space exist but seldom have these techniques been brought together holistically to solve the broader set of challenges. There is a need to systematically integrate these proven solutions while also providing new capabilities. This is challenging, particularly when the sum of the parts itself must be made reusable and applicable across a variety of pub/sub technologies.

To address some of these broader challenges, this paper presents the design, implementation, and evaluation of PubSubCoord, which is a cloud-based coordination and discovery service and middleware for geographically dispersed pub-/sub applications. PubSubCoord supports scalability in terms of data dissemination as well as coordination, dynamic discovery, and configurable QoS properties. Our experimental results validate our claims.

Insights gained from this research and current limitations have provided us directions for future work. For example, we have not fully explored the fault tolerance and security dimensions in current work. Similarly, the edge broker bottleneck remains to be resolved. Furthermore, our work uses overlay networks; thus we do not have control over the QoS over the network links. It is possible to use software defined networking (SDN) to control the network QoS for pub/sub traffic and provide differential services to different pub/sub flows. A recent effort [26] has already showed how pub/-sub and SDN can be combined to perform filtering at the level of SDN controllers. Since the systems of interest to us

will almost always comprise heterogeneous technologies, we need to investigate how the existing architecture can support WAN-scale pub/sub across distributed isolated networks using heterogeneous pub/sub technologies (e.g., between DDS and MQTT). Our future work aims to address these limitations.

PubSubCoord is designed with reuse in mind and can easily be adopted in industrial settings. To that end, the middleware and test harness can be downloaded from: `www.dre.vanderbilt.edu/~kyoungho/pubsubcoord`.

6. REFERENCES

[1] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to siena. *The Computer Journal*, 50(4):444–459, 2007.

[2] R. Baldoni, M. Contenti, and A. Virgillito. The evolution of publish/subscribe communication systems. In *Future directions in distributed computing*, pages 137–141. Springer, 2003.

[3] P. Bellavista, A. Corradi, and A. Reale. Quality of service in wide scale publish-subscribe systems. *IEEE Communications Surveys & Tutorials*, 2014.

[4] F. Cao and J. P. Singh. Efficient event routing in content-based publish-subscribe service networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 929–940. IEEE, 2004.

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

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

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

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

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

[10] A. Hakiri, P. Berthou, A. Gokhale, D. Schmidt, and T. Gayraud. Supporting End-to-end Scalability and Real-time Event Dissemination in the OMG Data Distribution Service over Wide Area Networks. *Elsevier Journal of Systems Software (JSS)*, 86(10):2574–2593, Oct. 2013.

[11] A. Hakiri, P. Berthou, P. Patil, and A. Gokhale. Towards a Publish/Subscribe-based Open Policy Framework for Proactive Overlay Software Defined Networking. Technical Report ISIS-15-115, Institute for Software Integrated Systems, Nashville, TN, USA, 2015.

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

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

[14] M. A. Jaeger, H. Parzyjegla, G. Mühl, and K. Herrmann. Self-organizing broker topologies for publish/subscribe systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 543–550. ACM, 2007.

[15] M. Kim, K. Karenos, F. Ye, J. Reason, H. Lei, and K. Shagin. Efficacy of techniques for responsiveness in a wide-area publish/subscribe system. In *Proceedings of the 11th International Middleware Conference Industrial track*, pages 40–45. ACM, 2010.

[16] M. Li, F. Ye, M. Kim, H. Chen, and H. Lei. A scalable and elastic publish/subscribe service. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1254–1265. IEEE, 2011.

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

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

[19] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 58–68. ACM, 1994.

[20] OMG. The data distribution service specification, v1.2. `http://www.omg.org/spec/DDS/1.2`, 2007.

[21] P. Patil, A. Hakiri, and A. Gokhale. Bootstrapping Software Defined Network for Flexible and Dynamic Control Plane Management. In *1st IEEE Conference on Network Softwarization*, London, UK, Apr. 2015. IEEE.

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

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

[24] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Networked group communication*, pages 30–43. Springer, 2001.

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

[26] M. A. Tariq, B. Koldehofe, S. Bhowmik, and K. Rothermel. PLEROMA: A SDN-based High Performance Publish/Subscribe Middleware. In *Proceedings of the 15th International Middleware*

Conference, Middleware '14, pages 217–228, New York, NY, USA, 2014. ACM.

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

[28] K. Zhang and H.-A. Jacobsen. Sdn-like: The next generation of pub/sub. *arXiv preprint arXiv:1308.0056*, 2013.