A Coordination and Discovery Service for QoS-enabled Data-Centric Publish/Subscribe in Wide Area Networks

Kyoungho An and Aniruddha Gokhale
ISIS, Dept of EECS
Vanderbilt University
Nashville, TN 37235, USA
Email: {kyoungho.an, a.gokhale}@vanderbilt.edu

Takayuki Kuroda
Knowledge Discovery Research Laboratory
NEC Corporation
Kawasaki, Kanagawa, Japan
Email: t-kuroda@ax.jp.nec.com

***Abstract*—A special class of Internet of Things called the Industrial Internet of Things (IIoT) operates in a large, distributed and dynamic environment comprising sensors all the way to large server clusters. A key requirement for IIoT is a scalable messaging service that supports multiple quality of service (QoS) properties, such as timeliness and resilience. Although existing pub/sub standards, such as the Object Management Group (OMG)'s Data Distribution Service (DDS) for data-centric pub/sub, support a range of QoS properties and dynamic discovery of peers, they are effective only in local area networks (LANs). Moreover, they lack an effective coordination and discovery service needed by brokers that can bridge multiple different LANs. To address these limitations, this paper presents *PubSubCoord*, which is a cloud-enabled coordination and discovery service for QoS-enabled data-centric pub/sub for wide area network (WAN) operations. *PuSubCoord* realizes a WAN-scale, low-latency data dissemination architecture by (a) balancing the load using elastic cloud resources, (b) clustering brokers by topics for affinity, and (c) minimizing the number of data delivery hops in the pub/sub overlay. *PubSubCoord*'s coordination mechanism uses ZooKeeper to support dynamic discovery of brokers and pub/sub endpoints located in isolated networks. Empirical results evaluating the performance of *PubSubCoord* are presented for (1) scalability of data dissemination and coordination, and (2) deadline-aware overlays employing configurable QoS to provide low-latency data delivery for topics demanding strict service requirements.**

***Keywords*-Data Distribution Service, Publish/Subscribe, Middleware, Discovery, Coordination, Cloud Computing**

I. Introduction

Emerging paradigms, such as the internet of things (IoT), connect machines and devices in a loosely coupled manner to form intelligent and large-scale systems. The publish/-subscribe (pub/sub) communication paradigm is attractive for these emerging systems since it provides a scalable and decoupled data delivery mechanism between communicating peers. A special class of IoT referred to as industrial IoT (IIoT) [1] found in domains such as transportation, healthcare, manufacturing, and energy requires different quality of service (QoS) properties to be satisfied, such as timeliness, reliability, and security, for their applications that are deployed over wide area networks (WANs). For example, wind farms have different requirements for data analysis depending on the type of analysis (*e.g.*, frequency of data arrival for machine-level analysis is every 40 milliseconds and for plant-level analysis is every one second). The key to successful data analysis relies on how effective is the system in collecting and delivering data across large number of entities at internet scale in a timely, and reliable manner.[1]

Many pub/sub messaging solutions [2], [3], [4] including research efforts [5], [6], [7] exist that can operate in WANs. Some of these even support QoS properties, such as availability [6], [7], configurable reliability [2], durability [3], and timeliness [8], [9]. However, these solutions tend to support only one QoS property at a time and in most cases, support for configurability is lacking. Moreover, dynamic discovery of endpoints, which is a key requirement for IIoT, is often missing in these solutions.

The presence of large amounts of generated data in IIoT motivates the need for data-centric pub/sub with support for configurable, multiple QoS properties. The Object Management Group (OMG)'s Data Distribution Service (DDS) [10] standard for data-centric pub/sub holds substantial promise for IIoT applications because of its support for configurable QoS policies, dynamic discovery, and asynchronous and anonymous decoupling of data endpoints (*i.e.*, publishers and subscribers) in time and space. However, there still remain many unresolved challenges in using DDS in WAN-based IIoT applications. For instance, DDS uses multicast as a default transport to dynamically discover peers in a system. If the endpoints are located in isolated networks that do not support multicast, then these endpoints cannot be discovered by each other. Secondly, even if these endpoints were discoverable, because of network firewalls and network address translation (NAT), peers may not be able to deliver messages to the destination endpoints.

One approach to supporting DDS in WAN-based pub/sub relies on broker-based solutions [11], [12]. It is conceivable to think that these broker-based solutions in conjunction with

[1] https://www.gesoftware.com/sites/default/files/Industrial_Big_Data_Platform.pdf

the data-centric and configurable QoS features provided by DDS can readily make it useful for IIoT. However, this is not the case for the following reasons. IIoT use cases illustrate heterogeneity in the kinds of devices and networks involved, the number and types of data-centric topics of interest that must be managed, and significant number and churn (*i.e.*, joining and leaving) of the endpoints. Thus, a solution for dynamic discovery and QoS-enabled dissemination that can scale to large number of endpoints is desired. Since brokers are necessary to overcome issues with NAT and firewalls, the scalable discovery and dissemination solution desired for IIoT must also provide effective coordination among potentially large number of distributed brokers.

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

- To address the scalability and low latency requirements of data dissemination across WANs, PubSubCoord introduces a two-level broker hierarchy deployed over a pub/sub overlay network, which provides a maximum two-hop dissemination path for data across distributed, isolated networks.
- To achieve dynamic discovery and data routing between brokers, PubSubCoord exploits and extends the ZooKeeper coordination service [13] to synchronize dissemination paths for the dynamic network of brokers and endpoints.
- For those dissemination paths that need both low latency and reliability assurances, PubSubCoord trades off resource usage in favor of deadline-aware overlays that build multiple, redundant paths between brokers.

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

The remainder of this paper is organized as follows: Section II provides background information on the underlying technologies; Section III describes the design and implementation of PubSubCoord; Section IV shows experimental results validating our claims; Section V compares PubSubCoord with related work; and Section VI presents concluding remarks and alludes to future work.

II. Overview of Underlying Technologies

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

A. OMG Data Distribution Service (DDS)

The OMG DDS specification defines a distributed pub-/sub communications standard [10]. 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, as shown in Figure 1.

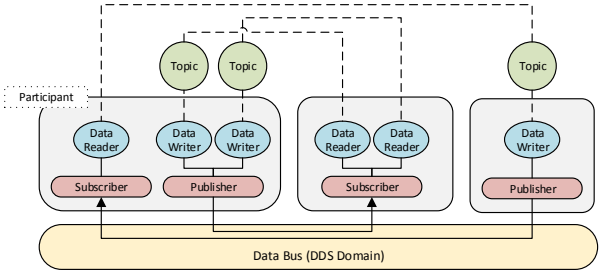


Figure 1: DDS Architecture

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.

A domain is used to logically partition the global data space into groups that are isolated from each other within which the participants, *i.e.*, publishers and subscribers can communicate. 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.

A Topic is a logical channel between DataWriters and DataReaders that specifies the data type of publication and subscription. The topic names, types, and QoS of DataWriters and DataReaders must match for them to communicate with each other.

B. OMG DDS QoS Policies

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

The **reliability** QoS controls the reliability of data flows between DataWriters and DataReaders at the transport level.

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

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

C. DDS Routing Service

Since PubSubCoord relies on a broker-based architecture, we have leveraged and extended an existing DDS broker solution. Specifically, we have used the DDS Routing Service, which is a content-aware bridge service for connecting geographically dispersed DDS systems [11]. It integrates DDS applications across LANs as well as WANs. DDS Routing Service leverages all the entities of DDS and enables DDS applications to publish and subscribe data across domains in multiple networks without any changes to the applications.

D. ZooKeeper

ZooKeeper is a service for coordinating processes within distributed applications [13]. The ZooKeeper service consists of an ensemble of servers that use replication to accomplish high availability with high performance and relaxed consistency. ZooKeeper provides the *watch* mechanism to notify a client of a change to a *znode* (*i.e.*, a ZooKeeper data object containing its path and data content). There exist many coordination recipes using ZooKeeper that are often needed for distributed applications, such as leader election, group membership, and sharing configuration metadata. PubSubCoord exploits these capabilities in its design.

III. Design of PubSubCoord

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

A. PubSubCoord Architecture

Figure 2 shows the PubSubCoord architecture depicting three layers: a coordination layer, a pub/sub overlay layer, and the physical network layer. The pub/sub overlay comprises the two-level broker hierarchy representing the logical network of brokers and endpoints in a system. An edge broker is directly connected to endpoints in a LAN (*i.e.*, an isolated network) to serve as a bridge to other endpoints placed in different networks. A routing broker serves as a mediator to route data between edge brokers according to assigned and matched topics that are present in the global data space. The coordination layer comprises an ensemble of ZooKeeper servers used for coordination between the brokers.

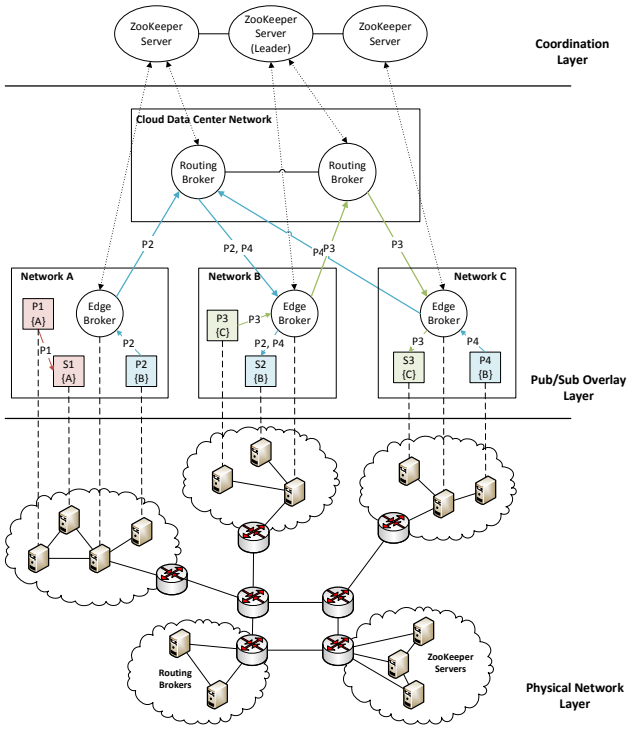


Figure 2: PubSubCoord Architecture

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

network are incurred by data flowing from one isolated network to another (*e.g.*, network A to B).

B. Rationale for PubSubCoord Design Decisions

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

1) 2-level Broker Hierarchy and Scalability: Traditional WAN-based pub/sub systems form an overlay network with brokers to which endpoints can be connected. The brokers exchange subscriptions they receive from subscribers, by which they build a routing path. The published messages are routed to matching subscribers through the routing decision. The main challenge of this approach is how to build states among brokers to route messages towards matching subscribers efficiently. To resolve this challenge, our solution clusters brokers by matching topics and routes topic data through routing brokers responsible for specific topics to minimize the overall number of data exchange and connections between brokers.

In the traditional broker-based pub/sub systems, if a local broker fails, it halts not only a service for endpoints connected to this broker but also service for endpoints connected to other brokers because local brokers are used as intermediate routing brokers. To overcome these limitations, PubSubCoord is structured by harnessing a two-tier architecture similar to the BlueDove system [14].

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

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

2) Need for a Coordination Layer: Although a 2-level broker hierarchy resolves issues with maintaining substantial state, we needed an approach so that the brokers can form this 2-level hierarchy and set the connections between the edge and routing brokers. To that end, PubSubCoord uses a coordination layer comprising an ensemble of ZooKeeper servers, which help brokers discover each other and build broker overlay networks using coordination logic.

The data model of ZooKeeper is structured like a file system in the form of znodes with a simple client API (*e.g.*, only read and write). This hierarchical namespace is actually meant to manage group membership, however, we repurpose it to manage pub/sub endpoints that are grouped by topics. Figure 3 shows znode data tree structure of PubSubCoord stored in ZooKeeper servers. The root znode contains three znodes: *topics*, *leader*, and *broker*. The *topics* znode contains children znodes for every unique topic that has endpoints interested in it, which in turn become the children of the specified topic znode. The *leader* znode is used to elect a leader among routing brokers. The *broker* znode has children znodes for each routing broker where its locator information (*i.e.*, IP address and port number of a routing broker) is stored. The leader uses this information to associate a selected routing broker's locator to a topic znode after the topic assignment.

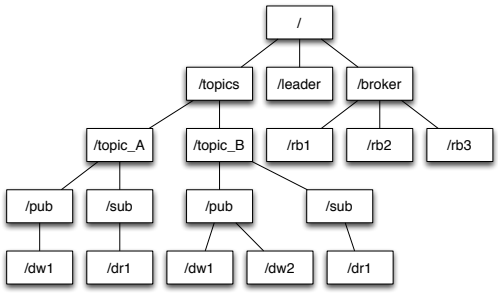


Figure 3: PubSubCoord ZNode Data Tree Structure

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

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

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

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

ZooKeeper notifies the appropriate edge brokers to update their paths to the right routing broker.

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

4) Deadline-aware Overlay Optimizations: PubSubCoord also supports an optimization to both improve reliability and latency by providing an additional one hop path over the overlay that directly connects communicating edge brokers. Figure 4 illustrates the concept. These optimizations can be leveraged by pub/sub streams that require stringent assurances on reliable and deadline-driven data delivery.

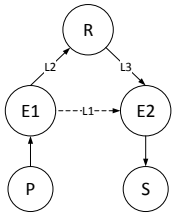


Figure 4: Multi-path Deadline-aware Overlay Concept

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

C. Broker Interactions and Implementation

In this section we describe how the brokers interact and the actual process of updating their internal states used in routing the streamed data. Routing brokers can be divided into two kinds: leader routing broker and worker routing broker. A leader routing broker manages the cluster of routing brokers and assigns topics to workers in a way that balances the load. Worker routing brokers relay pub/sub data between edge brokers. The leader routing broker can also serve as a worker routing broker.

Figure 5 presents the sequence diagram showing the interactions of the routing brokers. Each routing broker initially connects to the ZooKeeper servers as a client. The cluster of routing brokers subsequently elect a leader among themselves. The leader routing broker registers a listener (*i.e.*, event detector that is notified when the registered znode changes) on the *topics* znode (shown in Figure 3) to receive topic relevant events (*e.g.*, creation or deletion of topics). For example, as shown in Figure 5, when $TopicA$ is created, the leader assigns the topic to the least loaded worker, which currently is decided based on the number of adopted topics by that worker. However, other strategies can also be used in the load balancing decisions (*e.g.*, least loaded based on CPU utilization or the number of connections). Next, the leader updates a locator of the assigned worker broker on the corresponding znode that is created for $TopicA$, *i.e.*, a child of *topics* znode – see the leftmost node in row three of Figure 3. This locator information will then be used by edge brokers interested in $TopicA$.

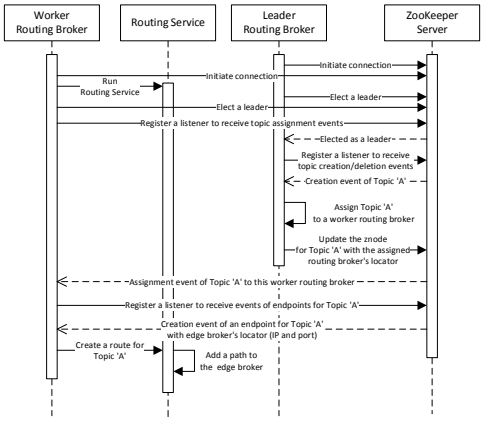


Figure 5: Routing Broker Sequence Diagram

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

Figure 6 shows the corresponding sequence diagram for edge brokers. Like routing brokers, edge brokers initially connect to ZooKeeper servers as clients. Edge brokers make use of *built-in entities* (*i.e.*, special pub/sub entities for discovering peers and endpoints in a network supported by the underlying pub/sub messaging system) to discover endpoints in local networks. For example, when a pub or sub endpoint interested in $TopicA$ is created, built-in entities receive discovery events via multicast, and then edge brokers

create a znode for the created endpoints.

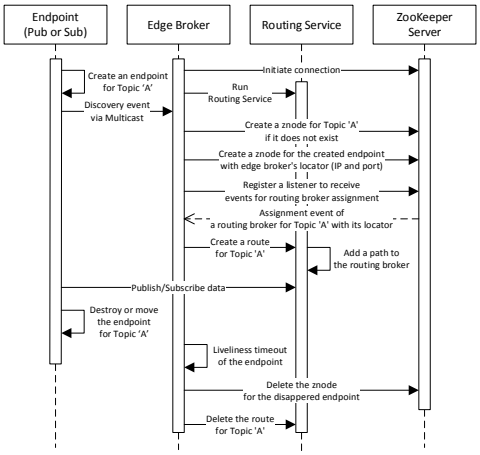


Figure 6: Edge Broker Sequence Diagram

Edge brokers register a listener on a topic znode (*e.g.*, *topic_A* in Figure 3) in which the created endpoint is interested in to obtain the locators for the routing broker that is in charge of that particular topic. Once a locator of a routing broker is obtained, an edge broker initiates a data dissemination path to the routing broker through the Routing Service. If the created endpoints move to different networks or are deleted, a timeout event occurs by virtue of using the *liveliness* QoS (*i.e.*, it is used to detect disconnected endpoints where the timeout values are configurable) and accordingly the znodes for endpoints are deleted from coordination servers and a route created in Routing Service is also terminated. Thus, mobility of publisher and subscriber endpoints is also supported by the PubSubCoord design.

IV. Experimental Validation of PubSubCoord

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

A. Overview of Testbed Configurations and Testing Methodology

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

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

To evaluate our solution, we measure end-to-end latency from publishers to subscriber, and CPU usage on brokers for scalability of data dissemination. CPU usage is shown along with latency to understand how different settings, *i.e.*, number of topics per network and number of routing brokers, affect dissemination scalability. Moreover, we measure latency of coordination requests and the number of data objects and notifications on ZooKeeper servers to show coordination scalability. To measure end-to-end latency from publishers to subscribers, we calculate time differences with timestamps of events on publishers and subscribers. Because publishers and subscribers run in different machines, we exploit the Precise Time Protocol (PTP) [15] that guarantees fine-grained time synchronization for distributed machines, and achieves clock accuracy in the sub-microsecond range on a local network.

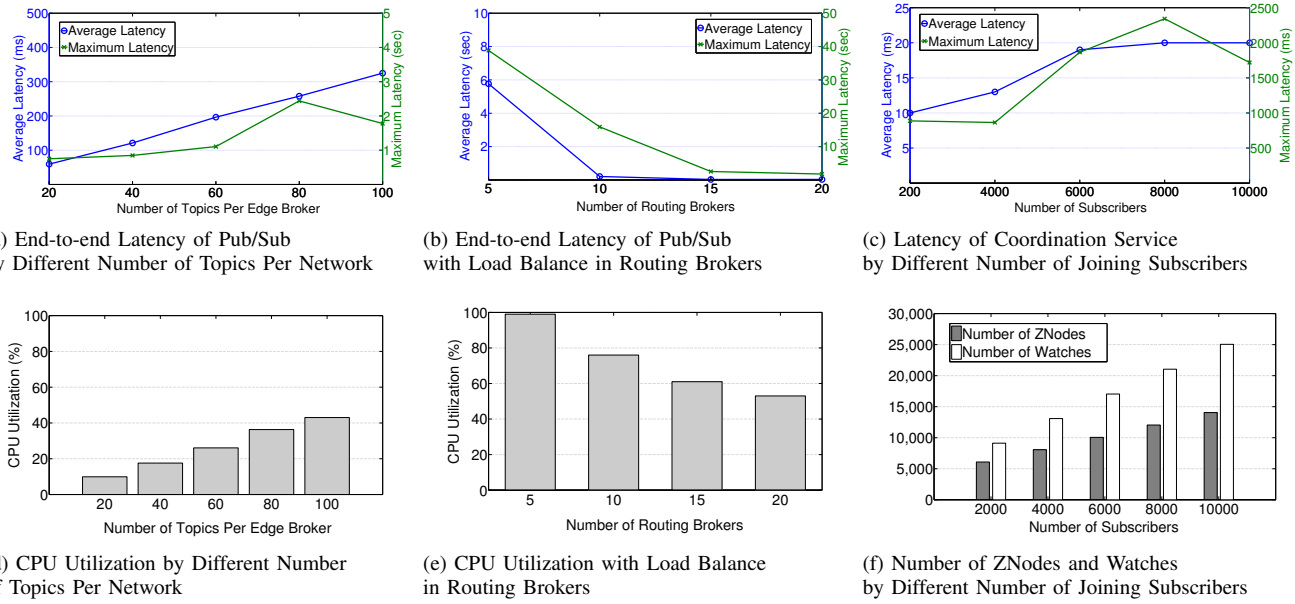
B. Scalability Results

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

For end-to-end latency of measurements, we collect latency values of 5,000 samples in total for each subscriber and use values only after 1,000 samples since the latency

[2]https://wiki.openstack.org/wiki/Neutron
[3]http://www.openvswitch.org

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



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

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

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

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

(e) CPU Utilization with Load Balance in Routing Brokers

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

Figure 7: Scalability Experiments

values of the initial samples are not consistent due to coordination and discovery process overhead until system stabilizes (*e.g.*, time for discovery of brokers and creating routes).

1) Scalability of the Broker Overlay Layer: Since the edge brokers are responsible for delivering data incoming from other brokers to subscribers in a local network, the computation overhead on edge brokers grows linearly as the number of adopted topics increases. Figure 7a and 7d 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 by the number of adopted topics, and average and maximum latency values grow as well. From these results, we can infer that if the number of incoming streams increases due to more number of topics per network, it affects latency values even though CPUs of edge brokers are not saturated.

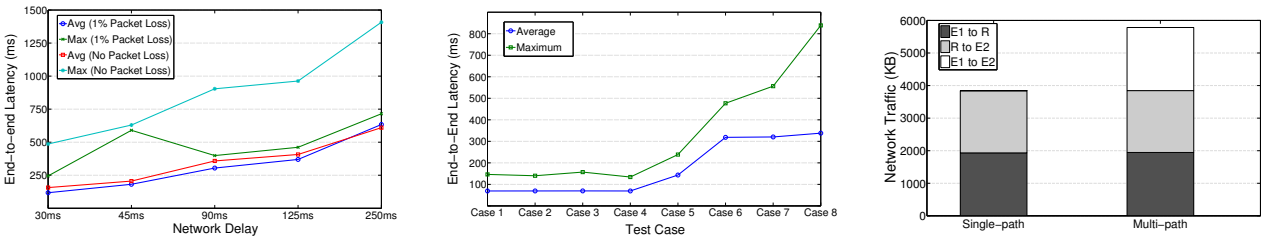
Our solution supports load balancing in the group of routing brokers and makes it possible to flexibly scale systems with the number of topics. Figure 7b and 7e present latency and CPU usage by different number of routing brokers. When the number of routing brokers is small, in this case 5, the CPU of the routing brokers become saturated and latency values are adversely impacted. However, after increasing the number of routing brokers to 10, latency values improve. The results in Figure 7e also validate that CPU usage linearly decreases by increasing the number of routing brokers.

2) Scalability of the Coordination Layer: We evaluate the scalability of a ZooKeeper-based centralized coordination service by increasing the number of simultaneous joining subscribers. Figure 7c shows latency, *i.e.*, the amount of time it takes for the server to respond to a client request. Figure 7f presents the number of used znodes and watches. We use *mntr*, a ZooKeeper command for monitoring service[5], to retrieve the experimental values presented in our results. We increase the number of subscribers from 2,000 to 10,000 in steps of 2,000. The average latency increases from 10 milliseconds to 20 milliseconds and the number of znodes and watches linearly increase approximately 2,000 and 4,000, respectively by the increased number of subscribers. The reason why the number of watches are twice compared to the number of znodes is that it needs to notify brokers for both publishers and subscribers if they have matching pub/sub endpoints.

C. Deadline-aware Overlays

We also conducted experiments to validate our deadline-aware overlays showing latency and overhead by comparing the performance parameters for multi-path and single-path overlays. A topology used for these experiments was shown in Figure 4. We use Dummynet [16] to simulate network delays and packet losses, which are common in WANs. These parameters are varied depending on geographic locations of brokers, which is a factor influencing the need for deadline-aware overlays. For multi-path overlay experiments, we use delay and loss data provided by Verizon, which shows latency and packet delivery statistics for communication between different countries across the globe.[6] We categorize

[5]http://zookeeper.apache.org/doc/trunk/zookeeperAdmin.html
[6]http://www.verizonenterprise.com/about/network/latency



(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 8: Deadline-aware Experiments

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 I) and experimented 8 possible combinations with given links (*i.e.*, L1, L2, and L3 as shown in Figure 4), and test cases described in Table I.

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

V. Related Work

Prior research on pub/sub systems can be classified into topic-based, attribute-based, and content-based depending on the subscription model. Topic-based model, such as Scribe [17], TIB/RV [18], and SpiderCast [19], groups subscription events in topics. In 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 [5], [6] allows subscribers to express their interests by specifying conditions on the data content of events, and the system filters out and delivers events based on the conditions. The OMG DDS adopts a data-centric model that groups subscriptions in topics, validates types of topic events, and also filters out events by conditions on data content using a special topic called *Content-Filtered Topic (CFT)*. Besides, it matches subscriptions based on offered and requested QoS parameters to disseminate data with assured service levels.

Pub/sub systems tend to form overlay networks to support application-level multicast rather than using IP-based multicast owing to the fact that IP multicast is not supported in WANs and the limited number of IP-based multicast addresses would not fit the potential number of logical channels for fine-grained subscription models [20]. Overlay architectures for pub/sub systems can be categorized into broker-based overlay [5], [6], [18], structured peer-to-peer [17], and unstructured peer-to-peer. GREEN [21] supports configurable overlay architectures for different network environments. PubSubCoord adopts a hybrid approach that constructs unstructured peer-to-peer overlays in LANs by dynamically discovering peers via multicast, and broker-based overlays in WANs.

BlueDove [14] is similar to our approach in that it achieves scalability and elasticity by harnessing cloud resources, and is a two-tier architecture to reduce the number of delivery hops and for simplicity. However, this service is designed for enterprise systems deployed in the cloud and does not consider the restrictions of physical locations of pub/sub endpoints. In our system, pub/sub endpoints located in different networks dynamically discover each other with the help of edge brokers, and therefore we consider physical restrictions of pub/sub endpoints and they cannot connect to any edge brokers.

Bellavista et al. [22] study QoS-aware pub/sub systems

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

over WANs and compare multiple existing pub/sub systems supporting QoS including DDS. In [9], the authors evaluate a pub/sub system for wide-area networks named Harmony and techniques for responsive and high available messaging. The Harmony system delivers messages through broker overlays placed in different physical networks, and pub/sub endpoints communicate via local brokers located in the same network. Although this effort describes a WAN-scale pub/sub solution with QoS support, it centers on selective routing strategies to balance responsiveness and resource usage in view of the fact that its architecture is based on multi-hop broker networks unlike our 2-hop solution.

IndiQoS [8] also proposes a pub/sub system with QoS support to reduce end-to-end latency by exploiting network-level reservation mechanisms, where message brokers are structured using distributed hash table (DHT). Similar to IndiQoS, we pursue low-latency and high availability but our solution also supports other QoS policies such as configurable transport reliability, data persistence, ordering, and resource management by controlling depth of history data and subscribing rate. We do not use a DHT solution for brokers and so a comparison along these lines will be part of our future work.

Recent research [11], [12] has broadened the scope of DDS to WANs by bringing in routing engines to disseminate data from a local network to others. Our solution utilizes similar routing engines and additionally solves the discovery and coordination problem between routing engines that otherwise requires significant manual efforts for large-scale systems. Finally, [23] suggests separation of control and data plane in next generation pub/sub systems, which is motivated by software-defined networking (SDN).

VI. Concluding Remarks

Emerging paradigms such as the Industrial Internet of Things illustrate the need to disseminate large volumes of data between a large number of heterogeneous entities that are geographically distributed, and require stringent QoS properties for data dissemination from the publishers of information to the subscribers. This paper presents the design, implementation, and evaluation of PubSubCoord, which is a cloud-enabled coordination and discovery service for internet-scale pub/sub applications. PubSubCoord supports scalability in terms of data dissemination as well as coordination, dynamic discovery, and configurable QoS properties. The test harness and capabilities in PubSubCoord are available for download from www.dre.vanderbilt.edu/pubsubcoord.

A. Key Insights and Discussion

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

- **PubSubCoord disseminates data in a scalable manner for systems having many pub/sub endpoints and topics**. The experimental results show that PubSubCoord can deliver streamed data within 100 milliseconds for a system having 10,000 subscribers and 1,000 topics distributed across more than 100 networks. As the number of topics increases in a system, our solution uses elastic cloud resources and load balancing techniques to deliver data in a scalable way. However, if the number of adopted topics per edge broker increases, service quality becomes worse as shown in the experimental results because edge brokers need to deal with more number of forwarding operations between routing brokers and pub/sub endpoints. If a system requires higher frequency or more number of topics per network, edge brokers possibly become bottleneck, so an elastic solution for edge brokers will be needed.
- **Centralized coordination service like ZooKeeper can serve as a pub/sub control plane for large-scale systems**. Our solution employs a centralized service for coordinating pub/sub brokers for its consistency and simplicity, and our experimental results show that average latency of the coordination service is 20 milliseconds for 10,000 subscribers and the number of data nodes and notifications linearly increase by organizing its data tree in a hierarchical way. Our experiments use a standalone server for coordination, but multiple servers as quorum can be used for scalability and fault-tolerance and ZooKeeper guarantees consistency of data between multiple servers. The quorum is more scalable for read operations, but not for write operations that require synchronizing data between servers. In future, we plan to carry out experiments with increasing the number of coordination servers to understand its scalability for pub/sub broker coordination in depth.
- **Configurable QoS supported by DDS can be used for low-latency data delivery in WANs by building multi-path overlays**. We use configurable *deadline* QoS to deliver data at low-latency by establishing selective multi-path overlays, and validate this approach by providing experimental results. Since not every path can be a delay-sensitive path, we need some higher level policy management (*e.g.*, offered and requested QoS management between network domains) to decide what characterizes a delay-sensitive path. In addition, although this approach assures low-latency data delivery, it occurs extra overhead by duplicating data delivery from multiple paths. To reduce the costs, we can utilize *ownership* QoS that dynamically selects an owner of data streams to reduce data traffic from backups, and the owner is changed to a backup when the owner fails to meet its deadline. Our deadline-aware overlay optimizations are possible due to capabilities of DDS; implementing similar optimizations for other messaging systems will require identifying similar opportunities.
- **End-to-end QoS management is required for ef-**

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

ACKNOWLEDGMENTS

We would like to thank DOC group members for their valuable feedback. 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.

REFERENCES

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

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

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

[4] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, vol. 10, no. 6, pp. 87–89, 2006.

[5] 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)*, vol. 19, no. 3, pp. 332–383, 2001.

[6] 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*. IEEE, 2002, pp. 611–618.

[7] 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*. ACM, 2009, p. 16.

[8] 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*. IEEE, 2005, pp. 101–108.

[9] 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*. ACM, 2010, pp. 40–45.

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

[11] 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*, vol. 86, no. 1, pp. 108–124, 2013.

[12] A. Hakiri, P. Berthou, A. Gokhale, D. C. Schmidt, and G. Thierry, "Supporting end-to-end scalability and real-time event dissemination in the omg data distribution service over wide area networks," *Submitted to Elsevier Journal of Systems Software (JSS)*, 2013.

[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*, vol. 8, 2010, pp. 11–11.

[14] 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*. IEEE, 2011, pp. 1254–1265.

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

[16] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.

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

[18] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen, "The information bus: an architecture for extensible distributed systems," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5. ACM, 1994, pp. 58–68.

[19] 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*. ACM, 2007, pp. 14–25.

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

[21] 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*. Springer, 2005, pp. 732–749.

[22] P. Bellavista, A. Corradi, and A. Reale, "Quality of service in wide scale publishsubscribe systems," *IEEE Communications Surveys & Tutorials*, 2014.

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

APPENDIX A.
BROKER IMPLEMENTATION DETAILS

In this section we describe some implementation details that use the OMG DDS pub/sub messaging system as the underlying pub/sub technology and the Curator framework that implements ZooKeeper.[8] Algorithms 1 and 2 describe the pseudo code of callback functions implemented in edge brokers and routing brokers, respectively. Callback functions are invoked by either the DDS endpoint discovery events in built-in entities or notifications by ZooKeeper services.

A. Edge Broker Implementation

Algorithm 1 Edge Broker Callback Functions

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

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

- ENDPOINT CREATED: This function is invoked when an endpoint in a network is created and activated by a built-in DDS DataReader.
- TOPIC NODE CACHE LISTENER: This function is invoked when a topic znode managed by an edge broker is created, deleted, or updated. It is activated by a ZooKeeper client process.
- ENDPOINT DELETED: This function invoked when an endpoint in a network is deleted and activated by a built-in DDS DataReader.

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

[8]http://curator.apache.org

The ENDPOINT CREATED callback function first creates a znode for a created endpoint (*i.e. ep* in Algorithms 1) that contains the topic name, type, QoS settings. If a relevant topic to the created endpoint has not appeared in an edge broker before, a cache for the topic znode and its listener for the topic are created to receive locator information of an assigned routing broker. When the znode for the topic is updated by a leader routing broker, it triggers the TOPIC CACHE LISTENER callback described in Algorithm 1.

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

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

B. Routing Broker Implementation

Algorithm 2 Routing Broker Callback Functions

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

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

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

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

When an endpoint is created or deleted, the edge brokers create or delete znodes for endpoints and these events will trigger the ENDPOINT CACHE LISTENER function in routing brokers that are responsible for topics involved with the endpoints. The data of znode cache for an endpoint (*ep* in the ENDPOINT CACHE LISTENER callback function) contains the locator of an edge broker where the endpoint is located as well as the topic name, type, and QoS settings. If the event type is creation, it adds the locator of the edge broker to the DDS Routing Service if it does not exist. Thereafter, it requests the DDS Routing Service to create a route for the topic based on the information provided by the content of the *ep* znode from this routing broker to the edge broker, if it does not exist. If the event type is deletion, it has to delete the locator and the topic route from the DDS Routing Service on the condition that no endpoints for that topic still exist.