Experience Paper: An Autonomous and Dynamic Coordination and Discovery Service for Wide-Area Peer-to-peer Publish/Subscribe

Kyoungho An[*]
Real-time Innovations (RTI)
Sunnyvale, California, USA 94089
kyoungho.an@gmail.com

Shweta Khare
Dept of EECS, Vanderbilt University
Nashville, Tennessee, USA 37235
shweta.p.khare@vanderbilt.edu

Aniruddha Gokhale
Dept of EECS, Vanderbilt University
Nashville, Tennessee, USA 37235
a.gokhale@vanderbilt.edu

Akram Hakiri
ISSAT Mateur, Univ de Carthage
Mateur, Bizerte, Tunisia 7050
akram.hakiri@gmail.com

ABSTRACT

Industrial Internet of Things (IIoT) applications are mission-critical, which require a scalable data sharing and dissemination platform that supports quality of service (QoS) properties such as timeliness, resilience, and security. Although the Object Management Group (OMG)'s Data Distribution Service (DDS), which is a data-centric, peer-to-peer publish/subscribe standard supporting multiple QoS properties, is well-suited to meet the requirements of IIoT applications, its design and current technology limitations constrains its use to local area networks only. Moreover, although broker-based bridging services exist to inter-connect isolated DDS networks, these solutions lack autonomous and dynamic coordination and discovery capabilities that are needed to bridge multiple, isolated networks on demand. To address these limitations, and enable a practical and readily deployable solution for IIoT, this paper presents and empirically validates *PubSubCoord*, which is an autonomous, coordination and discovery service for DDS endpoints operating over wide area networks.

CCS CONCEPTS

•Computer systems organization →Cloud computing; Peer-to-peer architectures; •Applied computing →*Event-driven architectures;*

KEYWORDS

Data Distribution Service, Pub/Sub, Discovery, Coordination

ACM Reference format:
Kyoungho An, Shweta Khare, Aniruddha Gokhale, and Akram Hakiri. 2017. Experience Paper: An Autonomous and Dynamic Coordination and Discovery Service for Wide-Area Peer-to-peer Publish/Subscribe. In *Proceedings of ACM International Conference on Distributed and Event-based Systems, Barcelona, Spain, June 19–23, 2017 (DEBS'17),* 10 pages.
DOI: 10.1145/3093742.3093910

1 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. A class of IoT, referred to as Industrial IoT (IIoT) [6], found in domains such as transportation, healthcare, manufacturing, and energy requires different quality of service (QoS) properties, such as timeliness, reliability, and security, for their applications that are geographically distributed and operate 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 a large number of entities at Internet-scale in a timely, and reliable manner.[1]

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. Specifically, the need to scalably and reliably disseminate large volumes of IIoT data in real-time motivates the use of scalable, data-centric pub/sub with support for configurable QoS properties that can operate over WANs. Many pub/sub messaging solutions exist today that can operate over multiple networks including industrial solutions [19, 27, 33] and research efforts [12, 15, 29]. Some of these even support QoS properties, such as availability [15, 29], configurable reliability [27], durability [19], and timeliness [11, 21]. However, these solutions either do not address the scalable data-centric needs, or tend to support only a subset of properties at a time and in most cases, support for configurable QoS properties, security, and resilience is lacking.

The Object Management Group (OMG)'s Data Distribution Service (DDS) [28] standard for data-centric pub/sub holds substantial promise for IIoT applications because of its support for configurable QoS policies, dynamic discovery of end points, low-latency and

[*] Work performed by the first author while being a graduate student.

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

DEBS'17, Barcelona, Spain
© 2017 ACM. 978-1-4503-5065-5/17/06…$15.00
DOI: 10.1145/3093742.3093910

resilience due to its peer-to-peer architecture. However, there still remain a number of unresolved challenges in using OMG DDS in IIoT systems deployed over multiple, distributed networks. For instance, DDS uses multicast as a default transport to automatically 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 data to the destination endpoints.

One approach to supporting OMG DDS over multiple distributed networks relies on broker-based solutions [18, 23], where brokers are used to inter-connect peers in different networks. It is thus conceivable to think that these broker-based solutions in conjunction with the data-centric and configurable QoS features provided by OMG DDS can readily provide a solution for IIoT. However, it is still challenging to realize such capabilities when the deployed system spans a large number of networks comprising pub/sub endpoints that illustrate heterogeneity in terms of topics, their types, and requested versus offered QoS policies. Additionally, such solutions tend to lack autonomous and dynamic discovery and coordination of brokers to connect pub/sub peers in multiple, disparate networks on demand.

Further, as industrial systems progressively integrate an increasing number of sub-systems located in multiple disparate networks, the number of deployed brokers may become very large. Consequently, the amount of effort to manage these dispersed brokers becomes unwieldy due to the distributed state and configuration. Moreover, forming an efficient overlay network of brokers that offers both scalability and low latency becomes even harder.

To address these technical concerns while still benefiting from technologies, such as OMG DDS, and to realize a practical and readily deployable solution that IIoT applications can leverage, we present *PubSubCoord,* which provides an autonomous and dynamic coordination and discovery service for geographically distributed brokers to transparently connect pub/sub endpoints and realize scalable and low-latency, data-centric pub/sub systems. Our earlier work presented only a vision of PubSubCoord [3, 4]. This paper significantly extends this vision by delving into the technical details of its design, and makes the following contributions:

- To address the scalability and low latency requirements of data dissemination across multiple distributed networks, and to limit the complexity of managing a broker network, PubSubCoord introduces a two-level broker hierarchy deployed over a pub/sub overlay network, which in turn incurs only two hops in the worst case in the data dissemination path across the overlay of distributed, isolated networks.
- To achieve autonomous and dynamic discovery and data routing between brokers, PubSubCoord exploits and extends the open-source ZooKeeper coordination service [20] in a novel way to create dissemination paths for the dynamic overlay network of brokers and endpoints.
- For those dissemination paths that need both low latency and high-availability requirements, PubSubCoord trades off resource usage in favor of deadline-aware overlays that build multiple, redundant paths between brokers.
- An implementation of PubSubCoord using Real Time Innovation (RTI)'s implementation of OMG DDS as the underlying pub/sub messaging system is presented and empirically evaluated to demonstrate the feasibility of the solution and its ability to support end-to-end QoS properties without incurring any undue overhead introduced by the broker architecture.

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

2 DESIGN AND IMPLEMENTATION OF PUBSUBCOORD

In the realm of (I)IoT, there is an emerging trend towards exploiting the entire spectrum of resources ranging from the edge to the cloud [32]. The separation of concerns and latency/scalability benefits of edge-cloud architectures motivates the two levels of the broker hierarchy in our PubSubCoord design.

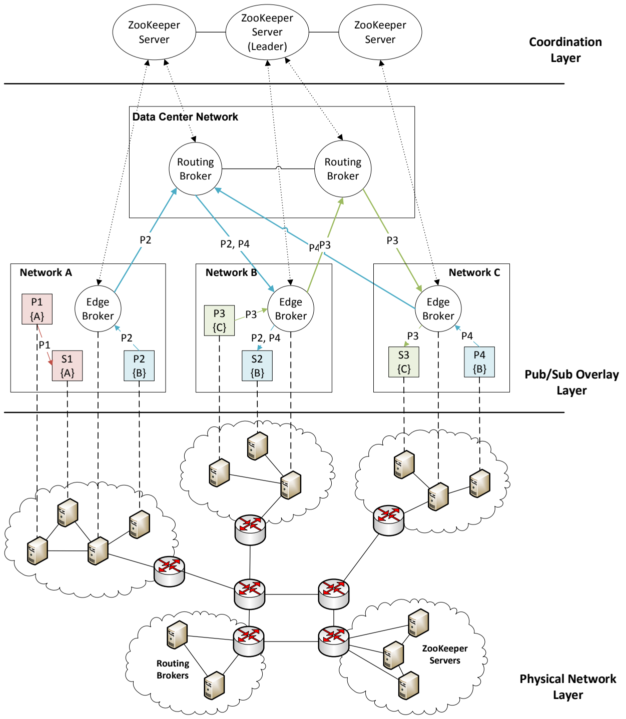


Figure 1: PubSubCoord Architecture

Figure 1 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 overlay network of brokers and pub/sub endpoints in a system. An *edge broker* is directly connected to endpoints in a local area network

(LAN) to serve as a gateway 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 spread across the WANs. The coordination layer comprises an ensemble of ZooKeeper servers used for coordination between the brokers. ZooKeeper has been used by many industrial solutions, such as FaRM [14], Ambry [26], CoRAL [34] and Netflix's Curator for distributed coordination. The routing broker layer resides in the cloud (or fog) along with the coordination logic while the edge broker layer resides in the edge.

Data dissemination in PubSubCoord can be explained using an example from Figure 1. 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 low latency without incurring a hop to the routing broker layer. $P2$, $P4$, 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.

In the remainder of this section, we discuss the different design decisions and implementation aspects of PubSubCoord.

2.1 Scalability via Separation of Concerns

Context and challenges: 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 that 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 subscriptions. Maintaining such distributed state may become unwieldy. Secondly, 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. Moreover, the dissemination latency suffers due to multiple broker hops.

Solution approach: To resolve these challenges, PubSubCoord's broker overlay layer is structured as a two-tier architecture of brokers with strict separation of responsibilities: at the bottom tier are *edge brokers* that manage pub/sub issues within an isolated network, and at the top tier are *routing brokers* in the cloud, which route traffic among different edge brokers. Thus, we incur a maximum two-hop latency over the broker overlay network for communication across isolated LANs.

Design details, consequences and their resolution: 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 topic load and number of connections between edge brokers.

An immediate consequence of our 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 multiple different topic data among the large number of endpoints. Having multiple routing brokers and organizing them in multiple levels of hierarchy similar to domain name service (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 and thereby scalability of the system. 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. The number of routing brokers can thus be scaled up/down to dynamically adapt to system load using cloud-based autoscaling algorithms, such as the one we proposed in prior work [31]. Thus, our design enables dynamic adaptation to system load and autoscaling to existing demand. This design can also take care of faults in routing brokers where a new routing broker can be spawned on demand in case of failure and the failed broker's topics can be re-distributed among existing brokers.

2.2 Low Latency for Intra-LAN Data Dissemination

Context and challenges: Systems that cater to supporting WAN-scale operations may tend to overlook LAN-specific issues and force a one-size-fits-all WAN-scale solution even for a LAN-scale operation, thereby incurring unnecessary overhead and performance penalties on applications. For PubSubCoord this implies that any intra-LAN data dissemination should not incur the overhead of WAN latencies stemming from having to utilize the cloud-based routing broker layer.

Solution approach: It is in this regard that the two-level broker solution of PubSubCoord and separation of concerns addresses these potential problems, where the edge broker layer handles all the LAN-specific pub/sub issues.

Design details, consequences and their resolution: In our solution, any pub/sub traffic that is local (i.e., where publishers and subscribers reside in the same isolated network) is not allowed to reach the routing brokers; rather the traffic and dissemination is handled by the edge brokers themselves thereby avoiding the penalty of a round-trip WAN latency and in turn avoiding undue overhead imposed on the routing brokers.

A consequence of our design is that the edge broker may get overloaded and/or fail. To that end, elastic and load balancing decisions similar to that used in the cloud can be used for edge brokers [13]. Similarly, edge brokers can be made fault tolerant using mechanisms such as our prior work [5].

2.3 WAN-scale Autonomous and Dynamic Endpoint Discovery and Dissemination

Context and challenges: In IIoT systems, the publishers and subscribers that are matched with each other are most likely distributed across separate isolated networks, and moreover they may join or leave the system 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. As discussed earlier, the dynamic discovery mechanisms of OMG DDS do not work at WAN scale due to limitations with IP multicast in the WAN and NAT/firewall issues.

Solution approach: To address this need, PubSubCoord uses a coordination layer (top layer shown in Figure 1) comprising an ensemble of ZooKeeper [20] servers, which help brokers discover each other and build broker overlay networks using the PubSubCoord coordination logic. ZooKeeper is an open-source, highly efficient, centralized and replicated service which provides generic distributed coordination and synchronization primitives such as leader election, barrier synchronization,locking, group membership,sharing configuration metadata, etc. The data model of ZooKeeper is structured like a file system in the form of znodes (*i.e.*, a ZooKeeper data object containing its path and data content). Applications using the znodes are required to specialize the znodes to add application-specific semantics. ZooKeeper provides a *watch* mechanism to notify a client of ZooKeeper of a change to a *znode* that is being watched by that client. Specializing the ZooKeeper znode structure for our needs and using its watch mechanisms provides us with a capability that can take autonomous and dynamic decisions.

Design details, consequences and their resolution: Figure 2 shows the znode data tree structure of PubSubCoord specialized 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. A leader routing broker makes load balancing decisions for topic distribution among the cluster of 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 topic assignment.

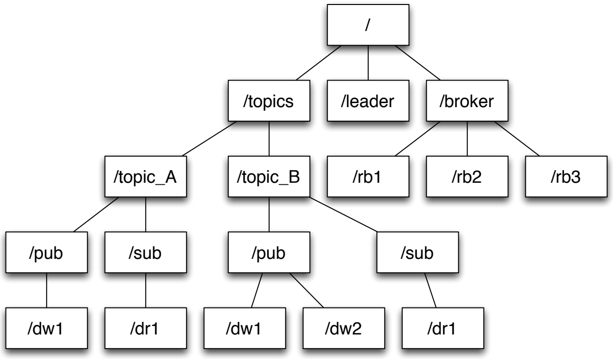


Figure 2: Specialized ZNode Tree for PubSubCoord

Dynamic changes in the system (*e.g.*, broker or pub/sub endpoint join/leave, topic creation/deletion) are reflected in the creation, update, and deletion of znodes in the znode-tree. Brokers use the watch mechanism to set watches on interesting znodes to receive change notifications and to take actions autonomously. PubSubCoord exploits the *ephemeral* mode feature of ZooKeeper, where a specific znode in the tree (and its subtree) is automatically deleted from the tree when the client session handling this znode is lost thereby providing automatic cleanup of state.

We rely on ZooKeeper to provide the desired fault tolerance and persistence of the znode tree. Details on all the interactions that take place in this context are provided in Section 2.6 where we show how brokers discover each other and establish communication routes.

2.4 Overload Management and Fault Tolerance

Context and challenges: Performance of PubSubCoord can be impacted by at least two factors: load and failures in the routing and edge brokers; and network congestion on the two hop route between edge and routing brokers over the broker overlay.[2] Load on a routing broker depends on the number of topics it manages and the number of edge brokers it interconnects through itself. Thus, appropriate strategies are needed for routing broker load balancing. In the case of faults, although many kinds of faults are possible, we focus only on tolerating failures in the routing broker layer.

2.4.1 Routing Broker Load Management

. **Solution approach:** Load balancing is handled by the leader routing broker, which is elected among the routing brokers using ZooKeeper. In the current implementation, the leader routing broker assigns a new topic to the least loaded routing broker in terms of number of topics.

Design details, consequences and their resolution: 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 remaining 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.

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. By using the Strategy design pattern our design can allow plugging in other load balancing schemes (*e.g.*, least loaded based on CPU utilization, the number of connections, QoS policies, or DHT-based load balancing).

2.4.2 Deadline-aware Overlay Optimizations

. **Solution approach:** The 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.

[2] Overload management and fault tolerance at the Edge Broker layer is an ongoing work.

Design details, consequences and their resolution: Figure 3 illustrates the idea. These optimizations can be leveraged by pub/sub flows that require stringent assurances on reliable and deadline-driven data delivery. For example, OMG DDS uses *deadline* QoS as a contract between pub/sub flows, which can be 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.

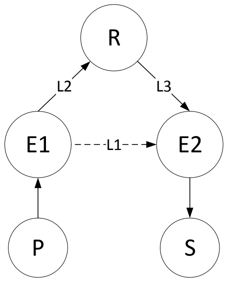


Figure 3: Multi-path Deadline-aware Overlay Concept

Note that this strategy is vulnerable to abuse by applications, and doing this for every edge broker will be infeasible due to the very large connection state that every edge broker must manage. Thus, higher level policies and admission control mechanisms will be needed to restrict such optimizations to the most critical flows.

2.4.3 Broker Fault Tolerance

. **Solution approach:** 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.

Design details, consequences and their resolution: There may be transient periods of time when brokers fail and their load has to be migrated. We have not yet evaluated the impact of such transient unavailability of service and its impact on loss of samples being disseminated. Our ongoing work is designing solutions to minimize the impact of these situations. Second, the ZooKeeper server itself may fail. To provide a scalable and fault-tolerant service at the coordination layer, ZooKeeper supports the notion of 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.

2.5 Concrete Instantiation of PubSubCoord with OMG DDS

To demonstrate our ideas and enable a readily deployable solution, we have implemented PubSubCoord in the context of OMG Data Distribution Service (DDS) where the pub/sub endpoints use the OMG DDS technology, specifically the RTI Connext DDS implementation, along with its requested-offered QoS model. We have used Curator,[3] 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. The edge and routing brokers are entities we provided beyond existing DDS software entities. The edge brokers collocate themselves with RTI Connext's Routing Service, which provides edge-based routing capabilities. We have evaluated our design in an experimental testbed as we show later.

2.6 PubSubCoord with OMG DDS in Action

So far we have presented the architectural elements and implementation details of PubSubCoord. We now present details on the runtime interactions among the architectural elements that realizes the various capabilities of PubSubCoord. We describe how the brokers interact and the algorithms they execute to update their internal states used in routing the streamed pub/sub data. We show these interactions concretely in the context of publishers and subscribers that use OMG DDS and its QoS policies.

2.6.1 Routing Broker Responsibilities. Figure 4 illustrates the sequence diagram showing the runtime interactions of the routing brokers. The corresponding 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. In our case, it is supplied by RTI's DDS implementation.

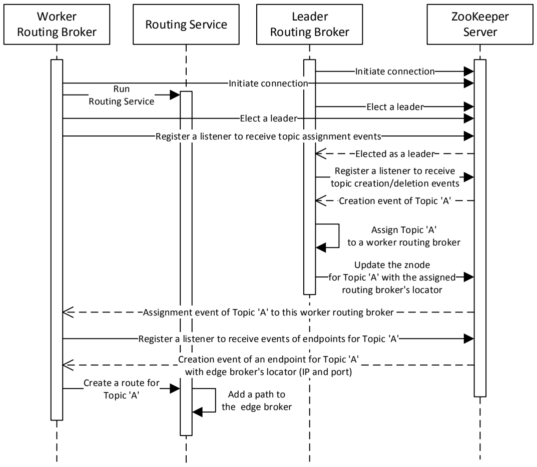

Worker Routing Broker
Routing Service
Leader Routing Broker
ZooKeeper Server
Initiate connection
Initiate connection
Run Routing Service
Elect a leader
Elect a leader
Register a listener to receive topic assignment events
Elected as a leader
Register a listener to receive topic creation/deletion events
Creation event of Topic 'A'
Assign Topic 'A' to a worker routing broker
Update the znode for Topic 'A' with the assigned routing broker's locator
Assignment event of Topic 'A' to this worker routing broker
Register a listener to receive events of endpoints for Topic 'A'
Creation event of an endpoint for Topic 'A' with edge broker's locator (IP and port)
Create a route for Topic 'A'
Add a path to the edge broker

Figure 4: Routing Broker Sequence Diagram

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

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 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).

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.
- ENDPOINT LISTENER – This function is invoked when children pub/sub endpoints of a znode for an assigned topic are created, deleted, or updated.

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.

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

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. 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 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.

2.6.2 Edge Broker Responsibilities. The interactions of the edge broker with ZooKeeper are similar to that of the routing broker. For space considerations, we do not provide its detailed behavior. In summary, 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.

3 EXPERIMENTAL VALIDATION OF PUBSUBCOORD

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

3.1 Overview of Testbed Configurations and Testing Methodology

Our testbed is a private cloud managed by OpenStack comprising 60 physical machines each with 12 cores and 32 GB of memory. Both Edge and Routing broker instances run in their own Virtual Machine

(VM), while multiple publisher and subscriber test applications share a VM. Each Virtual Machine (VM) used in our experiment was configured with one virtual CPU and 2 GB RAM. To experiment in a multi-network environment, we used Neutron[4], an OpenStack project for networking as a service, to create virtual networks.

All publish/subscribe end-points, were implemented using RTI Connext 5.1[5] and were configured to use RELIABLE *reliability* QoS to avoid data loss at the transport level through data retransmission; 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. Depending on the application's requirements, the QoS policies can be varied and hence the performance results may change according to different QoS settings.

Each publisher test application sends a 64 byte data sample every 50 milliseconds (other sizes could be used). We used *netem*[6] network emulator to emulate 20 milliseconds roundtrip LAN latencies and 80 milliseconds roundtrip WAN latencies. We measure the end-to-end latency from publishers to subscribers to evaluate our solution. End-to-end latency was calculated as the time difference between the send timestamp at the publisher and reception timestamp at the subscriber. The Precise Time Protocol (PTP) [10] was used for fine-grained time synchronization across all VMs. 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 initial samples are not consistent due to coordination and discovery overhead (*e.g.*, time for discovery of brokers and creation of routes). The broker CPU usage is also measured along with the latency values to understand how different settings, *i.e.*, number of topics per network and number of routing brokers, affect dissemination scalability.

3.2 Data Locality Results

Edge Broker layer in PubSubCoord is responsible for dissemination of local traffic thereby preventing WAN latencies in sending data all the way to a cloud back-end for local data dissemination. To evaluate the latency benefits of the Edge Broker layer, we measure the end-to-end latency of data dissemination under different values of data locality, *i.e*, fraction of topics in an isolated network which are local to the network and do not have interested subscribers in another network which would necessitate going through the cloud Routing Broker layer.

We created five local networks each serviced by its own Edge Broker VM and five Routing Broker VMs. In each of the five isolated networks we used up to 10 client VMs for hosting pub/sub test applications where each client VM can host a maximum of 20 pub/sub test applications. We created up to 100 topics in each isolated network with different values of data locality to observe its effect on the dissemination latency. For example, in case of 100 topics per isolated network and a data locality of .9, 90 topics are local to this isolated network and the remaining 10 topics are global (*i.e* have interested subscribers in some other networks).

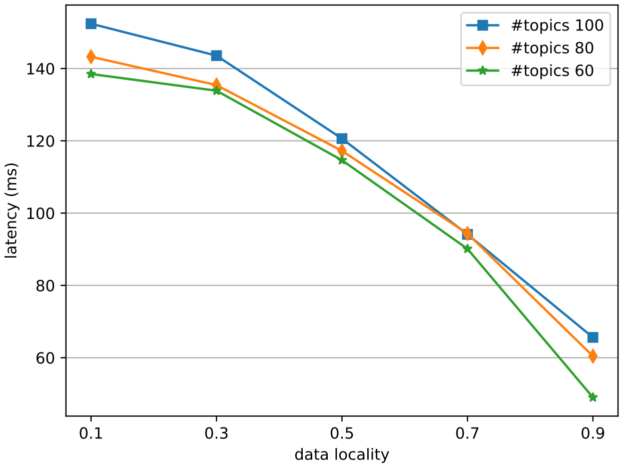


Figure 5: End-to-end Latency with increasing values of data locality

Figure 5 shows the end-to-end latency of data dissemination for increasing values of data locality for 60, 80 and 100 topics per isolated network. As seen in Figure 5, as the data locality increases, the end-to-end dissemination latency decreases since an increasingly large fraction of messages are local to an isolated network and do not incur WAN latencies for their dissemination thus validating the design of the Edge Broker Layer.

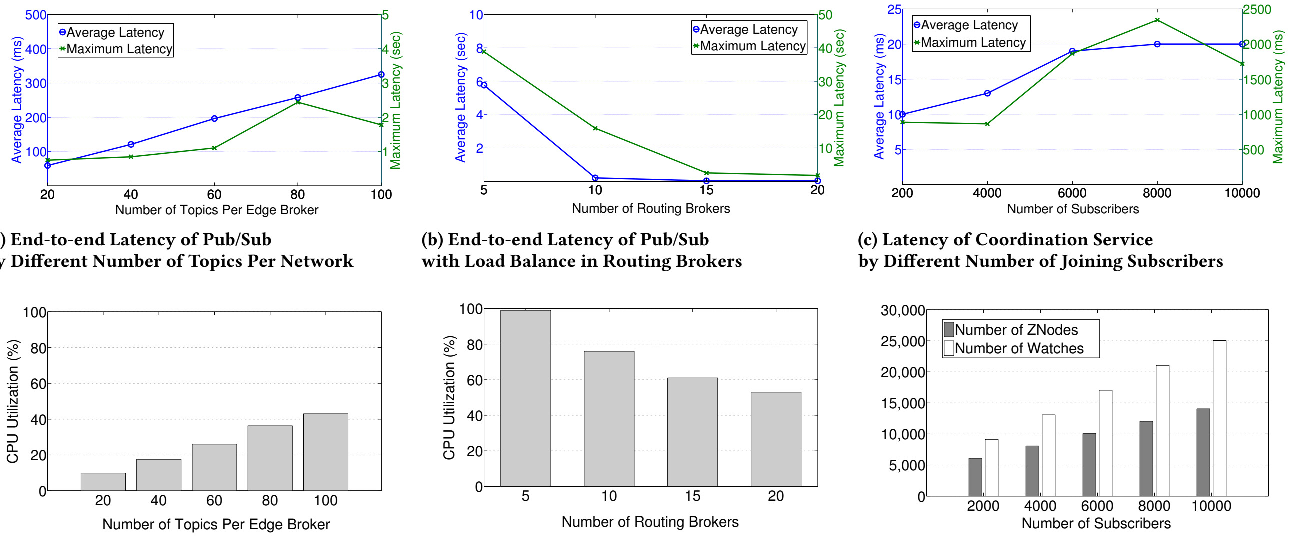
3.3 Scalability Results

We used a total of 400 VMs in our scalability experiments: 160 VMs were used for the brokers (120 VMs for edge brokers and 40 VMs for routing brokers) and of the remaining 240 VMs, 40 VMs were used for publishers and 200 VMs for subscribers. Each of these VMs runs either 25 publisher or 50 subscriber test applications. We placed 50 publishers or 100 subscribers in each network (*i.e.*, 2 VMs for each network, thereby creating a total of 1000 publishers and 10,000 subscribers). Subscribers in each network are interested in 100 topics out of 1000 topics in the system.

3.3.1 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 6a and 6d show the latency and Edge Broker CPU utilization results for increasing number of topics in each local network. The CPU utilization increases linearly with the number of adopted topics; the average and maximum latency values grow as well.

Our solution supports load balancing at the Routing Broker layer, allowing the system to scale with the number of topics in the system. Figure 6b and 6e present latency and CPU usage for different number of routing brokers. When the number of routing brokers is small, in this case 5, the CPU of the routing brokers becomes saturated and latency gets adversely impacted. However, after autoscaling the number of routing brokers to 10, latency values

[4] https://wiki.openstack.org/wiki/Neutron
[5] https://community.rti.com/rti-doc/510/ndds.5.1.0/doc/pdf/RTI_CoreLibrariesAndUtilities_UsersManual.pdf
[6] https://wiki.linuxfoundation.org/networking/netem



(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 6: Scalability Experiments

improve. The results in Figure 6e also validate that CPU usage decreases linearly by increasing the number of routing brokers.

3.3.2 Scalability of the Coordination Layer. We evaluate the scalability of a ZooKeeper-based centralized coordination service by increasing the number of simultaneously joining subscribers from 2,000 to 10,000 in steps of 2,000. Figure 6c shows latency, *i.e.*, the amount of time it takes for the server to respond to a client request, increases from 10ms to 20ms. We used *mntr*, a ZooKeeper monitoring service [7], to retrieve the number of used znodes and watches. Figure 6f presents results for increasing number of used znodes and watches as the system scales. These results show that the overhead of ZooKeeper based centralized coordination service remains acceptable even at scale. In our experiments, we have used zookeeper in standalone mode, however, as we increase the number of ZooKeeper servers in the ensemble, the latency overhead is expected to decrease further.

3.4 Deadline-aware Overlays

Deadline-aware overlays can be used for topics which have stricter data delivery requirements in case of congested, lossy and slow WAN links. To evaluate deadline-aware overlays in PubSubCoord, we compare the dissemination latency and broker overhead for deadline-aware multi-path vs single-path overlays. We used the topology shown in Figure 3 for our experiments. Verizon's[8] delay and loss dataset was used for emulating WAN link behavior with Dummynet [30]. We categorized 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 with 8 possible combinations on the given links (*i.e.*, L1, L2, and L3 as shown in Figure 3). These test cases are described in Table 1.

Table 1: Deadline-aware Overlays Experiment Cases

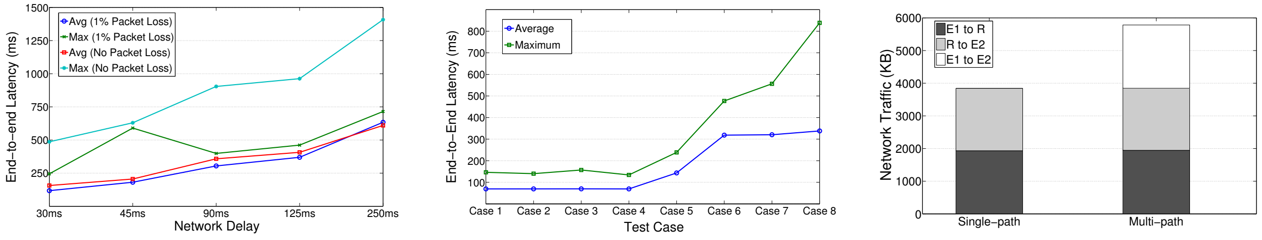
Test Cases	L1	L2	L3
Case 1	A	A	A
Case 2	A	A	B
Case 3	A	B	A
Case 4	A	B	B
Case 5	B	A	A
Case 6	B	A	B
Case 7	B	B	A
Case 8	B	B	B

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

Figure 7a, shows the average and maximum latencies for single-path overlays for increasing values of emulated network latencies under both conditions- with and without packet loss. Figure 7b, shows the average and maximum latencies for deadline-aware multi-paths for all 8 cases as described in Table 1. We see that test cases 1 to 5 for multi-path overlays perform better than any of cases for single-path overlays; and all cases of multi-path overlays outperform single-path overlay with 125 milliseconds network delay and 1% packet loss. Hence, topics with strict delivery requirements can benefit from deadline-aware overlays under adverse WAN link conditions.

However, maintaining multi-path overlays does impose additional computation and network transfer overhead at the edge broker. We measured the network transfer overhead for 10,000 samples from a publisher to a subscriber for both single-path and

[7] http://zookeeper.apache.org/doc/trunk/zookeeperAdmin.html
[8] 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 7: Deadline-aware Experiments

multi-path overlays by using *tcpdump*[9]. Figure 7c, shows the additional network transfer overhead imposed by multi-path overlay.

4 RELATED WORK

Publish-Subscribe systems can be classified into three types (based on the trade-off between simplicity and performance on one hand and expressiveness on the other) [16] as follows: (1) Topic-based, (2) Attribute-based, and (3) Content-based systems. The most expressive form is Content-based publish-subscribe [7], where subscriptions can be arbitrary boolean functions on the entire content of messages (e.g., XML documents). In Attribute-based pub-sub [22], messages are annotated with various attributes and subscriptions are expressed as predicates over these attributes. A publication matches a subscription if and only if all its attribute values satisfy the corresponding predicates of the subscription. Topic-based pub-sub [17] is the simplest form in which publishers tag their publications with a topic name; subscribers declare their interest by submitting subscriptions to specific topic names and all subscribers that have subscribed to a topic receive the message. Despite its simplicity, Topic-based pub-sub is widely used in industry, for example: Google Cloud Messaging (GCM), Amazon SNS, Apache Kafka, Apache Hedwig, Spotify, etc.

However, not many solutions support a variety of configurable QoS policies [8, 11] such as reliability, persistence, durability, deadline based delivery, security etc., which is much needed for IIoT application domains. The OMG DDS [28] is a Topic-based publish subscribe standard which supports low latency, peer-to-peer data delivery with a variety of configurable QoS settings. It has been deployed in many critical application domains such as healthcare [10],smart-grids [11], etc. However, as noted earlier, current technology limitations restrict the use of DDS to a LAN.

To support WAN-scale data dissemination, pub/sub systems tend to form an overlay network of brokers. On the basis of the overlay topology, pub/sub systems can be categorized into: (1) Tree-based overlays [12], (2) cluster-based overlays [9], (3) structured/unstructured peer-to-peer overlays [2, 35], and (4) cloud-based overlays [7, 17, 22, 24, 25, 36]. In tree-based overlays, brokers are organized into a tree overlay. Tree-based overlays incur multi-hop routing latencies, lack reconfiguration flexibility and impose costly maintenance of routing state information. In cluster-based overlays, brokers are organized into an unstructured overlay, where semantically similar brokers are grouped into the same cluster. However, due to frequent churn of nodes, maintaining semantic overlays leads to high latency. In peer-to-peer overlays, there are no dedicated brokers; instead each participating client is responsible for a small portion of the subscription space.

Although the peer-to-peer architecture is amenable to operate in wide-area networks with unreliable links and high node churn, it incurs high latency due to multi-hop routing and is not suitable for taking advantage of a well-engineered cloud environment where network links and server membership are much more stable. Cloud-based solutions offer low-latency, scalable, single-hop overlay routing of messages to all interested subscribers. However, most of the proposed solutions [22, 25, 36] are for attribute-based pub/sub where they support only a fixed attribute space for participating clients. If there are multiple applications with different attribute spaces, these cloud-based solutions do not offer a way in which multiple attribute spaces can be supported simultaneously.

Dynamoth [17] is similar to our solution but it provides a load-balanced topic-based pub/sub service in the cloud with single-hop routing. Taking inspiration from recent edge/fog computing advances, PubSubCoord introduces a second Edge Broker layer, which allows us to gain latency benefits for local data dissemination. Similarly, for edge/fog centric low-latency data dissemination, FogMQ [1] supports online migration of message brokers to facilitate near-the-edge data analytics.

PubSubCoord adopts a two-layer architecture for low latency data dissemination; transparent and autonomous bridging of isolated DDS LANs and separation of local data dissemination concerns from the cloud. Like many cloud-based solutions, the Routing Broker layer in PubSubCoord allows us to use elastic cloud resources. Although architecturally different from pub/sub systems like Apache Kafka and Hedwig, PubSubCoord also uses Zookeeper as its centralized coordination service.

Recent research including our prior work [18, 23] has broadened the scope of DDS to WANs by bringing in routing engines to disseminate data from a local network to others. Many of these prior efforts require either invasive changes to existing applications or need specific changes at different layers of the networking stack, which makes it hard to readily deploy such solutions. Our proposed solution utilizes similar routing engines introduced in the related

[9] http://www.tcpdump.org
[10] https://www.rti.com/industries/healthcare
[11] https://www.rti.com/industries/energy

work and additionally solves the automatic discovery and coordination problem between routing engines that otherwise requires significant manual efforts for large-scale systems.

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

ACKNOWLEDGMENTS

This work is supported in part by NSF CAREER CNS 0845789 and NSF US Ignite CNS 1531079. 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] Sherif Abdelwahab and Bechir Hamdaoui. 2016. FogMQ: A Message Broker System for Enabling Distributed, Internet-Scale IoT Applications over Heterogeneous Cloud Platforms. *arXiv preprint arXiv:1610.00620* (2016).

[2] I. Aekaterinidis and P. Triantafillou. 2006. PastryStrings: A Comprehensive Content-Based Publish/Subscribe DHT Network. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. 23–23. DOI: http://dx.doi.org/10.1109/ICDCS.2006.63

[3] Kyoungho An and Aniruddha Gokhale. 2014. PubSubCoord: A Cloud-enabled Coordination Service for Internet Scale OMG DDS Applications. In *Poster Session of 8th ACM International Conference on Distributed Event-based Systems (DEBS)*. IEEE, Mumbai, India.

[4] Kyoungho An, Aniruddha Gokhale, Sumant Tambe, and Takayuki Kuroda. 2015. Wide Area Network-scale Discovery and Data Dissemination in Data-centric Publish/Subscribe Systems. In *Proceedings of the Poster Session of ACM/IFIP/Usenix Middleware Conference*. ACM/IFIP/Usenix, Vancouver, Canada, 234–245.

[5] Kyoungho An, Shashank Shekhar, Faruk Caglar, Aniruddha Gokhale, and Shivakumar Sastry. 2014. A Cloud Middleware for Assuring Performance and High Availability of Soft Real-time Applications. *Elsevier Journal of Systems Architecture (JSA)* 60, 9 (Dec. 2014), 757–769. DOI: http://dx.doi.org/10.1016/j.sysarc.2014.01.009

[6] Marco Annunziata and Peter C Evans. 2012. Industrial Internet: Pushing the Boundaries of Minds and Machines. *General Electric* (2012).

[7] R. Barazzutti, T. Heinze, A. Martin, E. Onica, P. Felber, C. Fetzer, Z. Jerzak, M. Pasin, and E. Rivire. 2014. Elastic Scaling of a High-Throughput Content-Based Publish/Subscribe Engine. In *2014 IEEE 34th International Conference on Distributed Computing Systems*. 567–576. DOI: http://dx.doi.org/10.1109/ICDCS.2014.64

[8] Paolo Bellavista, Antonio Corradi, and Andrea Reale. 2014. Quality of Service in Wide Scale Publish-Subscribe Systems. *IEEE Communications Surveys & Tutorials* (2014).

[9] Fengyun Cao and J. P. Singh. 2004. Efficient event routing in content-based publish-subscribe service networks. In *IEEE INFOCOM 2004*, Vol. 2. 929–940 vol.2. DOI: http://dx.doi.org/10.1109/INFCOM.2004.1356980

[10] Lewis Carroll. 2012. IEEE 1588 Precision Time Protocol (PTP). http://www.eecis.udel.edu/~mills/ptp.html. (2012).

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

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

[13] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*. ACM, 301–314.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. 401–414.

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

[16] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *ACM Computer Survey* 35 (June 2003), 114–131. Issue 2. DOI: http://dx.doi.org/10.1145/857076.857078

[17] J. Gascon-Samson, F. P. Garcia, B. Kemme, and J. Kienzle. 2015. Dynamoth: A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 486–496. DOI: http://dx.doi.org/10.1109/ICDCS.2015.56

[18] Akram Hakiri, Pascal Berthou, Aniruddha Gokhale, Douglas Schmidt, and Thierry Gayraud. 2013. 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 (Oct. 2013), 2574–2593. DOI: http://dx.doi.org/10.1016/j.jss.2013.04.074

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

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

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

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

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

[24] X. Ma, Y. Wang, and X. Pei. 2015. A Scalable and Reliable Matching Service for Content-Based Publish/Subscribe Systems. *IEEE Transactions on Cloud Computing* 3, 1 (Jan 2015), 1–13. DOI: http://dx.doi.org/10.1109/TCC.2014.2338327

[25] Xingkong Ma, Yijie Wang, Qing Qiu, Weidong Sun, and Xiaoqiang Pei. 2014. Scalable and elastic event matching for attribute-based publish/subscribe systems. *Future Generation Computer Systems* 36 (2014), 102 – 119. DOI: http://dx.doi.org/10.1016/j.future.2013.09.019 Special Section: Intelligent Big Data ProcessingSpecial Section: Behavior Data Security Issues in Network Information PropagationSpecial Section: Energy-efficiency in Large Distributed Computing ArchitecturesSpecial Section: eScience Infrastructure and Applications.

[26] Shadi A Noghabi, Sriram Subramanian, Priyesh Narayanan, Sivabalan Narayanan, Gopalakrishna Holla, Mammad Zadeh, Tianwei Li, Indranil Gupta, and Roy H Campbell. 2016. Ambry: LinkedIn's Scalable Geo-Distributed Object Store. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 253–265.

[27] OASIS. 2014. MQTT Version 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. (2014).

[28] OMG. 2007. The Data Distribution Service specification, v1.2. http://www.omg.org/spec/DDS/1.2. (2007).

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

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

[31] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient Autoscaling in the Cloud using Predictive Models for Workload Forecasting. In *4th IEEE International Conference on Cloud Computing (Cloud 2011)*. IEEE, Washington, DC, 500–507.

[32] Mahadev Satyanarayanan. 2011. Mobile Computing: The Next Decade. *SIGMOBILE Mob. Comput. Commun. Rev.* 15, 2 (Aug. 2011), 2–10. DOI: http://dx.doi.org/10.1145/2016598.2016600

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

[34] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. 2017. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *To appear in the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017)*. ACM.

[35] Spyros Voulgaris, Etienne Riviere, Anne-Marie Kermarrec, Maarten Van Steen, and others. 2006. Sub-2-Sub: Self-Organizing Content-Based Publish Subscribe for Dynamic Large Scale Collaborative Networks.. In *IPTPS*.

[36] Y. Wang and X. Ma. 2015. A General Scalable and Elastic Content-Based Publish/Subscribe Service. *IEEE Transactions on Parallel and Distributed Systems* 26, 8 (Aug 2015), 2100–2113. DOI: http://dx.doi.org/10.1109/TPDS.2014.2346759