A Programmable and Reliable Publish/Subscribe System for Multi-Tier IoT

Wei-Tsung Lin, Rich Wolski, Chandra Krintz
Department of Computer Science
University of California, Santa Barbara, CA 93106
{weitsung, rich, ckrintz}@cs.ucsb.edu

***Abstract*—We introduce *Canal*, a programmable, topic-based, publish/subscribe system that is designed for multi-tier cloud deployments (e.g. edge-cloud, multi-cloud, IoT-cloud, etc.). *Canal* implements a triggered computational (i.e. "serverless") programming model and provides developers with a uniform and portable programming interface. To achieve scalability and reliability, *Canal* combines the use of a distributed hash table (DHT) and replica consensus protocol to distribute and replicate functions, state, and data. *Canal* also decouples replica placement from the DHT topology to allow developers to optimize function placement for different objectives. We evaluate *Canal* using a real-world multi-tier IoT deployment and we use *Canal* to compare placement strategies, end-to-end performance, and failure recovery using both benchmarks and a real-world IoT-edge application. Our results show that *Canal* is able to achieve both low latency and reliability in this setting.**

***Index Terms*—publish/subscribe, serverless, data-driven applications, IoT, edge computing**

I. Introduction

With the emergence of the Internet of Things (IoT) and edge-based cloud computing, data-driven applications have become increasingly powerful. By leveraging IoT deployments across tiers (sensors-edge-cloud), developers are able to collect a wider range of sensor data and perform more advanced analytics and machine learning "near" where data is generated, while enabling decision support and intelligent actuation and control of a vast array of objects in the world around us.

Technologically, designers of IoT deployments often envision a scalable data management infrastructure in the form of a publish-subscribe (pub/sub) framework [1]–[3] that locates and matches data publishers with interested data subscribers. These systems provide discovery services and a higher-level, more abstract messaging pattern than direct point-to-point messaging or network multicast. Consumers express their interest by subscribing to data carrying certain attributes (content-based) or explicitly labeled by category (topic-based). Producers label data and forward it to data brokers, which implement data discovery by subscribers, and route data to them.

Pub/sub systems typically do not implement a computational model that supports processing "in stream." Developers must implement their own functionality in separate frameworks, manage failures, and optimize applications and deployments manually. This latter limitation is particularly challenging because pub/sub systems intentionally abstract away locality information associated with subscribers and publishers. Thus there is no way for developers to intelligently "place" their application functionality in ways that exploit locality for either reliability or performance optimization, both of which are critical for IoT applications that consume battery power or that require low-latency response at the edge.

To address these issues, we present *Canal*. *Canal* is a portably programmable, reliable, distributed, pub/sub system for multi-scale IoT deployments that includes both in-stream computational functionality and locality management capabilities. *Canal* implements the serverless computing programming model but adds to this model data discovery and reliable data persistence implemented via strongly-consistent replication. *Canal* developers implement applications as a set of independent functions (uploaded to the distributed system) that subscribe to data topics from producers. *Canal* invokes the functions asynchronously when data to which they have been subscribed becomes available.

To implement scalable and reliable data discovery and data persistence in the same framework, *Canal* couples a distributed hash table (DHT) for fast, robust lookup with consensus-based, strongly consistent data replication. *Canal* replicates topic data, application functions, and DHT state to facilitate fault tolerance efficiently. Developers use a uniform and portable programming interface (API) to implement functions, manage their placement, and control the performance/reliability trade-off as required by their applications.

We empirically evaluate *Canal* using multi-scale cloud environments consisting of edge clouds and a large-scale private cloud from a working IoT deployment currently in use. Our evaluation shows that applications can be easily deployed in *Canal* and tuned to meet different performance, availability, and reliability requirements.

II. Related Work

While pub/sub systems have been widely studied, their role in distributed and heterogeneous, IoT- and cloud-based settings has been the focus of recent research. MQTT is a lightweight, topic-based, pub/sub protocol designed for message exchange between devices. MQTT employs a centralized design with which publishers and subscribers connect via a data broker. A single broker server routes data from publishers to subscribers based on topic interest. MQTT-SN [1] extends of MQTT for

This work has been supported in part by NSF grants CNS-1703560, CCF-1539586, and ACI-1541215.

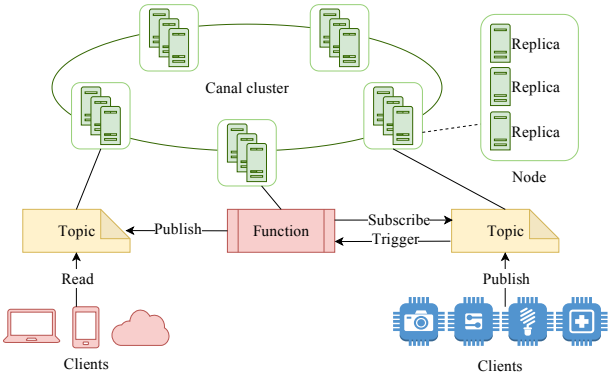


Fig. 1: The system overview of *Canal*.

use by sensor networks. To reduce power consumption for messaging, MQTT-SN decreases message payload and topic name size, and uses UDP to transfer messages.

RabbitMQ [3] is a broker system that supports MQTT client. In RabbitMQ, brokers host "exchanges" that route messages to in-memory message queues for consumption. RabbitMQ provides high availability and reliability via replication. Apache Kafka [2] is a data stream processing platform that focuses on maximizing throughput. Kafka partitions and replicates topics to tolerate faults and balance load. Flume [4] and Facebook Scribe [5] are similar systems that manage log data.

The paradigm of integrating data processing capabilities into brokers is sometimes referred to publish-process-subscribe pattern [6], [7]. A common strategy for implementing this pattern leverages external systems such as Apache Storm, Spark, or Flink [8] to extend pub/sub with data processing capabilities. Such linking unfortunately requires that both data ingress (e.g. Kafka) and processing be deployed, managed, and configured separately, which is complex and error prone. *Canal* is unique in that we co-design and seamlessly couple messaging, portable data-driven computation, and data persistence within a single wide-area distributed system.

Hermes [9] and Scribe [10] are peer-to-peer pub/sub systems. They use a distributed hash table (DHT) to cluster brokers and handle more data streams. P2S [11] is similar but uses Paxos to replicate broker data for fault tolerance and availability. DHT protocols such as Pastry [12] and Tapestry [13] approach the locality problem via routing mechanisms. The routing mechanisms in these systems increase the probability of routing distance minimization between nodes that are "near-by" in terms of network latency. While sharing the same motivation with these systems, *Canal* differs in that it allows users to specify the data placement and replication strategy.

III. Design and Implementation

Canal implements topic-based publish/subscribe messaging, event-triggered computation, distributed discovery, and strongly-consistent data replication. Fig. 1 illustrates the *Canal* system. A *Canal* deployment is a cluster of geo-distributed, virtual nodes organized as a DHT to facilitate distributed lookup. Nodes service client requests and host topics that are published by clients. *Canal* routes published data to subscribers and exports an API that clients use to create, access, and publish data according to topic.

Canal is unique in that it also embeds "in-stream" distributed application execution to automate data-driven response, analysis, and computation. Specifically, *Canal* implements the serverless programming and deployment model (also known as function-as-a-service) [14], [15]. The serverless model is well suited to our setting in that it is event-based, scalable, and tolerant to intermittent connectivity. Using this model, programmers write simple event handlers (i.e., application functions), which are invoked by the platform in response to an event.

In *Canal*, events correspond to topic publish operations. Developers use *Canal* function templates to construct their applications and the *Canal* client API to register and deploy their applications. Functions can access topic data, process and analyze data from multiple topics, and publish new data to geo-distributed topics, when they execute. That is, *Canal couples serverless FaaS programming with pub/sub for data management to create a new, high-level, scalable programming technology for IoT*. To do so, *Canal* defines the following core system abstractions.

- **Topic:** A topic is an object that encapsulates data elements of the same type. Each element in a topic is identified by a strictly increasing index.
- **Function:** A function is an event handler implemented by an application that *Canal* executes in response to a publish event on a topic to which it has subscribed.
- **Node:** A node consists of one or more physical or virtual machines, which service client requests and host topics that are published and subscribed to by applications and clients.
- **Replica:** A replica is a copy of a topic and its functions. For redundancy and parallelism, each node can implement multiple replicas.

Each topic in *Canal* is implemented as an append-only log of data values that is fixed-size and automatically aged and garbage collected. Past data can be retrieved for processing (up to the configurable length of the log history); append-only storage also reduces contention overhead and enhances data durability since data elements within a topic are immutable. *Canal* maintains the list of functions subscribed to each topic and triggers all subscribed functions when data is published to a topic.

Functions can retrieve the most recent version of the topic or a previous version, if available, via its index. This index is returned upon a publish (an append) and the history can be scanned using it via the client API. We describe the *Canal* storage and replication model in more detail in Section III-B.

Topics and functions can be placed on any node in a *Canal* cluster. This flexibility enables developers to optimize their applications in different ways, e.g. for locality (co-locating functions and topics) or for resilience (geo-distributing replicas). For example, a machine learning application that subscribes to IoT devices for its input data may wish to place

the IoT topics near where the data are generated, and place the functions that perform more heavyweight computation on more powerful nodes at the edge or in the cloud.

To enhance availability and reliability, a node can also implement multiple replicas. All the topics and functions as well as the topic lookup entries stored in a node are replicated. A group of replicas in a node uses a consensus protocol to elect a leader. The leader is responsible for all the queries and requests from other nodes. If the leader fails, a new leader is elected among the remaining working replicas and assumes the leader role. We detail how *Canal* manages and uses data replicas in Section III-B.

Beyond hosting data and functions, nodes handle client queries and publish requests. Clients specify topics, which *Canal* uses to identify hosting nodes using distributed lookup.

A. The Canal Runtime and Lookup System

For serverless computation and persistent logs, *Canal* uses *CSPOT*, an open source, distributed serverless runtime system that executes event-triggered functions across multi-scale, heterogeneous devices without modification [16]. *CSPOT* implements FaaS semantics coupled with an intrinsic log data structure that is storage persistent.

Canal achieves scalable, distributed lookup (for clusters in which nodes can join and leave dynamically) by extending protocols from *Chord* [17] using CSPOT. Chord is a scalable peer-to-peer lookup protocol that implements a distributed hash table (DHT). It provides one simple operation – mapping a key and its value to a node – but it does so in a way that is robust with respect to wide-area network and node failure.

The topology of the DHT cluster is structured as a ring. Each node in a Chord ring is equivalent to a *Canal* node and *Canal* uses the Chord protocol to maintain the integrity of the ring. When joining a cluster, each node gets an m-bit identifier by hashing the unique name or IP address of the node using SHA-1.The protocol places the node, ordered by its identifier, on a ring with a circular key space of 2^m. When inserting or querying about a key, the key is hashed again with SHA-1 to compute its identifier k. The protocol uses the key's identifier to decide which node is responsible for storing the key-value pair. Once the identifier is acquired, the protocol forwards the query to the node whose identifier is equal to or immediately after k, i.e. the successor of k (denoted by $successor(k)$).

For scalability, a node maintains only a subset of successor addresses in an address table. A node with identifier n maintains the addresses of its immediate successors (we use three herein). Each node also maintains m pointers (called fingers) to other nodes in the ring. The ith finger in an address table is denoted as $finger(i)$ and it is defined by $sucessor(n+2^{i-1})$. Upon receiving a query of target key k, a node n checks if its immediate successor is the successor of k by testing if $k \in (n, successor]$. If so, it answers the query with the successor's address, where the value of key k can be found. If the node's immediate successor is not the successor of k, the node passes the query to the closest preceding node of k in the finger list. The query is passed along the ring in this way until it reaches a node who can answer the query. This protocol is proven to be scalable such that in a Chord cluster with N nodes, each node need only store $O(log(N))$ addresses, and a query can be answered in $O(log(N))$ communications.

Chord maintains the ring relationship between nodes using a *stabilization protocol*. When node n joins the cluster, it queries the cluster to find its own $successor$. Once the query is answered, *Canal* inserts node n into the ring ahead of $successor(n)$ by notifying $successor(n)$ about its new successor. After being notified, $successor(n)$ adds node n as its $predecessor$. The next time $successor(n)$'s old predecessor sends a heartbeat to $successor(n)$, it learns that a new node was inserted and updates its successor from $successor(n)$ to node n, which completes the "stabilization" process. Each node in the ring sends a heartbeat message to its successor. When node n leaves the cluster, its predecessor $predecessor(n)$ learns this fact via the failed attempt to send the heartbeat. In this case, $predecessor(n)$ uses the next successor as its immediate successor and sends a heartbeat to its new successor. When $successor(n)$ receives the heartbeat from $predecessor(n)$, it learns that n has left and updates its predecessor to $predecessor(n)$.

When a topic is created, *Canal* hashes the topic name for use as a lookup key. The address of the node that hosts the topic is stored in the successor of the topic name hash. This design allows *Canal* to decouple data placement from address lookup, making application deployment strategies more flexible while achieving a fast, logarithmic querying time.

B. Replication and the Canal Data Model

Chord's DHT design provides a simple way to replicate data for fault tolerance. When a key-value pair is stored in a node, it is replicated to immediate successors of the node. Because nodes are connected and ordered as a ring, when a node leaves the cluster unwillingly due to failure, its successor will assume the node's position on the ring and take over the responsibility of the departing node. When the departing node's successor does so, it has a data replica and thus it can serve queries seamlessly without performing any data migration. *Canal* uses this mechanism to replicate the address information and subscription lists for topics.

However, this mechanism cannot support the data in topics or application functions. On a DHT ring, nodes are randomly ordered by their identifier hash. In essence, replicating data to a node's successors is replicating data to random nodes in the cluster geographically. This works well for replicating lightweight key-value pairs such as topic address and subscription list, but it is inefficient for large data objects. In addition, random replication precludes exploiting data locality.

Our approach extends RAFT [18] (a consensus algorithm for implementing strong consistency) with locality optimizations and then uses RAFT as the storage mechanism for managing the DHT state *and* topic data. *Canal* also extends the Chord protocols to take advantage of the RAFT locality optimizations. RAFT uses a consensus algorithm and leader election to manage a replicated log among the members of a

"RAFT cluster." Our approach to using RAFT for scalable data persistence equates each *Canal* node in the DHT with a RAFT cluster. Thus, unlike in the original Chord implementation where each DHT node is identified by network address, our approach encodes RAFT replica information into each Chord *successor*, *predecessor*, and *finger* node identifier. We then modify the Chord protocol to use this information to access persistent state from the most up-to-date RAFT replica.

C. Canal's DHT Implementation

RAFT provides a simple leader election algorithm to guarantee that there is only one leader at any time. Only the leader responds to client requests and appends entries to its log. RAFT implements a strong leader approach in that the leader actively replicates its log to the other members (followers). If a follower receives conflicting entries from the leader, it overwrites its own log with the entries from the leader. Followers never ask the leader to resolve the conflict and the leader never rewrites its own log. Once the leader confirms a log entry is replicated to a majority of replicas, it commits the entry and notifies its followers.

We divide time into RAFT *terms* across each RAFT cluster (recall that there is one RAFT cluster defined for each *Canal* node in the *Canal* DHT), and define a random timeout for each. If a follower does not hear from the leader by the timeout, the follower promotes itself as a leader candidate and starts a new term. The candidate starts an election by requesting votes from all the members in its cluster. According to the RAFT protocol, each member can only vote for one candidate in each term, and it can only vote for a candidate if the candidate has more up-to-date entries than its own. If a candidate's vote requests are accepted by the majority of members (itself included), it is promoted and becomes the leader. This algorithm guarantees that only members with most up-to-date log entry will be elected as new leader, so that the committed log entries are never rewritten. After an election is complete, each replica in the RAFT cluster agrees on the leader for the cluster.

All DHT queries and client requests to publish to a topic or read data must be answered by the RAFT leader that represents the *Canal* node during the current *term*. *Canal* records the replica host addresses for each host in the RAFT cluster associated with each *Canal* node. When a *Canal* node joins the DHT ring and triggers the stabilization process, it is updated with the RAFT replica host addresses associated with its *successor* in the DHT ring, and similarly, with the replica addresses associated with its *predecessor*. It then queries one replica in the *successor* RAFT cluster to determine the current leader to be used whenever that *successor* is contacted when implementing the Chord protocol. Similarly, it determines the current leader for its *predecessor* and the *Canal* nodes listed in its finger table.

If a leader fails, the node becomes unavailable temporarily until a new leader is elected among working RAFT replicas. Any attempt to contact RAFT replica during leader election stalls until the election is complete. A *Canal* node discovers a new leader in another *Canal* node's RAFT cluster when an attempt to contact that leader fails. In the event that a *Canal* node fails to contact the leader of another *Canal* node's RAFT cluster, it will query the other replicas in that cluster to determine the current (possibly new) leader. If leader election is in progress, the discovery request stalls and may eventually time out. Otherwise, the replica responds with the current leader's host address and the *Canal* node uses that address as the new leader until the next leader change.

Additionally, *Canal* uses RAFT to replicate each *Canal* node's DHT *successor* list as a way of ensuring that a RAFT leadership change does not trigger a DHT ring restabilization. The *successor* list changes only when *Canal* nodes join or leave the DHT ring. By replicating this state, a *Canal* node can always discover the current leader of its *successor* RAFT cluster once the ring is stabilized.

In the case when the majority of replicas fail, the node leaves the *Canal* cluster and *Canal* triggers a DHT restabilization. When/If the node recovers from failure and becomes available again, it will try to send a heartbeat to its last known successor. This will also cause *Canal* to initiate DHT ring stabilization, which will enable the node to rejoin the cluster.

D. Canal Data Management

Since a node can host multiple topics, *Canal* uses a single data object to store the data from all topics the node hosts. However, as described previously, the persistent objects in our serverless runtime are append-only logs. To support this and multi-topic objects (that are replicated), *Canal* employs a mapping object per topic. This object records the index of the data element when published, in the multi-topic data object. When a data element is published to a topic, *Canal* encapsulates it with its topic name and appends to the multi-topic data object. When the entry is replicated and committed, each committing replica checks which topic the entry belongs to and writes the index of the committed entry to the topic's mapping object. When a client reads data in the topic by index, *Canal* first checks the topic's mapping object to get the index of the data element in the multi-topic data object. It then reads and returns the data element from the data object. Only the multi-topic data object is replicated (i.e. managed) by the RAFT algorithm. Mapping objects are indirectly (and automatically) replicated since each replica updates its mapping object when a data element is committed.

E. Canal API and Data Publishing

Canal exports a simple client API for publish/subscribe operations. The client API provides the following functions:

- ***create_topic(topic_name):***
  Create a topic *topic_name* on the calling node.
- ***subscribe(topic_name, function_name):***
  Subscribe the function *function_name* on the calling node to the topic *topic_name*.
- ***publish(node, topic_name, data):***
  Publish data to *topic_name*.

- ***latest_index(node, topic_name):***
  Get the latest index of published data in *topic name*.
- ***get(node, topic_name, index):***
  Get the data in *topic name* by its index.

A node in a *Canal* cluster calls *create_topic()* to create a topic and store its address in the DHT. *create_topic()* also creates the topic mapping object as described in the previous section. A node can also call *subscribe()* to subscribe a function to a topic. There must be a binary file named *function_name* on the calling node for the call to succeed. The binary function can be written and compiled using the provided toolchain and it should follow the *Canal* function template: *function name(topic name, index, data)* . *function_name* can be any UNIX-style filename. *topic_name*, *index*, and *data* will be filled in by the *Canal* runtime when the function is triggered.

Any client can use *publish()*, *latest_index()*, and *get()* to publish or read data. All of these operations take an optional *node* parameter to specify the address of the node that should receive the request. The node address can be the address of a leader or follower replica of any node in the *Canal* cluster. It does not have to be the node hosting the topic, as the topic name will be used to query the actual address of the topic. The *node* parameter can be omitted if the calling client is a node itself in the *Canal* cluster.

When a client requests a node to publish data to a topic, the node (referred to as the publishing node) queries the *Canal* ring by topic name to determine where the topic data are hosted; it uses this information to acquire the subscription list for the topic. The query is answered with the addresses of all the replicas of the node hosting the topic (the topic node). The publishing node then appends the data element to the multi-topic data object of the topic node as described in the previous section. Once the data element is committed using the RAFT algorithm, the topic node notifies the publishing node and the later triggers the functions on the subscription list to consume the published data.

IV. Evaluation

In this section, we evaluate the performance of *Canal* using different replica placement strategies. Specifically, we instrument its processing pipeline (which include Chord and RAFT extensions) and employ a series of benchmarks and a wide range of testing configurations using the resources from a working IoT deployment.

We deploy *Canal* clusters with different configurations using virtual machine (VM) instances on three private and edge clouds, called *Campus*, *Lab*, and *Farm*. Each cloud runs Eucalyptus v4.2.2, which is API-compatible with AWS EC2 and S3. *Campus* is a shared, private cloud located on the UCSB campus connected via 10Gb Ethernet. It is part multi-campus cloud federation [19] that consists of 1400 cores and 30TBs of storage. *Lab* is an edge cloud located in the authors' research lab that consists of 9 Intel Next Units of Computing (NUC) devices connected via a gigabit switch. *Farm* is also a NUC-based edge cloud sited on a remote research reserve roughly

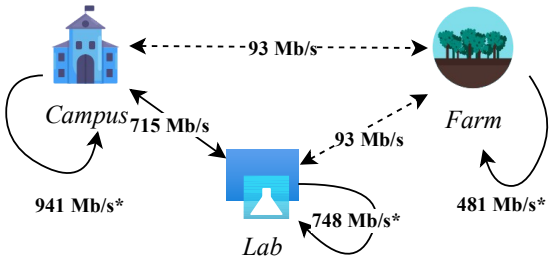


Fig. 2: Bandwidth across our multi-cloud, experimental deployment as measured by iperf. *'d measurements show the bandwidth between instances within the same cloud.

50 miles from the campus. Locally, the NUCs that comprise the *Farm* cloud are interconnected by a gigabit switch (with far less bisection bandwidth than *Lab*) but traffic to and from the site must traverse a 100 Mb (and often lossy) microwave link connecting the site to campus.

Each VM instance has 2 cores; *Campus* instances have 1GB memory while *Lab* and *Farm* instances have 4GB memory. During our experiments we observe that *Canal* uses very little memory and, as such, instance memory size does not significantly impact performance or our results. This deployment consisting of two edge clouds (*Farm* at the remote site and *Lab* on campus) and a large private cloud that is used to process camera trap wildlife images. Fig. 2 shows the average bandwidth between clouds in this study, as measured by iperf. For our experiments, we configure *Canal* with 8 *Canal* nodes. Each *Canal* node implements a RAFT cluster with 3 replicas. Each RAFT replica is hosted by a single cloud VM (i.e. the full deployment is 24 VMs.)

Note that co-locating geographically all of the RAFT replicas within the same cloud associated with any single *Canal* node within the same cloud speeds the RAFT consensus protocol, but creates the possibility for DHT ring churn. In particular, when the microwave link fails, the *Farm* cloud is completely disconnected, thus all *Canal* nodes completely hosted in that cloud drop from the ring and *Canal* restabilizes using the nodes hosted in *Lab* and *Campus*. Alternatively, when the replicas are geographically distributed, the loss of all nodes at one site (typically the *Farm* cloud due to the failure of the microwave link) does *not* cause a ring restabilization. Rather, after leader election for any leaders that were lost, the system continues using the remaining replicas without a Chord stabilization phase. We investigate this trade-off more quantitatively in Subsection IV-E.

Thus we focus on two placement strategies. The *colocated* strategy puts all RAFT replicas associated with a *Canal* node in the same cloud. Alternatively, the *distributed* strategy places one replica from each *Canal* node in each of the 3 clouds. In this way, we can detail the trade-off between DHT restabilization overhead inherent in the *colocated* strategy with delays introduced by RAFT leader-election prevalent in the *distributed* strategy.

Moreover, since the leader of RAFT cluster is in charge of replicating data, handling Chord queries, and function subscription, *Canal* is more sensitive to RAFT leader failure

than follower failure. We have extended RAFT for *Canal* so that it will elect leaders so that it is locality-aware as a way of introducing locality control into the protocol. Using this *Canal* feature we compare the *distributed* *Canal* strategy when all of the RAFT leaders are in the *Lab* cloud (denoted $distributed_{Lab}$) to putting all of them in the less reliable *Farm* cloud (denoted $distributed_{Farm}$). We omit the other configuration combinations due to space constraints.

A. Publish Benchmark

To evaluate the performance of *Canal*, we implement a simple benchmark using the *Canal* client API. We start a *Canal* cluster and create a test topic of size 8KB on one node (referred as the topic node). The data published to the test topic is replicated among the topic node's three replicas, either *distributed* or *colocated* depending on the configuration. We create a test function that records a timestamp and returns, which we subscribe to the test topic. The function is triggered (recording a timestamp) each time new data is published to the test topic.

We use a simple client, which calls *publish*(), to publish data to the test topic at a fixed rate. We experiment with different publishing rates and run each test for 10 minutes. Finally, in the following experiments, we place the client, topic, and function on the same node.

Fig. 3a shows the average publishing latency for different publishing rates. Surprisingly, distributing the replicas across the wide-area network links (including the slow microwave link) degrades publication performance only slightly compared to colocating them. However, the much slower internal network switch in the *Farm* introduces a large negative performance impact when the RAFT leaders are hosted there, as illustrated in Fig. 3b.

The difference shown in these graphs indicates the importance of locality in an IoT setting. In particular, without the ability to determine a preference for RAFT leader placement (one of the enhancements to RAFT that *Canal* makes), the relative publication throughput difference is dramatic.

Fig. 4 shows the latency a client experiences when calling *publish*(), waiting for data replication and the subscribing function to complete (i.e. the earliest moment after publication that the data is available to a subscribing function). The latencies when the workload is light (API functions are called and measured sequentially without overlapping) and when the benchmark is publishing data at the cluster's maximum publishing rate are both shown.

This experiment shows that when the RAFT leader is in a cloud with high availability and connected with a high performance network (*Lab*), the colocated configuration has a slightly lower latency than its distributed counterpart. At the maximum publishing rate, the publishing latencies of $colocated_{Lab}$ is 616 milliseconds, 175 milliseconds lower than the $distributed_{Lab}$ configuration (791 ms). However, when the RAFT leader is sited in a less capable edge cloud (*Farm*), $colocated_{Farm}$ does not achieve the same advantage. In the $colocated_{Farm}$ configuration, because all replicas are under stress from internal *Canal* function invocations, replication, and request responses, the latency is higher than in $distributed_{Farm}$ configuration, where followers do not slow down the overall system.

B. Subscriber Latency

We next evaluate the latency experienced by a subscriber when extracting data from a topic. With the same setup, this experiment sequentially calls *get*() on the test topic 10000 times. We compute average *get*() latency as the total time of benchmark divided by the 10000 which we present in Table I. The results indicate that the performance of *get*() is affected only by the network performance and not by the data publishing rate. In the best case where the client and the topic leader reside on the same instance, it takes 10 milliseconds on average to get data. When a slow network involved (e.g. *Farm* cloud), the worst case is more than 50 milliseconds on average for a get to complete.

TABLE I: *get*() latency

Leader location	Client location		
	Campus	*Lab*	*Farm*
Campus	23 ms *	45 ms	55 ms
Lab		10 ms *	50 ms
Farm			25 ms *

* Client and topic leader is on the same instance.

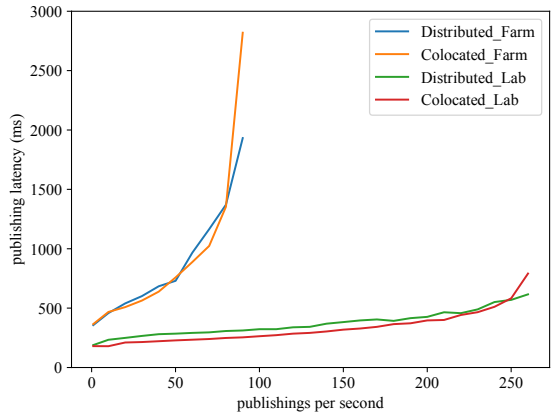
C. Data Availability

Recall that each *Canal* node is functional as long as a majority of its RAFT replicas are available and connected (we do not yet have a protocol in place for accessing and managing stale data in disconnected replicas). As Fig. 3a shows, replica placement can have a dramatic effect on throughput. In this subsection we look at the effect of placement on data availability.

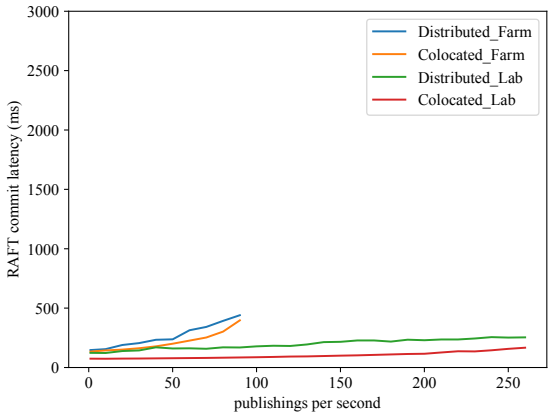
To do so, we analyze a trace of a working application [1] and hypothesize that there are two clouds: a public cloud aand an edge cloud. We artificially induce failures into the trace and observe how many of the requests (either publication or subscription) cannot be satisfied. We use 99.99% availability for the public cloud (taken from typical public cloud SLAs [21]) and vary the availability of the edge cloud. Note that these error rates grossly overestimate true cloud availability in all cases. Thus our results are conservative.

With all replicas located in the public cloud, the application would experience no failures even with only 99.99% availability. Putting 1 replica in on the edge (to improve publication latency) still results in 99% of the application requests succeeding even when the edge cloud's availability is on 90% but results in all requests being satisfied when the edge is 98% available – far less available than in practice. Thus the geodistributed strategy described previously both yields good performance and excellent data availability.

[1] We implement a temperature prediction application described in [20] to evaluate *Canal* in real world scenario. The source code of *Canal* and evaluation application can be found at https://github.com/MAYHEM-Lab/cspot/.



(a) Average publishing latency per data publishing



(b) Average RAFT commit latency per log entry

Fig. 3: Publishing latency

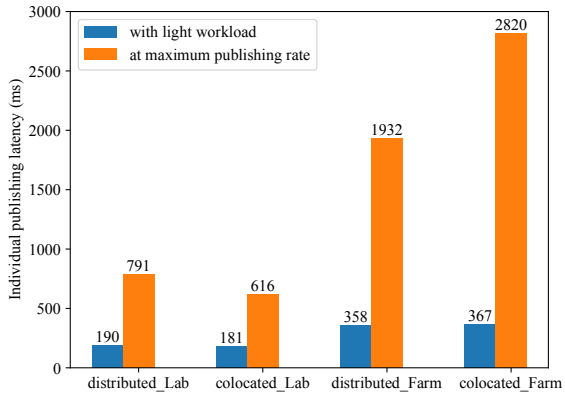


Fig. 4: Lowest publishing latency for each individual publishing in different cluster configurations

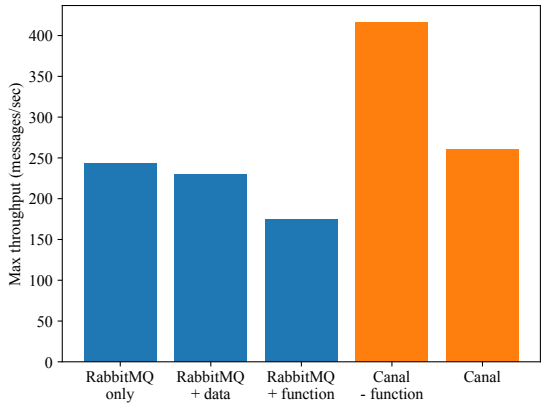


Fig. 5: The throughput comparison of RabbitMQ with MQTT and *Canal*.

D. Comparing Canal to MQTT

We next compare the performance of *Canal* to MQTT, the most commonly used topic-based publish/subscribe protocol in IoT. To setup a wide area distributed environment, we use RabbitMQ with MQTT client to deploy a broker network. RabbitMQ enhances the availability and reliability of MQTT brokers by supporting *clustering* and *federation*. A RabbitMQ cluster is a group of brokers whose status is replicated using the RAFT algorithm. When the leading broker fails, another broker is elected and takes over. A federation is formed via multiple clusters in a wide area. Once registered, clusters share and access the topics in other clusters in the federation.

RabbitMQ recommends forming a cluster with a group of brokers connected with strong network. In the following test, we use six *Campus* instances to create two clusters (three instances per cluster) and form a federation of two clusters. We use Paho asynchronous MQTT library to implement consumer and producer clients in C. The consumer client is placed in one cluster, while the producer client is in the other. QoS level is set to 1 (at least once delivery) as RabbitMQ does not support QoS level 2 (note that *Canal* supports QoS level 2 in these experiments). The consumer continuously consumes messages sent by the producer and the producer continuously publishes of fixed size (8KB). To determine the maximum throughput, we increase the number of messages the producer publishes per second until RabbitMQ starts buffering.

We compare the throughput of RabbitMQ 3.8.14 with MQTT clients to that of our system. Since RabbitMQ does not persist the data once the message is delivered, a process failure causes data loss. Thus we use CSPOT's append-only data object to persist messages to storage once they are received by the consumer client. When a consumer client receives a message, it writes it to all brokers' CSPOT objects. Since RabbitMQ only supports QoS level 1, we do not replicate the data. However, it does show how data persistence affects the performance. In addition, even though one can implement application logic in consumer client, we trigger a separate function for each received message to measure the performance impact of event-based (i.e. serverless) function invocation.

Fig. 5 shows the throughput comparison. Without persisting data or triggering functions, RabbitMQ can route 243 messages per second. When storing data to CSPOT to increase data

reliability, it can route 230 messages per second. If triggering functions, it can handle 175 messages per second. Compared to RabbitMQ federation, *Canal* can process 260 messages per second using the same instances and network, with data persistence (via CSPOT). If we remove the function triggering from *Canal*, it can route 416 messages per second to the topic. Since data persistence is core to the *Canal* design and implementation, we are not able to strip it for comparison.

E. Failure Recovery

As described previously, *Canal* uses RAFT replicas to avoid Chord ring restabilization. To determine the relative performance of each approach, we deployed *Canal* and Chord in the Campus cloud (to maximize Chord's performance during restabilization which is dominated by message latency) and compare Chord restabilization to *Canal* leader discovery after a node loss. The average *Canal* discovery time over 1000 artificially induced node failures is 667 milliseconds compared to an average of 1627 milliseconds for each Chord restabilization.

Moreover, Chord relies on randomized node placement in the DHT ring to avoid the effects of correlated failure. Each node requires a "chain" of successors so that it can reestablish the ring during stabilization. However, if all nodes in the chain are disconnected or down (e.g. they are all within a single geographic location that has become disconnected from the other sites), Chord cannot restabilize. With 3 sites and 8 nodes per site in our setting, each DHT node would require 9 successors in the chain to guarantee that at least one successor is located in a geographically remote location. In the experiment described above, we configured each DHT node with a chain of length 1 (to maximize Chord performance) however the chain needs to be rebuilt during restabilization thereby increasing the time to stability as a function of its length. Thus a pure Chord implementation in our setting would likely be significantly slower.

V. Conclusions and Future Work

In this paper, we present a distributed publish/subscribe system called *Canal*, that is straightforward to program and that is reliable and performant in IoT settings. Like a traditional topic-based publish/subscribe system, *Canal* supports data delivery by topic subscription. Unique to *Canal*, consumers of topics are triggered functions that react to data and perform a series of tasks. *Canal* uses a DHT as a scalable pub/sub brokerage for data stored in RAFT clusters that replicate the data. Through modifications to the Chord DHT and RAFT protocols, we enable developers to use this architecture to exploit locality in distributed IoT settings. We evaluate the system using benchmarks and a real-world IoT application in a production IoT deployment. Our work shows that *Canal* is fast, stable, and reliable.

As part of future directions, we are extending the system with intelligent load balancing and high-level language support for functions. We are also working on "autotuning" capabilities to optimize *Canal* performance and reliability in each deployment.

References

[1] A. Stanford-Clark and H. Truong, "MQTT for sensor networks (MQTT-S) protocol specification," *International Business Machines Corporation version*, vol. 1, 2008.

[2] "Apache Kafka," 2019, http://kafka.apache.org [Online; accessed Sep. 2019].

[3] "RabbitMQ," https://www.rabbitmq.com [Online; accessed 1-Nov-2016].

[4] "Apache Flume," http://flume.apache.org/, [Online; accessed 11-Mar-2021].

[5] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime data processing at facebook," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1087–1098. [Online]. Available: https://doi.org/10.1145/2882903.2904441

[6] H. Ristau, "Publish/process/subscribe: Message based communication for smart environments," in *IET*, 2008.

[7] B. Krishnamachari and K. L. Wright, "The publish-process-subscribe paradigm for the internet of things," USC ANRG, Tech. Rep. 2017-04, 2017.

[8] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1789–1792.

[9] P. R. Pietzuch and J. M. Bacon, "Hermes: a distributed event-based middleware architecture," in *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, 2002, pp. 611–618.

[10] M. Castro, P. Druschel, A. . Kermarrec, and A. I. T. Rowstron, "Scribe: a large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1489–1499, 2002.

[11] T. Chang, S. Duan, H. Meling, S. Peisert, and H. Zhang, "P2s: A fault-tolerant publish/subscribe infrastructure," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 189–197. [Online]. Available: https://doi.org/10.1145/2611286.2611305

[12] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware*, R. Guerraoui, Ed., 2001.

[13] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. Joseph, and J. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 41–53, 2004.

[14] "FaaS, PaaS, and the Benefits of the Serverless Architecture," https://www.infoq.com/news/2016/06/faas-serverless-architecture [Online; accessed 1-Nov-2016].

[15] "Function as a Service," https://en.wikipedia.org/wiki/Function as a _ _ Service [Online; accessed 12-Sep-2017].

[16] R. Wolski, C. Krintz, F. Bakir, G. George, and W.-T. Lin, "CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT," in *ACM Symposium on Edge Computing*, 2019.

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2001.

[18] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[19] "Aristotle Cloud Federation," https://federatedcloud.org [Online; accessed 12-Sep-2017].

[20] N. Golubovic, R. Wolski, C. Krintz, and M. Mock, "Improving the Accuracy of Outdoor Temperature Prediction by IoT Devices," in *IEEE Conference on IoT*, 2019.

[21] "Amazon Compute Service Level Agreement," https://aws.amazon.com/compute/sla/, [Online; accessed 8-Apr-2021].