Continuous Query-Based Syndication: Distributed, Expressive Messaging for the IoT

Gabriel Fierro
gt.fierro@berkeley.edu

Erik Krogen
erikkrogen@berkeley.edu

***Abstract*—Applications in the Internet of Things exist at a confluence of semantically isolated networks over buildings, physical spaces, cloud services, smart appliances and mobile and wearable devices. The utility of these applications is in their ability to capture capabilities of new families of smart, networked devices and integrate them with existing networks surrounding people, things and places. We present a novel publish-subscribe mechanism—continuous query-based syndication (CQBS)—that allows subscribers to richly describe the set of resources they require and maintain a consistent view of relevant data sources even as they change over time. We implement a highly available, distributed CQBS broker that uses a replicated coordinator to provide simple failover for embedded clients. Through benchmarks, we demonstrate that the system maintains reasonable 95th percentile latencies of under $10ms$ even under load.**

I. Introduction

The context of the Internet of Things has seen an increase in both the number and capabilities of small, low-powered, constrained devices wanting to interact with each other and the outside world, raising the question of how to conduct discovery and communication.

Ensembles of networked "things" interacting with dynamic applications require rich descriptive power to promote discovery across heterogeneous devices and services, which should be reflected in their communication mechanism. They also require the ability to react to changes in the layout or configuration of devices, whether these changes are generated by the device or by a human administrator. One of the characteristics that distinguishes the Internet of Things from prior collections of networked things is the higher number of devices in a space and the frequency with which those devices may enter or leave a space. Applications or services may need to react to the entrance or exit of a device or user. Discovery mechanisms that operate at discrete intervals—request-response or periodic advertisements—expect a delay proportional to the interval length.

Publish-subscribe (pub-sub) is an attractive approach because it decouples publishers from subscribers in space: communication through a well-known broker helps deal with firewalls and NATs and reduces load on popular publishers. There are two dominant "flavors" of pub-sub—topic-based and content-based—that traditionally identify tradeoffs between performance and expressiveness. In topic-based pub-sub systems, messages are published to logical channels that may be in a flat or hierarchical namespace, and subscribers identify a name or "glob" that matches topics. The benefits are that matching is typically fast and message overhead is small, but the expressive power of a "topic" is limited. In content-based pub-sub, subscribers specify predicates, which act as filters for incoming messages for publishers. While this scheme has richer descriptive power, it requires larger messages and computationally intensive brokers.

We propose an alternative pub-sub mechanism, continuous query-based syndication (CQBS), in which subscriptions are defined by SQL-style queries over publisher descriptions. Publisher descriptions contain "metadata" defining properties of the device, e.g. location, units of measure, groupings, etc. that describe the context and configuration of the producer. These queries are continuously evaluated to reflect the current configuration of all publishers, enabling subscribers to always receive messages relevant to their query even as the landscape of publishers changes.

In this paper, we present the design and implementation of a distributed CQBS broker, a message broker for service composition that uses *continuous query-based syndication* to enable dynamic and contextually-aware applications and services for the Internet of Things. The system is intended to serve as a ubiquitous message bus connecting an array of embedded, networked devices with a collection of distributed applications. In order to be effective, the design must meet the following requirements:

- High Availability: IoT applications must be robust to the inevitable hardware and network failures at scale. We wish to create a system that is resilient in the face of arbitrary machine failures.
- Scalability: Applications will want to connect across rooms, buildings or even cities, requiring a large number of ingress points. The system should be able to scale to large numbers of clients with reasonably high message rates.
- Simple Clients: The code necessary for a client to interact with the system should be very simple, since we assume that they may be embedded devices with limited programming facilities and computational resources.

The system as described in this paper is implemented in Go and is available online as free, open-source software at https://github.com/gtfierro/cs262-project.

II. Related Work

The CQBS distributed broker is most closely related to *publish-subscribe* systems. We examine each of the systems

below along the primary design dimensions of the CQBS broker: richness of publisher descriptions, expressiveness of syndication model, client complexity and fault tolerance. A summary of these systems and their properties can be seen in Table I.

A. Topic-Based Publish-Subscribe

Most publish-subscribe ("pub-sub") systems fall into one of two categories: topic-based and content based [9]. The most basic form of topic-based pub-sub is a channel model, in which producers (data publishers) transmit data associated with some channel name to a broker; subscribers list the channels in which they are interested. The benefits of this approach are its simplicity and speed—the "hot path" of a published message simply retrieves a list of subscribers—but each publisher is limited in expressive power to a single dimension (the name of the channel) [1]. Because of these limitations, some modern pub-sub systems use hierarchical topics with prefix and suffix matching using wildcards. The most popular of these are MQTT [2], Kafka [5] and XMPP [4].

MQTT is a lightweight publish-subscribe protocol popular for its simpicity and extensible messages. Producers of data in MQTT publish on hierarchical path-like topics such as `/apartment/gabe/livingroom/temperature`. This construction of topics is best for grouping publishers together along a limited number of dimensions, but quickly becomes unwieldy as the dimensionality and sparsity of the descriptions increase. For example, a temperature sensor could be described along the following dimensions:

- manufacturer and model number
- city, campus, building, floor and room number
- orientation or position within a room
- accuracy and precision of temperature sensor
- method of temperature sensing (e.g. IR, thermopile)
- who installed the temperature sensor and when it was installed
- sample rate of the temperature sensor

To be effective, a topic-based system must determine the order and syntax of topics so that subscribers can know they are consuming the appropriate streams. This is further complicated when considering other types of publishers which may include an entirely different set of descriptive tags. MQTT syndication supports prefix matching on topics and limited forms of suffix matching. To subscribe, applications specify explicit topics (`a/b/c/d`), one-level wildcards (`+/b/c/d`, `a/+/c/d`, `a/+/+/d`, `a/b/c/+`) and multi-level wildcards (`#`, `a/#`, `+/b/c/#`). While appropriate for basic subscriptions, this approach does not allow the expression of more complex predicates that contain "and", "or" or "not" relations: temperature sensors in Gabe's apartment, but not the ones in the kitchen or the bedroom.

MQTT supports three Quality-of-Service levels for message delivery: at most once, at least once, and exactly once. Most client implementations simply address the first two, keeping complexity and code-size down; it is entirely feasible to implement a MQTT client on an embedded device, and there exists an adaptation of MQTT (MQTT-S [3]) for non-TCP/IP networks. MQTT does not contain any explicit fault tolerance mechanisms: brokers may be distributed and "bridged" with the aid of an administrator, but failover logic is entirely implementation dependent. In summary, MQTT is insufficient for our goals because hierarchical topics are fundamentally constraining in their structure and syndication.

XMPP [4], or the Extensible Messaging and Presence Protocol, is an XML-based technology for instant messaging, video conferencing and more recently sensor communication (as seen in large, deployed systems such as Sensor Andrew [10]). Similar to channel-based systems, every entity (publisher or subscriber or broker) in a distribution of federated XMPP servers has some unique address from which it can send and receive messages. There are attempts to provide a more expressive pub-sub model on top of XMPP that allows for the discovery of services and subsequent subscription to relevant service providers [11]. These service descriptions can be extended to include arbitrary contextual data, which is a definite advantage over primitive address-based messaging.

The limitation of XMPP lies in its size and complexity. XMPP messages are XML-encoded and thus permit the expression of many different structures, but XML parsers are typically large and memory-intensive, rendering them inappropriate for embedded devices. Additionally, XML tends towards large messages, which can result in fragmented messages in embedded networks that lower throughput and delivery rate.

Apache Kafka [5] is a distributed messaging system designed for data pipelining in large, distributed, high-throughput applications. Kafka is not designed for deployment scenarios that require rich descriptions of the array of available data services, so publisher and subscriber interactions are done via hierarchical topics and wildcard matching (very similar to MQTT).

Kafka, unlike MQTT and XMPP, is designed to be fault tolerant: all topics are replicated in a cluster of brokers, and failover is automatic. To achieve the combination of fault tolerance and performance, Kafka clients must be carefully engineered, and are intended to be fuller applications rather than embedded devices. Thus, while Kafka may be suitable for a data anaylsis pipeline in a datacenter, it does not meet our requirements for message delivery and reception at the "edge" of an IoT network.

B. Content-Based Publish-Subscribe

In content-based pub-sub systems, publishers attach richer descriptions to messages, allowing subscribers to specify predicates that act as filters for which messages they receive. While this scheme has richer descriptive power, routing on a per-message basis is computationally expensive (reducing routing efficiency) and can require larger messages from publishers.

SIENA [6], the Scalable Internet Event Notification Architecture, aims to maximize the expressiveness of publishers and subscribers communicating in a wide-area network. SIENA publishers send messages containing a set of `<type, name,`

TABLE I: High-level comparison of features between systems

System Name	Publisher Descriptions	Syndication	Client Complexity	Client Failover	Fault Tolerant
redis [1]	N channels	List of channel names	Simple	None	Replicated cluster
MQTT [2][3]	Hierarchical topics	Wildcard matching	Simple	None	None
XMPP [4]	Unique names	List of publishers	Complex	None	Federated Servers
Kafka [5]	Hierarchical topics	Wildcard matching	Complex	Yes	Replicated broker cluster
SIENA [6]	attribute-value pairs	SQL-like predicate	N/A	N/A	Replicated broker, flexible routing
JMS [7]	attribute-value pairs	SQL-like predicate	Complex	Unknown	Yes
CORBA [8]	properties and names	property matching	Complex	No	No
CQBS Broker	attribute-value pairs	SQL-like predicate	Simple	Yes	Replicated brokers, coordinators

value> tuples, to which clients can subscribe using SQL-like filter expressions. This approach, designed to provide discovery of relevant data in a large number of heterogeneous messages, allows publishers to express richer metadata than would be feasible using a topic-based scheme. It also allows subscribers to more precisely define the data they need. For example, this could easily capture the proposed descriptive elements of the temperature sensor described above and allow a subscriber to filter on any combination of those attributes.

SIENA is fully distributed, using a tree overlay for routing and a special "merging" mechanism for pruning unnecessary delivery of messages to subtrees. The disadvantages of SIENA mirror those of other content-based systems: carrying a full description of a publisher with every message reduces the bytes available for application data in embedded, constrained networks typical of the IoT.

JMS [7], the Java Message Service, is a distributed messaging service for connecting distributed application components. Publishers send messages with headers containing standardized key-value pairs, but also contain lists of user-defined key-value properties. Like SIENA, JMS clients subscribe with SQL-like expressions that express constraints on the set of publisher properties. One difference between JMS and other systems is its default setting of exactly-once delivery, which complicates client logic and raises the network overhead of sending or receiving a message.

Distribution and fault-tolerance in JMS is possible, but requires specific configuration of which topics are distributed and among which servers they are distributed. The focus of JMS is on enterprise-type applications that are in need of a messaging system that can adapt to its needs, but does not need to do so dynamically.

CORBA [8], the Common Object Request Broker Architecture, is a data bus for communication amongst distributed objects that provides property- and name-based discovery and event notification. Distributed object models (including DCOM [12]) share many features with content-based pub-sub systems: similar to JMS, published messages contain key-value pairs in the header and body, and syndication is performed by subscribers specifying filters on those attributes. CORBA itself is limited because it is not designed to be distributed, and has no failover or replication mechanisms.

III. Continuous Query-Based Syndication

Continuous Query-Based Syndication (CQBS) is a hybrid publish-subscribe pattern that provides the expressiveness of content-based systems while retaining the simplicity of topic-based systems. The goal of CQBS is to provide a messaging system that can account for and adapt to the heterogeneity of data sources in the IoT. CQBS allows embedded publishers to describe themselves using rich metadata, and provides subscribers with the ability to discover and subscribe to relevant data sources and maintain a consistent view of the context of those sources.

```
// stream identifier for temperature data
UUID = "dd9ef92e-140a-11e6-b352-1002b58053c7"
// register client
metadata = {
  UUID = UUID,
  Location/Room = "410",
  Location/Building = "Soda",
  Location/City = "Berkeley",
  Point/Type = "Sensor",
  Point/Measure = "Temperature"
  UnitofMeasure = "Fahrenheit",
  UnitofTime = "ms",
  Timezone = "America/Los_Angeles"
}
register_msg := msgpack.encode(metadata)
// transmit to local broker
send_to_broker(register_msg)
while True:
    temp_val := read_sensor()
    msg := msgpack.encode({
             UUID = UUID,
             Value = temp_val
           })
    send_to_broker(msg)
    sleep(10)
```

Fig. 1: Pseudocode for a publisher registering a stream and publishing data to a local broker. The figure is explained in fuller detail below.

Here, we first establish the CQBS primitives—streams and metadata—before delving into how CQBS operates and what roles publishers and subscribers play. We then describe the design and implementation of an individual broker. We defer the discussion of the full distributed system to Section IV.

A. Streams and Metadata

A stream is a virtual representation of a specific sensor or actuator channel (a "capability") that is indexed by a 16-byte universally unique identifier (UUID). Each stream is described by *metadata*, which is a bag of key-value pairs: keys are required to be strings, but values may be any one-dimensional

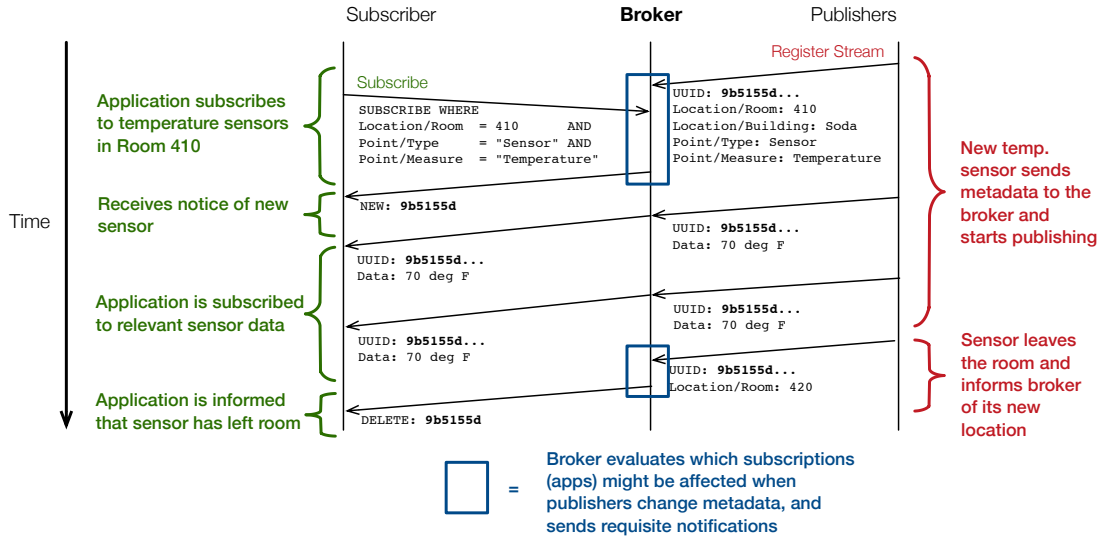

Fig. 2: The network traffic for a continuous query for discovering all temperature sensors in room 410 (omitted for brevity are additional constrains for building, units, etc). As new streams are registered, or their metadata changes to no longer fit the discovery constraints, the client is updated in real-time.

data type[1]. Key-value pairs are most effective when drawn from some well-known ontology (such as Semantic Sensor Web [13]), but our system places no restrictions on their content.

The association of metadata to a stream is done by the UUID; when a publisher creates a new stream, it registers that stream with the broker by sending a message containing the UUID and all of the metadata. The broker (or the coordinator, in the distributed case) stores the mapping from stream UUID to metadata. A publisher changes metadata by sending the "diff" of which keys and values have changed. A given producer (data provider) can have as many streams as it wishes. Each message contains at least the UUID of the originating stream, and can also contain any metadata changes, and of couse the published value itself, which can be any serializable object.

An example of metadata for a temperature sensor and the associated client logic can be found in Figure 1. In the initial registration message, along with the other metadata, the reporting process describes the thermostat as being in room 410 Soda. If this changes, such as if the sensor were on a piece of smart clothing or furniture, the sensor attaches the metadata update `Location/Room = "415"` to any outgoing message, where it is handled by the broker (described in the following section). A discussion of client complexity can be found in Section V.

This is a departure from the approach of content-based pub-sub systems, where although the producer may possess some unique identifier, it transmits any associated "content" (metadata) in every message. This verbose design choice may be appropriate for distributed sytems in which a publisher is a larger application that produces many different types of data, but when the data per-producer is relatively static (temperature sensors will always report temperature data), this flexibility is unneeded. It becomes more efficient to essentially "cache" the metadata of a publisher in a central location where it can be used for syndication.

The simple structure of a stream (essentially a set of special key-value pairs) means a stream can be well represented in nearly any application protocol. We choose MsgPack, a lean, typed binary serialization format that is simple enough to be encoded/decoded on embedded devices with limited code space.

B. Query-Based Syndication

A primary contribution of our distributed broker architecture is its continuous, query-based syndication. Queries are structured, SQL-like statements that define sets of constraints over stream metadata to express ad-hoc relationships between streams. Query-based syndication is the use of these queries to define the forwarding routes from publishers to subscribers. When an application sends a syndication query to the broker, that query is evaluated by the broker into a set of matching streams. The broker constructs forwarding paths for those streams to the subscriber; whenever those streams send a message to the broker, the broker forwards that message to the relevant subscribers.

The resolution of queries to routes is *continuous*; the broker reevaluates syndication queries as stream metadata evolves, and informs clients of changes in the set of the streams to which they are subscribed before adjusting the forwarding paths. These changes happen on any metadata event: stream registration, stream deletion and metadata updates on streams.

Queries are simple strings similar to the "where" clause of a SQL query. These predicates support basic operations

[1]In our implementation, values are restricted to strings: see the Implementation header at the end of this section.

on keys and values: equality, regex matching, existence, and combinations of these using `and`, `or` and `not`.

For example, a hypothetical daylighting application wants to adjust the brightness of lights corresponding to how much natural light is entering a room. It subscribes to the output of all dimmable lights in room 410 as well as all illumination sensors. There are two subscriptions:

```
-- dimmable lights
Room = 410 AND System = "Lighting"
AND has Actuator/Brightness
-- illumination sensors
Room = 410 AND Point/Type = "Sensor"
AND Point/Measure = "Illumination"
```

Figure 2 illustrates a typical exchange of messsages. First, a temperature sensor stream with UUID (starting with) `9b5155d` is registered as being in Room 410. Then, an application enacts a subscription to all temperature sensors in 410 Soda. The broker evaluates this query against its metadata store, and establishes forwarding paths for those streams. The broker then informs the subscriber of the set of streams it is subscribed to. As the sensor publishes, its messages are forwarded to the subscriber. Finally, the sensor is moved to another room — perhaps as part of a piece of smart clothing or furniture — and informs the broker of the change in its metadata. The broker sees that the `Location/Room` tag has changed, so it looks internally for all syndication queries that contain the `Location/Room` key. The broker reevaluates each of these queries, informs the application of the change in the set of its subscribed streams, and then adjusts the forwarding paths.

C. CQBS Broker Design

Continuous queries are an extension of traditional request-response relational queries to capture changes in a query's result set over time. All incoming queries are evaluated against the metadata database or returned from a cache. After the initial results of the discovery query are returned, the broker will continue to deliver updates on the result set to the subscribed client.

The broker maintains several data structures that are updated on any metadata event – such as registering streams, deleting streams or streams changing metadata – to avoid needing to reevaluate all registered subscriptions on every single event. The data structures provide fast lookup of all queries that involve a given metadata key, and store the mapping from a query to the set of the UUIDs for matched streams.

We decouple the metadata and query mechanism from the underlying database by implementing the query language using Go-yacc. This grants the ability to inject functionality at intermediate levels of the parsing process. For each submitted syndication query, the broker extracts the set of metadata keys to optimize the query reevaluation process, which involves two data structures.

The key-query table (the first data structure) maps metadata keys to the set of queries that involve them. Queries are consistently hashed to avoid amplification of the lookup table by duplicate queries with reordered clause terms. Any incoming metadata event will have its keys referenced against this lookup table, generating a set of queries that will need to be reevaluated after the incoming metadata changes are committed. The second data structure, the query-UUID table, stores which streams have been resolved for each query. It is updated whenever a query is evaluated, and simplifies identifying which streams have entered or left a result set for a given query upon a metadata event.

D. Implementation

The CQBS broker is implemented in Go [14], a statically typed, garbage-collected language with language-level support for concurrency in the form of multithreading as well as handling asynchronous operations. The language contains several built-in concurrency primitives: goroutines (lightweight processes), channels (for message passing) and a `select` statement (which helps implement non-blocking operations). For these reasons, the Go language is a natural fit for developing highly concurrent network systems.

Go channels, in both the buffered and unbuffered varieties, make relaying backpressure straightforward. Using channels to convey incoming and outgoing data between pipelined components means that when a component is too loaded to respond to pending tasks, it simply does not dequeue new tasks from the incoming channel. This forces upstream components to block in relaying their tasks to that component, and results in cascading backpressure extending to the client [15]. It is important to note that this backpressure is only applied to publishers when they are sending faster than the broker can sustain, *not* when a particular subscribed client is overloaded.

Having to manage queries against an underlying database during metadata changes introduces a number of blocking operations into the "hot path" of the broker. Goroutines, combined with synchronization primitives such as `sync.WaitGroup`[2] from the Go standard library, are a natural way to dispatch multiple concurrent operations in parallel and wait for their completion. Goroutines are scheduled by the Go runtime, and scale nicely over multiple cores.

The design and implementation of Go does pose several challenges for low-latency systems. First among these is the garbage collector, which is non-generational and mark-and-sweep. Concurrent garbage collection was introduced in Go 1.5, but heap allocations do noticeably contribute to increasing latency. Because Go is garbage collected, a large number of heap allocations can incur high latencies during operation. Many protocol encoder/decoders in Go create many temporary objects, and Go's compile-time escape analysis unnecessarily promotes many stack allocations to heap allocations, as revealed in [16].

Using typed formats such as MsgPack, CapnProto and Protobuf allows parsing code to "plan-ahead" for which types to use and how much space they will use. JSON is particularly bad at this; because it is a character-based format, parsing it requires many intermediate buffers and lookaheads to determine

[2]https://golang.org/pkg/sync/#WaitGroup

the size and type of elements in a received message. Using generated code for encoding/decoding can drastically reduce the number of allocations. We use the excellent `msgp` library[3], which reduced the allocations per message from roughly 20 to 3.

IV. System Design

A. Overview

To meet the goals of client simplicity, availability and scalability, we have developed an architecture which consists of two primary components: distributed brokers and centralized coordinators. See Figure 3 for an overview which will be described in more detail in this and the following sections.

The system contains one logically centralized coordinator; to all other entities in the system, the coordinator can be treated as a single machine. In actuality, this logical coordinator consists of three independent nodes to improve fault tolerance; see Section IV-C for more detail. This coordinator makes all of the decisions in the system, determining when a broker has failed, which brokers should forward which messages where, when changes occur to the set of publishers a subscriber is currently receiving messages from and which broker a client should contact if their broker fails. It then distributes these decisions to the brokers, which take appropriate action.

To do this the coordinator stores the current state of all brokers in the system, as well as information about all of the clients that are known to the system, i.e. what broker they are attached to, what query they are interested in (for subscribers), and what their current metadata is (for publishers). Publisher metadata is stored in a local instance of MongoDB [17] which is used for executing queries.

Brokers are numerous and may reside anywhere; for example, a deployment may consist of a broker located in each building which contains client devices, or brokers may be run on cloud computing nodes. Brokers are responsible for communicating with clients and for forwarding messages along routes as instructed by the coordinator. Any changes to the set of clients connected to the broker, or to the metadata of a publisher connected to the broker, are communicated back to the coordinator for handling so that the coordinator always has an up-to-date view of the entire system state.

B. Normal Operation

In this section we describe the events which take place under normal operation, i.e. in the case that there are no failures within the system.

A new subscriber enters the system. A subscriber contacts its local broker, Broker A, whose address can be hardcoded into the client or discovered through some network discovery protocol, e.g. Bonjour, UPnP, or mDNS. The subscriber submits a message to Broker A containing the query which defines which publishers' output it is subscribed to. Broker A forwards the message along to the coordinator, which evaluates the query against the set of publisher metadata currently known to the system and replies to Broker A with the (possibly empty) set of relevant publishers. If any relevant publishers are found, the coordinator will contact the broker at which they are located and instruct that broker to forward the publisher's messages to Broker A, which will in turn forward the messages to the subscriber. We can see this, for example, in Figure 3: `SmartThermostat.c` is publishing to a broker which has a forwarding link established to another broker to which `TemperatureAlarm.py` is connected, establishing a publication route between `SmartThermostat.c` and `TemperatureAlarm.py`.

A new publisher enters the system. A publisher contacts its local broker, Broker A, in the same manner as a new subscriber. The publisher submits its initial set of key-value metadata pairs to Broker A, which forwards them to the coordinator. The coordinator evaluates this new metadata against the set of currently active queries, notifies any subscribers whose queries apply to the new publisher, and constructs new forwarding routes from Broker A as in the previous case.

A publisher submits new metadata. This process is essentially the same as adding a new publisher, except that in addition to possibly creating new forwarding links, some may need to be removed if the metadata changed in such a way that a publisher is no longer relevant to a subscriber's query. Again, the coordinator will instruct Broker A about which brokers to create (and destroy) forwarding routes to.

Clients leave the system. When subscribers and publishers leave the system, a similar process is followed; the coordinator is notified, and forwarding routes are created or destroyed as necessary.

A publisher publishes a message. When a message does not contain any metadata changes, the coordinator is not involved in any part of the process. The broker to which the publisher is connected simply broadcasts the message over all forwarding routes relevant to that publisher, and the recipient brokers will forward the message to the subscriber.

C. Coordinator Fault-Tolerance

To all other components of the system, including clients, the coordinator appears to be a single node which is resilient to failures; however, to construct a highly available system the coordinator must be able to handle at least one node failure. To ensure that the coordinator will be resilient in the face of machine failure, we replicate its state across three independent nodes. Each coordinator node runs an Etcd [18] node, a reliable key-value store which internally uses the Raft distributed consensus protocol [19] to provide strong consistency semantics among its members. At any given time, one coordinator is designated as the leader; this is the only coordinator node which will accept messages from or send messages to brokers. A single "leader" key is stored in Etcd; whichever coordinator was able to create this key via an atomic create-if-not-exists operation is considered the leader. The key is marked with a time-to-live of a few seconds which is continually refreshed by the leader; if the leader fails to refresh

[3]https://github.com/tinylib/msgp

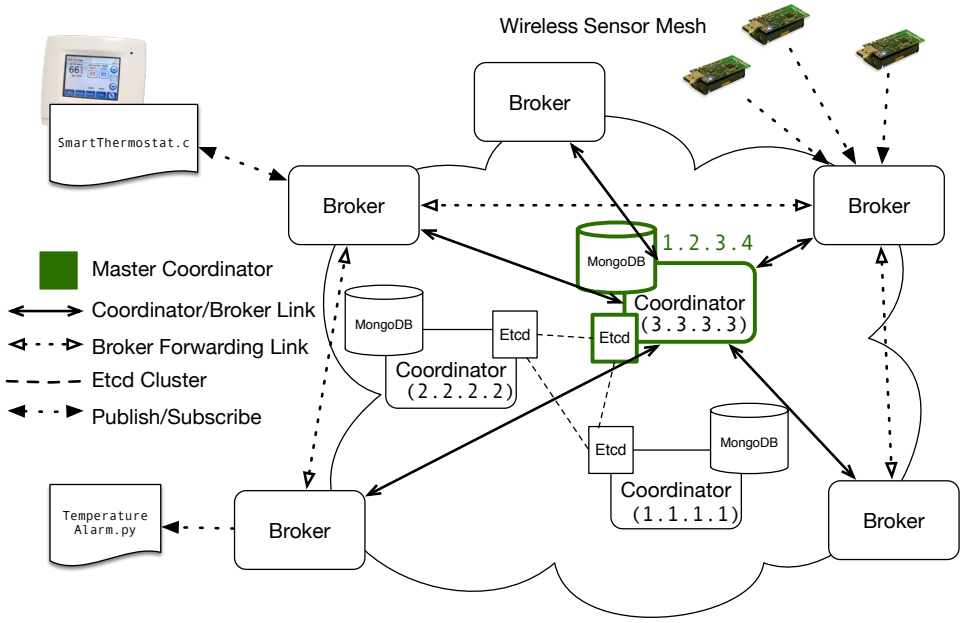

Fig. 3: Overview of the architecture of our brokerage system. Numerous clients communicate to a set of decentralized brokers which create a forwarding network between themselves as instructed by the centralized coordinator nodes.

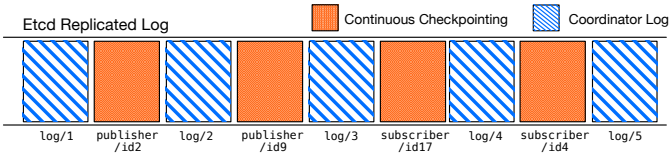

Fig. 4: Although they live in separate key spaces, the log and the continuous checkpointing are serialized via the Etcd log so that they can be leveraged for a consistent rebuild.

its ownership of the key, the key will disappear, allowing another replica to become the new leader.

To make this cluster of machines with potential leadership changes appear to brokers and clients as a single machine, we maintain one external IP address at which the current leader can always be contacted ("1.2.3.4" in Figure 3) in addition to the internal IP addresses which coordinators use to communicate with each other ("1.1.1.1", "2.2.2.2", "3.3.3.3"). Various mechanisms can be used to ensure that this IP address always points to the correct coordinator; we run coordinator nodes on Amazon Web Services (AWS) [4], which provides an "Elastic IP" feature which allows for machine instances to request that an IP be reassigned to them. Unfortunately the Elastic IP feature of AWS has delay of approximately 10–15 seconds (measured during our own experiments) before new requests are successfully routed to the correct host after the mapping has changed; this is a limitation of AWS and not fundamental to our protocol.

Every time a message is received from a broker, the leader logs the message to Etcd to make it durable before processing. The replicas watch this log, continually applying log entries to their state as if they received the message from a broker. This ensures that replicas are always very close to update-to-date with the state of the leader, lagging behind by only the latency of a write-read pair through Etcd (on the order of tens of milliseconds), making IP reassignment the only reason that coordinator failover is not extremely fast. When the leader is finished processing a message, it sends an acknowledgment back to the originating broker. If the broker does not receive an acknowledgment, it will resend the message. So, if a leader writes a message to the log and crashes before finalizing its processing, the broker will eventually resend the message to the new leader, which ensures that the necessary operations are eventually completed. Since messages are idempotent (e.g. attempting to create the same forwarding route twice has no effect the second time), it is acceptable for the new leader to redo some actions that the old leader carried out.

In addition to storing a log of inbound messages, the leader also stores the current state of each client and broker as a key-value pair within Etcd. This is essentially a form of continuous checkpointing which enables a newly instantiated coordinator replica to quickly catch up to the state of the leader. Rather than reading and executing the log from the beginning of time, a new replica can read the current state of the system via the client and broker keys up to some fixed point in time, then resume reading the log from that same point in time. To maintain consistency between the continuous checkpoint and the log, we make use of the fact that Etcd internally maintains a sequential log of events (a consequence of using Raft). Each modification to the Etcd store is marked with a revision number which indicates the order in which the modifications occurred. By choosing some revision number and reading all client/broker state up to and including that

[4]https://aws.amazon.com/

revision, then reading all log entries after that revision, it is ensured that the replica switches from reading the checkpoint to reading the log in a consistent manner; see Figure 4. Note that this works even if the client or broker key was overwritten (i.e. if a change to the state occurred which was then written to Etcd) because Etcd stores versioned copies of key-value pairs.

To clear old checkpoint versions and unnecessary log entries, the leader periodically performs garbage collection. Replicas periodically write the sequence number of the last log entry they have processed to a specific key in Etcd. The leader periodically checks these values and instructs Etcd to delete entries which both replicas have already read, as well as instructing Etcd to perform a "compaction" up to this point, which removes old versions of keys which are no longer needed.

D. Broker Fault-Tolerance

In addition to being resilient to coordinator failures, we require that our system continue functioning and all clients continue to be able to participate in the face of a broker failure. To this end, and with low client complexity in mind as a goal, we have designed a very simple fail-over protocol which allows the client to continue to operate on another broker during the period in which its local broker is not available.

In addition to being aware of its local broker as described in Section IV-B, each client must know the external address of the coordinator; this can be accomplished using the same discovery method used to find the local broker. First, a client attempts to contact its local broker, Broker A. If the client is unable to do so, it sends a request to the coordinator, which will supply it with the address of some other broker B which can service its needs until Broker A is available again. The client connects to Broker B and continues as usual. When Broker A becomes available, the coordinator instructs Broker B to sever its connection to the client, which will then attempt to contact Broker A as usual. This can be summarized by the following pseudocode:

```
while client_active:
  success := connect_to(local_broker_address)
  if success:
    // process ...
  else:
    broker_addr := connect_to_get_message(↙
     ↪ coordinator_address)
    success := connect_to(broker_addr)
    if not success:
      continue
    // process ...
```

E. Design Discussion

This design allows us to meet all of our goals. We have high availability via fault tolerance for both broker and coordinator failures. The necessary code for clients is very simple; publishers need to know how to send publication messages, subscribers need to know how to send query messages and receive publication messages and subscription difference messages (e.g. publisher A just became relevant to your query), and both need to know how to ask the coordinator for a fail-over broker. We also have scalability in terms of number of messages that can be sent through the system by removing any coordination from the normal message forwarding path, which can be scaled arbitrarily with the number of brokers.

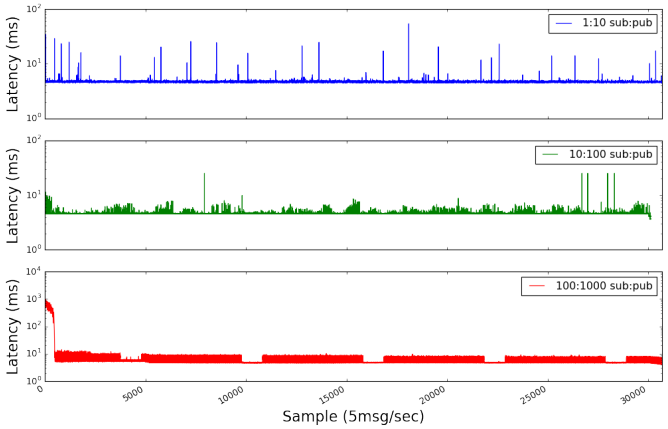

Fig. 5: Microbenchmark: standalone CQBS broker forwarding latency with increasing concurrency.

One aspect on which this design is lacking is that the coordinator is a bottleneck for changes to the state of the system (i.e. clients entering or leaving and publishers changing metadata). We assume that in comparison to the rate of messages being sent in the system, changes to the set of connected clients and to the metadata associated with publishers is relatively slow, so we consider this design to be acceptable. Part of the reason for choosing this design was its relative simplicity; we have considered two alternate designs which were considered and which we hope to evaluate as future work (see Section VI-B)..

V. Evaluation

Here we present the evaluation of a distributed CQBS broker. All brokers and coordinators machines were chosen to emulate commodity hardware; for CQBS to be an effective solution, it must not rely on intractably large resources. Hence, we chose to use the `t2.medium` AWS instance type with 2 vCPUs and 4 GB RAM, running Ubuntu 14.04. As we will demonstrate below, the CQBS system is amenable to commodity systems because its performance is limited by the serialization of the etcd log, and not by memory, CPU, disk, or network bandwidth.

A. Single-Broker Performance

First, we examine the latencies of the CQBS broker's forwarding mechanism in isolation—without the communication overhead imposed by the fully replicated system. We run a single broker on a `t2.medium` instance, backed by MongoDB. Using a ratio of 10 publishers to one subscriber, we run three benchmarks with 1, 10 and 100 subscribers (with 10, 100 and 1,000 publishers accordingly). Each publisher sends 5 messages per second; after the initial registration message (marked by the high latencies at the beginning

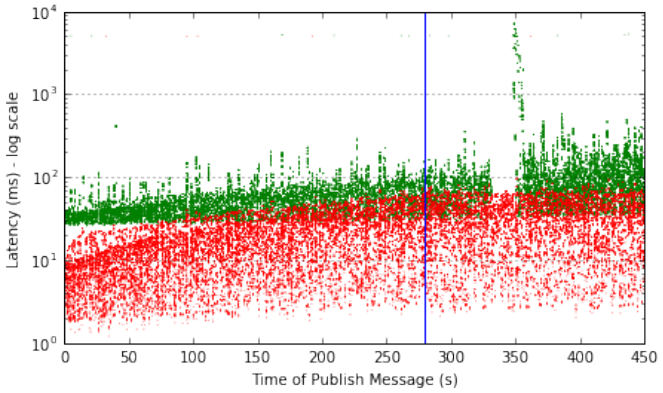

Fig. 6: Microbenchmark: coordinator latency in the face of failures. Red dots indicate running a single-node coordinator with no replication or fault tolerance; green dots indicate running a full three-node coordinator with fault tolerance via Etcd. At the blue line at approximately 280 seconds, the system switches from adding new publishers to existing publishers changing their metadata. At the gap seen in the replicated results at approximately 325 seconds, the leader node of the coordinator cluster was shut down.

of each benchmark), each publisher sends only its stream UUID and an increasing counter as its value. Each group of publishers/subscribers use entirely isolated sets of keys, so that they do not explicitly interfere with each other in the broker. These three microbenchmarks are shown in Figure 5, and demonstrate that the latency is fairly consistent as the amount of concurrency scales.

The spikes in latency seen in the $N = 1, 10$ graphs are due to the pauses enacted by Go's garbage collection. Fortunately, these do not affect the vast majority of requests. For the $N = 1, 10$ cases, the mean latencies, 95$^{\text{th}}$ percentile latencies and standard deviation are $4.67ms/4.85ms/0.55ms$ and $4.62ms/4.77ms/0.37ms$ respectively. For the $N = 100$ case, the garbage collection becomes more visible in the variability of response times, with mean, 95$^{\text{th}}$ percentile and standard deviation latencies of $13.58ms/8.87ms/67.13ms$. The troughs in the $N = 100$ case are an odd phenomenon, most likely caused by garbage collection in the single Go process used to generate the 100 subscribers and 1000 publishers, generating approximately 5000 messages per second.

The microbenchmark demonstrates that a single broker can quite easily handle a constant load of publishers and subscribers with low latency, even on commodity hardware.

B. Coordinator Performance

Next we examine the latency of the central coordinator's query evaluation and decision making processes. We run the coordinator in two different modes; one with a single-machine coordinator which has no fault tolerance and does not use Etcd, and one with a full three-node coordinator cluster which is fully replicated via Etcd. In both cases the coordinator nodes are run on `t2.medium` instances. We start 10 "dummy" brokers which act as if they have publishers and subscribers attached to them. These brokers load the system with a total of 500 subscribers, each of which subscribes to 10% of the total publishers in the system at any given time. At the start of the experiment there are no publishers in the system. New publishers are added to the system at intervals uniformly randomly spread between 50 and 500ms, until there are a total of 1000 publishers in the system. At this point, no new publishers enter the system, but the existing publishers start to change their metadata at the same frequency, causing the set of subscribers to which they are relevant to change over time. We measure the latency of handling the new publisher joining the system, which is defined as the time elapsed between when the publication message is first sent to the coordinator until the time at which a subscriber receives a notification about the new publisher's presence.

In the case of the fully replicated coordinator cluster, we also force the leader of the cluster to fail, causing one of the replicas to become the new leader, and causing all brokers to have to reconnect to this new leader.

Latency results are shown in Figure 6; each dot represents the latency from the viewpoint of a subscriber which received a notification about a new publisher. For the replicated and unreplicated cases respectively, the mean and 95$^{\text{th}}$ latencies are $95.5ms/191.2ms$ and $24.7ms/57.3ms$. We see that in both the replicated and unreplicated cases, latency increases as more and more publishers are added to the system (before 280 seconds), and levels off once the publishers are only changing their metadata (after 280 seconds). We also see that the unreplicated coordinator is over an order of magnitude faster in the best cases, and the highest latencies experienced by the unreplicated coordinator are similar to the lowest latencies experienced by the replicated coordinator. No request to the replicated coordinator takes less than 30–40ms, which is approximately the latency of an Etcd store operation in our experimental setup. This is indicative of the fact that the serialization process through Etcd is a significant bottleneck in our system that severely limits throughput and degrades latency, and indicates that moving Etcd storage farther from the hot path of coordinator decisions could be very beneficial; this is discussed further in Section VI-B.

At approximately 325 seconds we force one of the leaders to fail. This can be seen on the plot as a gap in latencies, since no messages could be sent during this time. The approximately 15 seconds for brokers to reconnect is primarily due to the latency of switching over the Elastic IP address via Amazon AWS API calls, as discussed in Section IV-C. The large spike in a small number of latencies immediately following is due to the fact that some brokers were able to reconnect more quickly, so they began sending messages destined for brokers which had not yet reconnected. However, once all brokers were able to reconnect, we see that the system is able to settle back into a stable state with reasonable levels of latency.

C. Full-System Performance

To evaluate the overhead of the continuous syndication mechanism, we construct a three broker, three coordinator cluster, with every component running on a different ma-

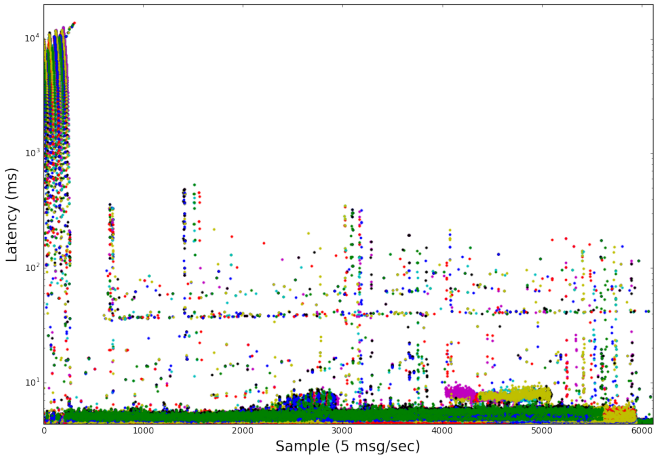

Fig. 7: Benchmark: forwarding latency with 30 subscribers and 360 publishers in a 3 broker, 3 coordinator system. Each publisher changes metadata every 30–60 seconds, causing the broker to reevaluate roughly a third of all subscriptions every few seconds.

chine. We then introduce 10 subscribers at every broker; each subscribes to 4 local publishers (same broker) and 4 remote publishers at each of the other two brokers for a total of 12 subscriptions. In total, there are 120 publishers per broker, with a total of 30 subscribers and 360 publishers in the benchmark. Each publisher changes its metadata every 30–60 seconds, causing the publishers to cycle among the subscribers. The results are illustrated in Figure 7, which illustrates the limitations of serializing the processing of all metadata changes and coordinator updates. While the 95$^{\text{th}}$ percentile latency is only $7.59ms$, the mean latency over the experiment was $129.60ms$ with a standard deviation of $891ms$. This variability is caused when several metadata updates arrive at the coordinator in quick succession: the coordinator buffers these changes until it can process them sequentially, making sure that the forwarding tables are correct and consistent. Because no related messages can be forwarded until the forwarding changes are known, a queue of metadata changes causes cascading delays until the system can catch up. This phenomenon is most prominent around samples 800, 1500, 3000 and 4000. The "waterfall" at the beginning of the stream is caused by all 30 subscribers and 360 publishers querying and registering at the same time, placing a large load on MongoDB and the etcd log.

When considered in conjunction with the microbenchmark above, it is clear that the serialization of the metadata changes and coordinator updates is the limiting factor in system performance. However, in the authors' experience with real world sensor systems using similar systems, these high rates of change are rarely seen: it is important to note that these loads are intentionally unrealistic to find the bottlenecks in the system.

D. Client Complexity

One goal of this system was to maintain a low level of client complexity. To evaluate this, we have written clients in the Go and Python languages. Publishers are capable of publishing values and modifying their metadata, and subscribers are capable of submitting queries and attacking handler functions to respond to inbound published messages and notifications about changes to the set of publishers which they are currently subscribed to. Both types of clients are capable of contacting the coordinator to seamlessly handle the failure of their local broker.

TABLE II: Lines of non-comment, non-whitespace code used to implement programmable clients in Python and Go. Base Code is the basic code necessary to communicate with the system, Failover is the code necessary to communicate with the coordinator to handle broker failures, and Subscriber/Publisher are the code necessary to implement subscriber- and publisher-specific functionality on top of the shared code.

Language	Base Code	Failover	Subscriber	Publisher	Total
Go	175	111	65	69	420
Python	105	60	28	33	226

We present figures for the number of lines of code necessary to create clients in both Go and Python in Table II. The Python client was easily developed in under one day of effort, indicative of the simple nature of the communication protocol and the ease with which it could be implemented on any number of platforms and devices. In this regard we consider ourselves highly successful, providing very high availability while managing to require extremely simple client logic.

VI. Future Work

A. Comparative Evaluation

While researching previous systems, we found many that purport to solve similar problems or had similar approaches (Section II); however, many of these systems are difficult to evaluate for at least one of the following reasons: they are an aging research system that no longer has an actively maintained codebase, the distributed nature was never fully explained or implemented, or most commonly, emulating the desired behavior to directly compare to our CQBS system involved an inordinate amount of implementation. For these reasons, we decided to focus our evaluation on the behavior of our own system, and defer an explorative comparison to other systems for future work.

B. Alternate Designs

Currently, a full Raft transaction is performed through Etcd on every change to the system state. While this provided a relatively simple design with strong consistency guarantees, it incurs a rather high latency that does not parallelize due to the serial nature of Raft transactions. One alternate method to explore would be to use Raft only for leader elections, and have the leader stream events directly to the other coordinators rather than submitting them to the Etcd log. It may be possible to achieve higher throughput using, for example, a 2-phase commit protocol.

Currently, the full state of the system is stored only at the coordinator, which can become a scalability bottleneck. We have considered one alternate design which is essentially the opposite of this, with the coordinator storing no system

state beyond the set of connected brokers. On inbound queries and metadata changes, a broker would broadcast to all other brokers, allowing them to evaluate the changes and set up new forwarding routes as necessary. This is unfortunately expensive as it requires a broadcast, but it does away with the necessity for replication via Etcd since brokers can recreate the state of their system via information they receive when clients reconnect to them, which may be a desirable tradeoff.

Another option we considered that provides a tradeoff between our current design and the one described above would be to have each broker store only its own state, and have the coordinator store only some sort of heuristic data. A broker would forward messages to the coordinator, which would not contain the full system state, but have enough information to narrow the possibly affected brokers to a smaller subset as opposed to having to broadcast to the entire system. This could, perhaps, mean storing ranges of metadata values that publishers at brokers contain. This pushes off the query processing effort onto the brokers and avoids full system broadcasts, making it scalable, but unfortunately still requires the coordinator to have a consistent view of the system to avoid the situation where the coordinator doesn't forward a message to a broker which it should have. However, as this view of the system is only required for performance rather than correctness (since all messages could still be broadcast), it may prove to be a desirable point in the design space.

VII. Conclusion

This paper presents the design and implementation of a distributed broker that implements continuous query-based syndication over richly-defined streams. The CQBS broker, which is implemented in Go, offers more expressive power while remaining tractable to implement on embedded, constrained clients typical of the Internet of Things. CQBS strikes a middle ground between the fast but restricted descriptive power of topic-based pub-sub systems and the rich descriptive power but heavyweight nature of content-based pub-sub. On top of this, we demonstrate how the system can be made highly available using a logically-centralized replicated coordinator to maintain consistency between distributed brokers.

References

[1] redislabs, Salvatore Sanfilippo, "redis pubsub," http://redis.io/topics/pubsub, 2016.

[2] D. Locke, "Mq telemetry transport (mqtt) v3. 1 protocol specification," *IBM developerWorks Technical Library], available at http://www. ibm. com/developerworks/webservices/library/ws-mqtt/index. html*, 2010.

[3] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-SA publish/-subscribe protocol for Wireless Sensor Networks," in *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*. IEEE, 2008, pp. 791–798.

[4] P. Saint-Andre, "Extensible messaging and presence protocol (xmpp): Core," 2011.

[5] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing." NetDB, 2011.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Achieving scalability and expressiveness in an internet-scale event notification service," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. ACM, 2000, pp. 219–227.

[7] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, "Java message service," *Sun Microsystems Inc., Santa Clara, CA*, 2002.

[8] S. Vinoski, "Corba: integrating diverse applications within distributed heterogeneous environments," *Communications Magazine, IEEE*, vol. 35, no. 2, pp. 46–55, 1997.

[9] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[10] A. Rowe, M. E. Berges, G. Bhatia, E. Goldman, R. Rajkumar, J. H. Garrett Jr, J. M. Moura, and L. Soibelman, "Sensor andrew: Large-scale campus-wide sensing and actuation," *IBM Journal of Research and Development*, vol. 55, no. 1.2, pp. 6–1, 2011.

[11] P. Millard, P. Saint-Andre, and R. Meijer, "Xep-0060: Publish-subscribe," *XMPP Standards Foundation*, vol. 1, p. 13, 2010.

[12] M. Horstmann and M. Kirtland, "Dcom architecture," *Microsoft Corporation, July*, 1997.

[13] A. Sheth, C. Henson, and S. S. Sahoo, "Semantic sensor web," *Internet Computing, IEEE*, vol. 12, no. 4, pp. 78–83, 2008.

[14] "Go Programming Language," http://golang.org/.

[15] M. Welsh, D. Culler, and E. Brewer, "Seda: an architecture for well-conditioned, scalable internet services," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 230–243, 2001.

[16] D. Vyukov, "Go Escape Analysis Flaws," https://docs.google.com/document/d/1CxgUBPlx9iJzkz9JWkb6tIpTe5q32QDmz8l0BouG0Cw/preview, February 2015.

[17] MongoDB Inc., "MongoDB," https://www.mongodb.com, 2016.

[18] CoreOS, "Etcd," https://coreos.com/etcd/, 2016.

[19] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 305–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643666