A Highly Resilient and Scalable Broker architecture for IoT applications

Souranil Sen
Stony Brook University
sosen@cs.stonybrook.edu

Aruna Balasubramanian
Stony Brook University
arunab@cs.stonybrook.edu

***Abstract*—A key bottleneck in using IoT devices for applications such as smart cities and smart homes is in scaling the architecture to hundreds of IoT devices. The applications typically use a publish/subscribe architecture. A *broker* collects data from the IoT devices (or publishers) and relays the data to the relevant subscribers such as monitoring and alerting systems. Unfortunately, the broker does not scale as the number of IoT devices increases, and is a *single point of failure*. In this work, we present *Nucleus*, a container architecture that (a) supports scaling to a large number of IoT devices, (b) provides high resiliency to failures, and (c) incurs low overhead in terms of data transfer between the IoT devices and the broker.**

The key intuition in *Nucleus* is to design a stateless broker that decouples the networking functionality of the broker from the state information it needs to maintain. The stateless broker relays information from the publishers to the subscribers, and a separate cache manager maintains the state information required. This architecture allows the brokers to be created rapidly upon failure since they are simple. The cache manager is more complex but can be developed independently of the broker. Finally, we implement the networking substrate between the IoT devices and the broker using MQTT, a protocol that is specifically designed for low bandwidth, lightweight communication.We implement *Nucleus* and evaluate it in terms of scalability, resiliency, and networking overhead.

***Index Terms*—Internet-of-Things, resiliency, scalability, mqtt, broker architecture**

I. Introduction

Internet-of-Things (IoT) is becoming the hotbed for innovation in the fields of Smart homes [37], Smart Cities [3], [29] and many other domains. For example, a large IoT deployment using sensors for pollution and CO2 detection is used for air quality detection [1]. Similarly, deploying IoT sensors on vehicles can enable applications such as smart parking [3] and real-time traffic monitoring [3]. We foresee the number of these IoT deployments to grow exponentially in the future enabling several critical services.

A key component of these IoT deployments is a publish/subscribe *broker*. A publisher is the IoT device that periodically publishes sensor data. Applications such as smart parking and air quality detection subscribe to the data published by the IoT device and are called subscribers. A broker is an entity that connects the hundreds of IoT publishers to their subscribers by sending data between the two.

The problem is that the broker is often a single point of failure. Since the brokers connect several hundreds of publishers and subscribers, they are often over-subscribed leading to packet losses and other failures. When the broker fails, the messages from the publishers are lost and the subscribers need to resubscribe to the broker when it comes back alive. In the event of an increased load from the IoT devices, the broker eventually starts dropping packets. Existing open-source publish/subscribe brokers such as Mosquitto [24] suffer from such scalability problems.

In this work, we design an open-source publish/subscribe broker that we call *Nucleus*. *Nucleus* achieves: (1) high scalability, (2) resilience to failure, and (3) low bandwidth communication between the broker and the publisher/subscriber. The key idea in *Nucleus* is to decouple the broker's communication architecture from the state information. In effect, *Nucleus* uses stateless brokers that can send and receive information from the subscriber and publisher respectively. The state information regarding all the connected subscribers and publishers and the information regarding subscriptions is maintained in a separate shared cache. This shared cache has high resiliency and fault tolerance.

The advantage of our architecture is that since the broker design is simple, they can be implemented as lightweight containers. The brokers can also be auto-scaled easily and quickly by existing container management tools, and new brokers can be spawned easily in case of failures. Maintaining the state information is more complex, but fault tolerance for the state information is implemented independently of the brokers themselves. We use a separate in-memory, key-value store to implement the shared state/cache.

Finally, we use MQTT (MQ Telemetry Transport) [25], a lightweight protocol for communication between the broker and the publisher/subscriber. The data transfer in publish/subscribe systems are different from the Internet in that messages are short but numerous. HTTP is ill-suited for this communication because of its high communication overhead per message. We find that MQTT is well-suited for this publish/subscribe environment.

We implement the broker using node.js [27] and use docker [6] containers to deploy the service. We implement the container management service using Kubernetes [20]. The cache is implemented using Redis [31], a publish/subscribe key-value data store that provides high reliability.

We evaluate *Nucleus* with both a physical IoT device (using a Raspberry Pi Zero) and emulated IoT publishers. The brokers are deployed as containers on the Google Cloud Container

Engine [10]. We show that the broker is able to modulate the CPU utilization even with an increase in the number of IoT publishers from 1 to 300. In comparison, a broker that maintains its own state experiences twice the loss compared to *Nucleus*. We show that *Nucleus* can successfully autoscale as the number of IoT devices increases. Finally, we show that the throughput in terms of the number of messages per second over MQTT is 4.5 times higher compared to HTTP.

II. Use cases and Challenges

In this section, we describe two IoT use cases and the existing challenges in these use cases.

A. Smart Cities and Smart homes

Despite no formal and accepted definition of "Smart City", it's roughly defined as a city that integrates information and communications technology (ICT) and Internet of Things technology in a secure and reliable fashion to manage the city's assets [29]. According to Smart City ICT requirements [29] they are of two major types - operational services and citizen-centric services.

An example of an operational service is air quality sensors such as *MiCS5524* [1]. These sensors detect CO, alcohol, and Volatile Organic Compounds. Applications such as smart waste bins subscribe to this sensor data to optimize waste collection routes in the city [29], [38]. Smart irrigation systems were deployed in 68% of the public parks in Barcelona to measure rain & humidity which resulted in a 25% increase in water conservation and a saving of $555,000/year [3].

In terms of citizen services, IoT devices equipped with GPS sensors have been used to enable applications such as smart parking [3], [29] and smart transport [3], [29]. A vehicular-scale IoT deployment provides fine-grained location information to enable these real-time services.

Smart homes is another area where a swarm of IoT devices has been used to support a wide-range of intelligent applications [4], [12], [26], [30]. For instance, smart lights can detect room occupancy to turn themselves off [30], [33] and smart air quality controls [1], [29] and smart temperature controls [26] modify the room temperature depending on weather and occupancy.

B. Broker architecture and Existing challenges

The IoT use cases described above is supported by a *publish/subscribe broker*. The IoT devices or the publishers generate small amounts of information, typically from their on-board sensors. Different applications, or the subscribers, subscribe to the data generated from the publishers and the broker facilitates the communication between the two entities. A publish/subscribe network provides levels of segmentation using *topics and subtopics*. For example, the GPS sensors in the public vehicles example may publish data on the topic *"transportLocation"*; the subscribers use this keyword to subscribe to the data.

The *broker* is the key component in the architecture since it is responsible for hundreds of IoT publishers interacting with possibly tens of subscribers. Unfortunately, existing brokers are *closed systems* [2], [14], [16] or use open source broker libraries that do not provide desired performance [24]. For instance, Amazon's AWS-IoT [2] or HiveMQ [14] are closed source broker services; because of their closed nature, it is hard to reason about their performance or innovate on new designs. Open source libraries such as Mosquitto [24] suffer from problems such as single-point-of-failure and lack of load balancing, which results in a complete breakdown of the system if the broker fails.

III. Nucleus Architecture

In *Nucleus*, our goal is to build an open source IoT publish/subscribe broker that has three design goals:

- Resiliency: We define resiliency in terms of *(i)* availability: minimizing the time taken for a broker to spin back up after a failure, and (2) integrity: persistent state and cache information even after a system failure.
- Auto-scaling: We design *Nucleus* to scale to a large number of publishers and subscribers, but also scale back in size when the load on the system reduces.
- Low bandwidth communication: We leverage a low-bandwidth, low-energy communication protocol for the broker interactions, well-suited for connecting several hundreds of devices.

The key idea in *Nucleus* is to decouple the broker's communication mechanism from the system that makes decisions on the communication policies. In effect, the *Nucleus* brokers are *stateless* and can be implemented using lightweight containers. The state information and the smarts that make the communication decisions are maintained separately in a highly available fast key-value data store.

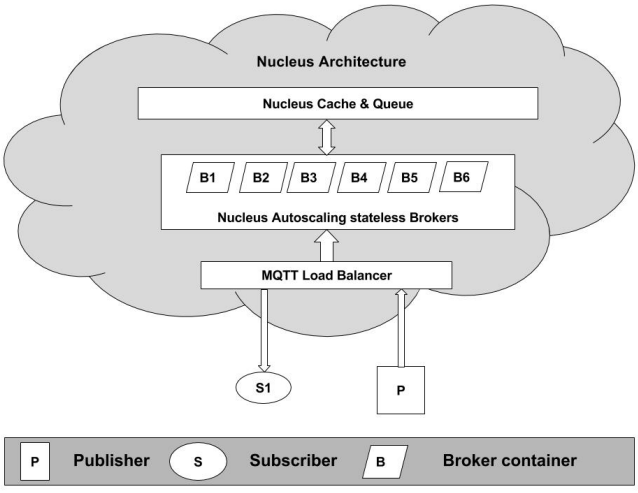

Fig. 1. *Nucleus* Architecture.

Figure 1 shows the *Nucleus* architecture. The *Nucleus* broker has three parts: (1) the *Nucleus* autoscaling broker, (2) the *Nucleus* shared queue and cache, and (3) the *Nucleus*

MQTT endpoint. *Nucleus* also uses a standard load balancer for balancing load across the brokers.

***Nucleus* stateless broker:** The stateless broker is implemented as a suite of lightweight containers. A container management service [20] auto scales the number of brokers. *Nucleus* uses CPU utilization as a trigger to increase and decrease the number of brokers. The user of the system specifies both the minimum number of brokers that need to be maintained and the CPU utilization threshold. As the utilization increases, the container management service spawns new brokers and scales back when the CPU utilization decreases.

The key to the autoscaling broker is its lightweight nature; since the broker only implements the communication mechanism and not the policies, they can be easily created and destroyed as needed (as shown in §V).

***Nucleus* Shared State and Queue:** A shared state maintains the information required by *Nucleus* for its publish/subscribe functionality. This shared state includes the ID and connection details of each subscriber and publisher connected to the broker, as well the topics and subtopics that are published and subscribed to. A separate queue stores all messages that are received from the IoT devices. *Nucleus* identifies the subscribers of the message based on the state information. This list of subscribers is sent to the broker which simply performs the message delivery. The load balancer balancers the load to and from the stateless brokers.

Nucleus uses Redis [31], an open source advanced key-value cache and store, which is often referred to as a data structure server to implement the shared state & as a queue. The values can contain strings, hashes, lists, sets, sorted sets, bitmaps, and hyperloglogs. We configure Redis to run in sentinel mode [32] for fault tolerance and automatic recovery on failure. Since the fault tolerance is only applied to the shared cache and not the entire broker architecture, we can implement resiliency at low cost.

MQTT communication endpoint: The communication between the broker and the publisher/subscriber is different from typical Internet communication because the amount of data that is exchanged is small (typically below 100 bytes) but numerous. A typical publisher may generate sensor data in a range of 10 to 100 times per second. Even though HTTP is the most common data transfer protocol, it is not well suited for sending a large number of small messages. HTTP incurs huge overheads due to its header sizes and requires synchronous handshakes to set up a connection; this makes scaling to a large number of subscribers difficult.

Instead, in *Nucleus*, we use the MQTT protocol [25] for communication between the broker and the publisher and subscriber. Similar to HTTP, MQTT works on top of a persistent TCP connection. However, unlike HTTP the headers are small and the MQTT connection set up does not require a synchronous handshake. MQTT provides flexible support for reliability that is configurable—from no reliability to high reliability implemented using acknowledgments.

IV. Implementation

We implemented the *Nucleus* architecture described in the previous section. The implementation is shown in Figure 2. We describe the implementation details below.

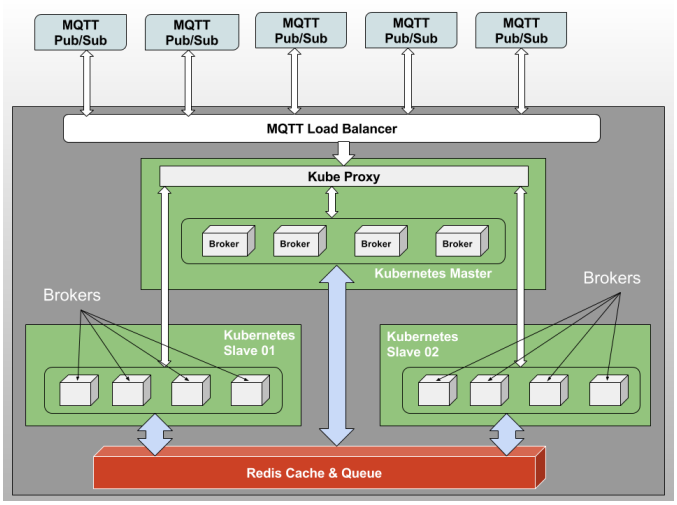

Fig. 2. Implementation.

A. Implementing the stateless broker and container management

The stateless broker is implemented using Node.js [27]. The broker communicates with the subscriber and publisher using the MQTT protocol as described earlier. The broker communicates with the *Nucleus* shared state and queue using a simple RPC.

Each broker is run as a lightweight docker container that can easily be created and torn down [6]. We use Kubernetes [20] to scale the brokers and to perform container management. Kubernetes is an open-source system for automating deployments, scaling, and management of containerized applications.

We configured a private Kubernetes [20] cluster on Amazon EC2 [7] with 1 master node and 6 child nodes. The master node assigns jobs to the child nodes. The Kubernetes cluster is used to set up the minimum number of brokers, the autoscaling configuration, and exposing internal services to the internet so that clients can connect to the broker. Kubernetes spawns new containers in case of failures or when the CPU utilization increases above a threshold. We use Heapster [13], a library to collect container statistics, to monitor the CPU utilization. We also use Sysdig [35] for real-time container monitoring.

B. Implementing shared cache and queue

The shared cache is implemented using Redis [31]. Redis provides an in-memory key-value store that is reliable and fast. The Redis implementation maintains all state information such as connected clients (subscribers and publishers) and topics and subtopics.

The queue is also implemented using Redis [31]. The *Nucleus* queue implementation ensures reliability since even

Component	Configuration
Broker Node	2 vCPUs,7.5 GB memory
Publisher Node (emulated)	4 vCPUs, 15 GB memory
Subscriber Node	Macbook Pro, 2.2 GHz Intel Core i7,16 GB memory, Intel Iris Pro 1536 MB GPU
Publisher (Real)	Raspberry Pi W Zero,1GHz, single-core, 512 MB memory

TABLE I
EVALUATION SET UP

if a broker fails, the data is stored in the queue and can be re-sent. Both the shared cache and queue can be implemented using several alternate technologies including Kafka [17] and MongoDB [23].

C. Implementing load balancer and communication

We use a TCP load balancer as the single endpoint for the publishers and subscribers to connect to. The load balancer chooses the right container to send the messages from and to the publishers and subscribers respectively. We also implement a load balancer in the Kubernetes cluster to balance the load across the slave nodes, using Kube Proxy [19].

Finally, we implement the publishers and subscriber also using Node.js that connect to *Nucleus* broker. We deploy the broker in the Kubernetes [20] cluster, and run the publisher and/or subscriber client in a Raspberry Pi W Zero, as shown in Figure 3.

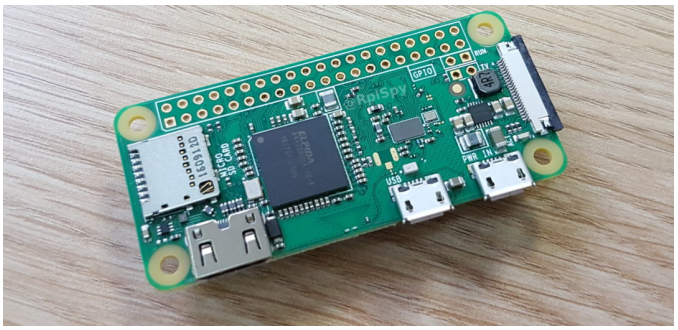

Fig. 3. Raspberry Pi W Zero.

V. EVALUATION

We evaluate the *Nucleus* implementation in terms of our design goals: reliability, scalability, and low bandwidth.

A. Set up

We deployed our *Nucleus* implementation using Google Cloud Engine(GKE) [10], the brokers running in Docker containers in the cluster each with limits of 100 millicores and 300MByte memory. Each of these brokers can handle roughly 12,800 connections. A TCP Load balancer is used at the endpoint in GKE [10], [36].

For experiments, we use both the Rasberry Pi publisher and emulated IoT publishers. We emulate 0–299 publishers using a separate Kubernetes [20] cluster, and add a real Rasberry Pi to the total number of publishers. The configurations used for the broker, the publisher and subscriber are shown in Table I.

Each publisher sends 79 bytes of data per second. The CPU threshold for autoscaling was set to 3%. During the experiments, the *Nucleus* shared cache maintains state information amounting to 15 entries per publisher and subscriber; i.e., for 300 publishers/subscribers, the number of entries in the cache is 4500.

For experiments on scaling, we artificially inject CPU load in the system, to increase the CPU utilization of the broker to 99%.

B. Evaluating scalability

We compare the performance of *Nucleus* with a system that does not support autoscaling. During the experiment, we increase the number of concurrent publishers from 1 to 300. Figure 4 shows how *Nucleus* (orange line) automatically scales the number of brokers to 4 and then to 8 as traffic increases, thus reducing the average CPU utilization to below the desired threshold. For the alternative broker that does not support autoscaling, the CPU utilization increases proportionally to the traffic.

To study the correctness of the scaling, we use Sysdig [35] to monitor the containers to simultaneously observe the CPU utilization and when a new broker is spawned. Our goal is to study the behavior of the autoscaling as the average CPU utilization increases. We set the relative CPU threshold to 3%.The sysdig tool allows us to superimpose the CPU utilization with information about when a broker is spawned. Figure 5 shows the CPU utilization and the time when new brokers are spawned (shown using the yellow vertical lines). The figure shows that a new broker is spawned when the CPU increases beyond the threshold.

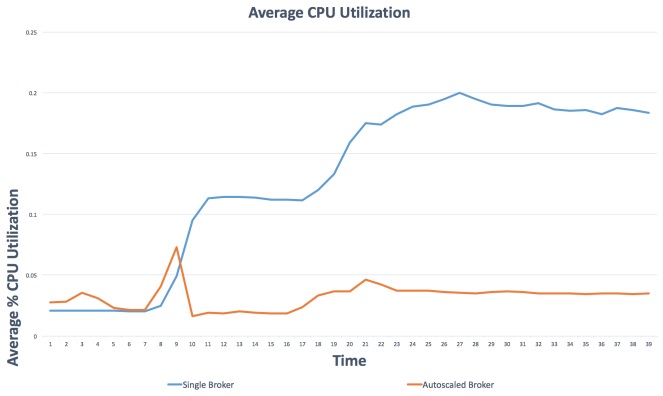

Fig. 4. Comparison of CPU load when using *Nucleus* broker versus a broker that does not support autoscaling.

C. Evaluating resiliency

To evaluate resiliency to failures, we emulate broker failure to measure the time taken for a new broker to be spawned. In our experiments, the maximum time to spin a broker upon failure over 10 experiments was 6 seconds. Figure 6 shows a

Protocol	Throughput(messages/sec)
MQTT	18
HTTP	4

TABLE II
THROUGHPUT OF MQTT VS HTTP

snapshot of the broker being spawned when an existing broker terminates.

Next, we evaluate the packet loss when the load on the system increases. As load increases, the data sent from the publisher is lost, leading to loss of information. However, the shared cache and queue architecture, as well as autoscaling, provides resiliency.

Figure 7 shows the packet loss with and without a shared cache and queue as the number of publishers increase. The broker without any cache has almost double the packet loss compared to *Nucleus* broker. Until 30 publishers, the advantages of the *Nucleus* architecture is not evident (not shown in the figure). But as the number of publishers increases the load on the broker increases, resulting in increased packet loss. However, the combination of *Nucleus*'s autoscaling and caching reduces the packet loss when using *Nucleus*.

D. *Evaluating throughput*

We evaluate the throughput to send data between the broker and the publisher/subscriber. In this experiment, we compare the performance of MQTT and HTTP. To measure the throughput, we configure the publisher to broadcast every *100ms* with a fixed packet size of 79 bytes for both the protocols. We then record the throughput in terms of the number of messages that can be sent per second.

Table II shows that MQTT can send 4.5 times as many messages compared to HTTP. The main reason is that MQTT incurs low overhead per message because of the small headers.

E. Nucleus *Tradeoffs*

The key tradeoff introduced by the *Nucleus* architecture is the additional access delays. By maintaining state information separate from the broker, the state information needs to be accessed through an additional query to the cache. In the default broker architecture, the state information is maintained in-memory, making it faster to access. In effect, *Nucleus* trades off additional delays in accessing state for other desirable properties including auto-scaling and resiliency.

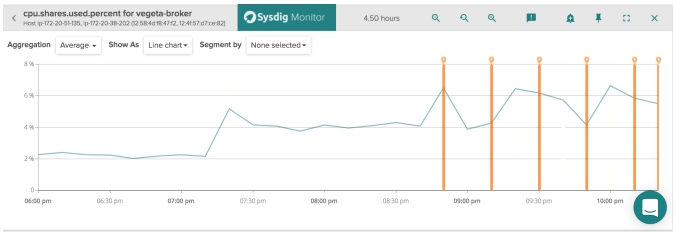

Fig. 5. Experiment showing how autoscaling occurs in response to increase in CPU utilization beyond a pre-defined threshold.

```
ubuntu@ip-172-31-26-60:~/kube-configs$ kubectl get pods
NAME                                READY     STATUS              RESTARTS   AGE
redis-master-815777315-lb30m        1/1       Running             0          9d
sysdig-agent-3gqvf                  1/1       Running             0          1d
sysdig-agent-5g46j                  1/1       Running             0          9d
sysdig-agent-5jc41                  1/1       Running             0          13d
sysdig-agent-7vbz5                  1/1       Running             0          13d
sysdig-agent-dt4c9                  1/1       Running             0          1d
sysdig-agent-lc82r                  1/1       Running             0          13d
sysdig-agent-mh564                  1/1       Running             0          3d
trunks-3621051960-8556f             1/1       Running             0          4h
trunks-3621051960-kd2tn             1/1       Running             0          10d
vegeta-broker-1428352281-g0s45      0/1       ContainerCreating   0          1s
vegeta-broker-1428352281-gddrn      0/1       ContainerCreating   0          1s
vegeta-broker-1428352281-l3s2x      1/1       Terminating         0          2m
vegeta-broker-1428352281-ppp5s      1/1       Terminating         0          3m
vegeta-broker-1428352281-w367g      1/1       Terminating         0          3m
vegeta-broker-1428352281-zbpgz      0/1       ContainerCreating   0          1s
```

Fig. 6. Snapshot of the monitoring tool that shows the time taken to spawn a new broker when an existing broker fails.

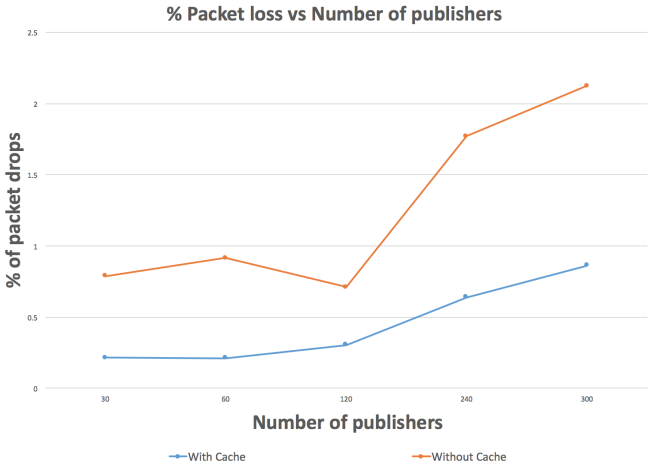

Fig. 7. Comparison of lost messages when using the *Nucleus* and using a broker architecture without the shared cache.

VI. RELATED WORK

Related research on Publish/Subscribe networks [11] have focussed on reducing latencies among brokers in a cluster [9], [34] and communicating with MQTT brokers via REST APIs [5]. The use of MQTT as a Publish/Subscribe protocol has been studied by others [15], [18], [21], [22], but these works do not focus on scalability and fault tolerance of the broker.

There are multiple companies that have closed-sources solutions to implement the cloud broker including IBM BlueMix [16], AWS IoT [2], HiveMQ [14]. The closed-implementation makes it harder to innovate or understand the performance implications of different design choices. There are a number of open source brokers available such as mosquitto [24], emqtt [8], and Eclipse Paho [28]. Mosquitto [24] uses a single broker cauding a single point-of-failure. Both Eclipse Paho and EMQTT are scalable messaging protocol but do not support cache and queue management.

VII. Conclusions

An important problem in IoT deployments is the scalability of the publish/subscribe broker that maintains the communication between the IoT publishers and subscribers. *Nucleus* addresses three main problems that face broker architectures: scalability without user intervention, resiliency in the face of failure, and the need for low bandwidth/low energy communication between the publishers and subscribers. The key idea in *Nucleus* is to decouple the communication service provided by the brokers from the service that maintains information about the publishers and subscribers. By designing a highly available and resilient shared cache and queue completely independently from a simple communications broker, *Nucleus* is able to scale to a large number of publishers and subscribers. Because of its simplicity, the communication brokers can be spawned easily, improving resiliency and providing auto-scaling. Finally, the *Nucleus* broker uses a lightweight communication protocol called MQTT for communicating with the publisher and subscriber. Our implementation and evaluation shows that *Nucleus* indeed achieves its three design goals of salability, resiliency, and low bandwidth communication.

References

[1] Adafruit MiCS5524 CO, Alcohol and VOC Gas Sensor Breakout. https://www.adafruit.com/product/3199.

[2] AWS IoT. https://aws.amazon.com/iot.

[3] Barcelona Smart City. http://datasmart.ash.harvard.edu/news/article/how-smart-city-barcelona-brought-the-internet-of-things-to-life-789.

[4] A. B. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon. Home automation in the wild: Challenges and opportunities. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2115–2124, New York, NY, USA, 2011. ACM.

[5] M. Collina, G. E. Corazza, and A. Vanelli-Coralli. Introducing the qest broker: Scaling the iot by bridging mqtt and rest. In *2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications - (PIMRC)*, pages 36–41, Sept 2012.

[6] Docker. https://www.docker.com/.

[7] Amazon AWS EC2. https://aws.amazon.com/ec2/.

[8] EMtt. http://emqtt.io/.

[9] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[10] Google Cloud Container Engine. https://cloud.google.com/container-engine/.

[11] D. Guinard and V. Trifa. *Building the Web of Things: With Examples in Node.Js and Raspberry Pi*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2016.

[12] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan. Bolt: Data management for connected homes. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 243–256, Seattle, WA, 2014. USENIX Association.

[13] Heapster. https://github.com/kubernetes/heapster.

[14] HiveMQ. http://www.hivemq.com/.

[15] U. Hunkeler, H. L. Truong, and A. Stanford-Clark. A publish/subscribe protocol for wireless sensor networks. In *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, pages 791–798, Jan 2008.

[16] IBM Bluemix. https://www.ibm.com/cloud-computing/bluemix/internet-of-things.

[17] Apache Kafka. https://kafka.apache.org/.

[18] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate. A survey on application layer protocols for the internet of things. *Transaction on IoT and Cloud Computing*, 3(1):11–17, 2015.

[19] Kube Proxy. https://kubernetes.io/docs/admin/kube-proxy/.

[20] Kubernetes. https://kubernetes.io/.

[21] S. Lee, H. Kim, D. k. Hong, and H. Ju. Correlation analysis of mqtt loss and delay according to qos level. In *The International Conference on Information Networking 2013 (ICOIN)*, pages 714–717, Jan 2013.

[22] D. Locke. Mq telemetry transport (mqtt) v3. 1 protocol specification. *IBM developerWorks Technical Library*, 2010.

[23] MongoDB. https://www.mongodb.com/.

[24] Mosquitto. https://mosquitto.org/.

[25] MQTT. http://mqtt.org/.

[26] Nest. https://nest.com.

[27] NodeJS. https://nodejs.org/en/.

[28] PahoMQTT. https://eclipse.org/paho/clients/python/docs/.

[29] R. Petrolo, V. Loscrí, and N. Mitton. Towards a smart city based on cloud of things. In *Proceedings of the 2014 ACM International Workshop on Wireless and Mobile Technologies for Smart Cities*, WiMobCity '14, pages 61–66, New York, NY, USA, 2014. ACM.

[30] PhillipsHue. http://www2.meethue.com/en-us.

[31] Redis. https://kubernetes.io/.

[32] Redis Sentinel. https://redis.io/topics/sentinel.

[33] Samsung SmartThings. https://www.smartthings.com.

[34] Y. Sun, X. Qiao, B. Cheng, and J. Chen. A low-delay, lightweight publish/subscribe architecture for delay-sensitive iot services. In *2013 IEEE 20th International Conference on Web Services*, pages 179–186, June 2013.

[35] Sysdig. https://sysdig.com/.

[36] TCP Cloud Load Balancer. https://cloud.google.com/compute/docs/load-balancing/tcp-ssl/tcp-proxy.

[37] Y. Upadhyay, A. Borole, and D. Dileepan. Mqtt based secured home automation system. In *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pages 1–4, March 2016.

[38] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32, Feb 2014.