Noctua: A Publish-Process-Subscribe System for IoT

Kwame-Lante Wright
University of Southern California
Los Angeles, CA 90007
kwamelaw@usc.edu

Bhaskar Krishnamachari
University of Southern California
Los Angeles, CA 90007
bkrishna@usc.edu

Fan Bai
General Motors
Warren, MI 48092
fan.bai@gm.com

***Abstract*—The publish-subscribe messaging scheme has proven to be an effective real-time communication abstraction for IoT applications; by decoupling sensors from actuators, it helps to ease deployment of such systems. However, many IoT applications consume data from various sources before they take an action, but they are not always interested in the raw data itself but rather a refined, computationally processed version of it. Network bandwidth and device energy are wasted when the computation is performed at end-points that are constrained devices. To address this issue, we advocate for an extension of the traditional publish-subscribe approach, a new messaging paradigm we refer to as *publish-process-subscribe*. We present Noctua, a framework that enables a publish-process-subscribe architecture for IoT applications. Through a real-system implementation in JavaScript based on Node.js, we demonstrate and evaluate how Noctua can help IoT developers by enabling more efficient use of network resources and reduces the strain on edge devices by delivering to them more meaningful data. We illustrate Noctua's capability through application examples including aggregating multiple sensor flows and providing radio signal-strength-based localization as a real-time service. We also incorporate role-based authorization and access mechanisms within Noctua to provide fine-grained support for privacy by facilitating the deployment of application-specific anonymization and filtering of raw data streams in a customized, differentiated manner for different sets of users.**

I. Introduction

As IoT devices continue to be adopted and their applications grow, there has been an increasingly diverse group of developers engaging with them. These developers have access to a variety of tools which remove the need for them to have extensive training or a background in hardware or software [1], [2]. In addition to the low cost of IoT devices, we believe this level of accessibility to people from a variety of backgrounds has contributed to the widespread adoption of such devices. People are now able to interact with and customize the experiences they have with their environments in ways never thought possible.

IoT devices have become much more powerful than the small embedded devices, or motes, typical of wireless sensor networks. Some IoT devices feature multi-core processors, embedded GPUs, tons of RAM, etc. While the hardware has improved, the fundamental limitations of the wireless medium over which these devices communicate has not changed. There is still great value in reducing the communication overhead of an application so that it can operate as efficiently as possible. To that end, tools and frameworks that help developers create efficient applications from the start are valuable.

A paradigm that has been gaining in prominence to s mupport the rapid development and deployment of real-time IoT applications is publish-subscribe implemented in protocols such as MQTT [3]. The benefit of the publish-subscribe approach is that it allows for fast and robust implementation of real-time many-to-many communications. It allows for asynchrony and is forgiving when faced with lossy and dynamic connectivity. However, the basic publish-subscribe paradigm doesn't provide mechanisms to enable efficient in-network computation that could be used to reduce bandwidth utilization and improve privacy.

To meet some of these major challenges associated with IoT systems, we advocate for an extension of the traditional publish-subscribe approach that we refer to as *publish-process-subscribe*, which allows for the en-route processing of data. There are several advantages and applications of the publish-process-subscribe paradigm:

- *Sensor Data Analytics, Fusion and Aggregation*: By allowing for en-route computation, data flowing from one or more publishers can be combined and processed together using data-analytics algorithms such as estimation and prediction and other machine learning algorithms to provide a more meaningful stream of refined, analyzed data for a subscriber.
- *Bandwidth, Latency, Energy Improvement*: Related to the above point, by processing raw data within the network, the total amount of data that is streamed can be reduced, improving bandwidth utilization. By performing data computations at a more powerful server en-route rather than compute-constrained and energy-constrained end points, the latency and energy expenditure associated with data computation could be reduced.
- *Privacy*: A raw data stream from one publisher could be processed through an anonymizing filter. Now, access controls could be set up via the broker so that certain authorized subscribers can have access to the raw data, while others are provided access only to the anonymized version.
- *Computation for Automated Control*: Computation specified over streaming published data could also be used to generate processed streams intended to control particular

actuators. This could be useful when there is limited computation at the client side.
- *Virtual Sensors*: To simplify application design, data from physical sensors may first be transformed into a virtual sensor. This would allow for changing and upgrading physical sensors in a deployment over time while providing the same abstraction to higher layer applications.

In this paper we describe an implementation of the publish-process-subscribe paradigm in the form of Noctua, a publish-subscribe broker that provides for flexible computation so that a client can subscribe to a processed version of published raw data from one or more publishing devices. Noctua is written in JavaScript and powered by Node.js. We demonstrate that this system enables the efficient use of network resources for a wide range of IoT applications. We also show how Noctua facilitates the automated implementation of a role-based access control mechanism to provide security and privacy for real-time IoT streams.

II. Publish-Subscribe and MQTT

The publish-subscribe communication paradigm lends itself well to the producer/consumer abstraction common in many IoT applications. Unlike the more common and intuitive request-response communication scheme, publish-subscribe messaging has the ability to decouple devices in both time and space [4]. Before going into more detail about publish-subscribe messaging, we will first provide an explanation of request-response messaging to better explain the advantages of publish-subscribe for IoT systems.

A. Request-Response Messaging

The request-response communication scheme is what we use when we retrieve a webpage from the Internet. As the client, we send an HTTP message to a web server that is waiting for a request to arrive. The web server is associated with a port at some IP address, either already known or retrieved through a DNS lookup. This address is used to route the request. The server processes the request and sends back the appropriate response, at which point the interaction ends. For additional content, this process is repeated multiple times. If a client wants to check if some content has changed, he or she would need to poll the server by sending more requests. This is a very inefficient method. We'll see in the following section how publish-subscribe messaging provides a better method for accomplishing this.

B. Publish-Subscribe Messaging

In contrast to request-response messaging, publish-subscribe messaging enables clients to specify interest in certain information. This interest is expressed through a subscription. When data relevant to that subscription is published, all subscribers will automatically receive an update containing the data. There are different variations of publish-subscribe but in this paper we will focus on the topic-based version, the type used by the MQTT protocol. Topics are names used to refer to certain data. We'll discuss MQTT further in the following section.

Whereas a request-response system has clients and servers, a publish-subscribe system typically consists of publishers, subscribers, and a broker. The broker is an analogue to a server in this messaging scheme. Brokers are responsible for accepting and keeping track of subscriptions and relaying new data to the interested parties as it becomes available. Publishers are the source of data in this system. They send updates to the broker on the topics they are associated with. Subscribers, on the other hand, consume this information, specifically from the topics they have interest in.

C. MQTT (Message Queue Telemetry Transport)

MQTT is an application-layer protocol for publish-subscribe messaging [3]. It is simple and lightweight making it a popular choice for IoT applications. MQTT uses TCP as its transport layer protocol but there is a variant, MQTT-SN (previously MQTT-S) [5], designed to run over UDP. Figure 1 provides an illustration of how data flows through an MQTT broker.

In MQTT, topics are specified as strings and provide a way for applications to refer to data they're interested in. An example of a topic is "car/temp". Slashes have a special meaning in MQTT. They are referred to as topic level separators and are used to specify a hierarchical structure to the data. This is taken into consideration when a subscriber uses a wildcard in their subscription. So for example, if a subscriber subscribes to "car/#", they will receive any data published to topics beginning with "car/", such as "car/temp" and "car/speed".

MQTT defines about a dozen message types. A few of particular interest are `CONNECT`, `SUBSCRIBE`, and `PUBLISH`. `CONNECT` messages are used when clients first establish a connection to an MQTT broker. If authentication is enforced, clients will be required to provide their login credentials in this message. The `SUBSCRIBE` message is used by clients to tell the broker which topics they are interested in. Finally, `PUBLISH` messages are used by clients, in particular data sources, to send updates to the relevant topics.

There are three QoS, or Quality of Service, levels supported by MQTT. These QoS levels, namely 0, 1, and 2, provide different end-to-end message delivery guarantees. QoS 0 ensures that a receiver gets a published message *at most* once; QoS 1 ensures *at least* once; and QoS 2 guarantees *exactly* once. These QoS levels require different amounts of traffic to satisfy their guarantees, with QoS 0 using the least, just a single MQTT message. QoS 1 requires a minimum of two messages while QoS 2 requires a minimum of four messages.

III. Noctua System Design

Noctua is essentially a publish-subscribe broker that has been augmented with computational capabilities. This computational component provides IoT software developers with a framework that makes it easier to develop more efficient applications. In the following subsections, we will provide an overview of the Noctua architecture, discuss its implementation, and show a simple example of how it can be used.

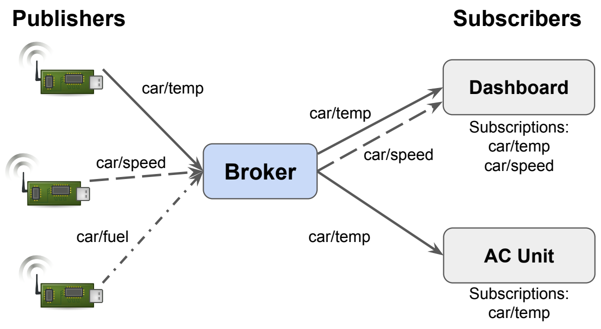


Fig. 1. Example data flow through an MQTT broker

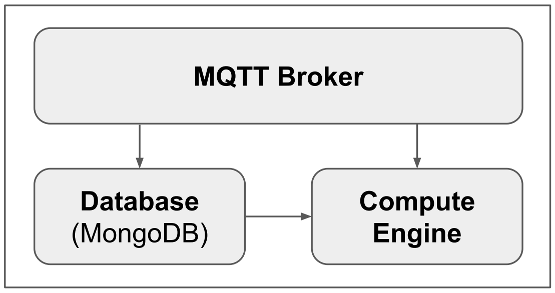


Fig. 2. Noctua Architecture

A. Architecture

At the core of Noctua is the Broker, as shown in Figure 2. The Broker is responsible for relaying messages to their subscribers, so by default it has access to the all of the data that may be relevant to an application. The Compute Engine uses this data to process any macros that are registered with it. We use the term *macro* to refer to the computation that an application is asking Noctua to perform on its behalf. A macro consists of one or more references to the data it depends on as well as some code representing the calculations to be performed. Macros can be created and even updated anytime during the operation of the system without disruption. An example macro will be shown in the following subsection. Macros are provided with an expressive syntax for accessing current and past values of data. To support this capability, Noctua incorporates a database which is used to store historical values of data that passes through it.

B. Implementation

Noctua is written in JavaScript and powered by Node.js [6], a server-side runtime environment built on Google's V8 JavaScript engine. Node.js is designed with an event-driven architecture that is highly optimized for network applications. This makes it well suited for the task at hand, as operations in Noctua are triggered when data passes through it. The Node.js plugin *aedes* provides Noctua with the framework for an MQTT broker. We use the HTTP protocol to handle macro operations, such as creation, updates, and deletion. MongoDB [7] is used as the database to store published values for future reference.

```
(Noctua.topic('home/tempF') - 32) * 5/9
```

Fig. 3. A macro for temperature conversion

```
{
    "name": "home/tempC",
    "code": "(Noctua.topic('tempF') - 32) * 5/9"
}
```

Fig. 4. JSON object for HTTP POST request to create macro

C. Macros

1) Language: Rather than create a new language, macros for Noctua are written in JavaScript, a scripting language that is already well known. The full language is supported, so all of the typical methods for flow control are available, e.g. *if* statements and *for* loops. We expose a Noctua-specific JavaScript Object to the runtime context of the macro such that it is able to pull in data that has already been published to the broker. We refer to this object as a *topic reference*. This data can then be treated like any other value in the code. The use of this object also allows Noctua to automatically determine a macro's dependencies, requiring no additional effort from the application developer. We consider a macro to be activated when a value arrives for one of its dependencies. A macro is only triggered when all of its dependencies are satisfied.

Figure 3 shows a simple example of a macro that converts temperature data from Fahrenheit to Celsius. This macro takes the most recent value published to the MQTT topic 'home/tempF' and performs some arithmetic operations on it to convert from one unit to another. The macro is triggered whenever a new value arrives on the 'home/tempF' topic. The computed value will then be published to a topic that corresponds to the name of the macro, which can be seen in Figure 4. In this case, the result will be published to the topic "Noctua/home/tempC". By default, the "Noctua" topic level is reserved for use by the broker and is prepended to the name of the macros to determine their associated topic for publishing.

The JSON object shown in Figure 4 shows the information needed to register a new macro with Noctua over HTTP. If a name is specified, it must be unique; otherwise a hash value will automatically be generated for the macro and returned. The code field is the only field that is required and it must contain valid JavaScript code.

2) Topic Reference Settings: Noctua supports an optional settings argument that can be specified in a topic reference. The options are listed in Table I. We will first explain what the options are and then provide an illustrative example to show why this feature is important.

The `cutoff` option can be be thought of as the amount of time a published value is acceptable for (with respect to a particular macro). So for example, if a cutoff value of 5 is specified, only a value received in the last 5 seconds of when the macro is triggered will be returned by Noctua. This option

TABLE I
TOPIC REFERENCE SETTINGS

Option	Default	Description
cutoff	0	Time in seconds that values are acceptable for
required	1	Minimum number of values to retrieve
limit	1	Maximum number of values to retrieve

```
function() {
    var t1 = Noctua.topic('home/front_temp', {
        cutoff: 60 })
    var t2 = Noctua.topic('home/rear_temp', {
        cutoff: 60 })
    return (t1+t2)/2
}()
```

Fig. 5. A macro for averaging two temperatures

can be applied to ensure that values used in calculations are either recent, or as we'll see later on, temporally correlated.

The `required` and `limit` options provide application developers with access to the historical values of a topic. The `required` option indicates the minimum amount of history needed by a macro for a topic while the `limit` option specifies the maximum. As an example, if the `required` option is set to 5, then an array of the 5 most recent values of a topic will be returned, assuming at least that many are available (otherwise the macro will not be triggered). If the `required` option is set to 5 and the limit `option` is set to 10, then an array of anywhere from 5 to 10 values may be returned. A macro may then iterate over those values to perform its calculation.

Figure 5 shows an example of a macro making use of the `cutoff` option. With a cutoff value of 60, the macro is specifying that only values received within the last minute should be considered. If the last temperature value for one of those topics arrived more than 60 seconds prior to when the macro is activated, then the macro will not be triggered and no value will be published.

The use of the `cutoff` option in this case provides two benefits. First, applying a cutoff prevents data used and published by our macro from being stale. For example, if the last temperature value published to a topic is from yesterday (perhaps a sensor has a long periodicity or lost connectivity), then that value may no longer be relevant to the application. Second, since the macro is applying a cutoff for all of its topics, the macro can ensure that the values are temporally correlated. Due to the decoupled nature of publish-subscribe systems, the last value published to two different topics may have arrived at wildly different times. The use of the cutoff, in this case, restricts the values used in this macro to have arrived within 60 seconds of each other.

D. Privacy

Noctua extends the purpose of the credentials provided in the MQTT `CONNECT` message. By default, MQTT only uses the credentials to authorize a connection to the broker. Once a user connects, he or she will have access to every topic and is free to subscribe and/or publish to them. Noctua takes the credentials a step further and enables an administrator to specify permissions on a per-topic basis. This allows for more fine-grained control over the data a user can and can't see. This level of granularity makes Noctua well suited for heterogeneous IoT systems where many devices may be sharing the same broker but don't necessarily have the same level of trust.

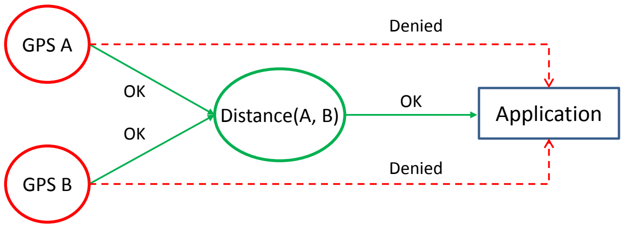


Fig. 6. Privacy protection

There is another feature that is gained from this high level of access control when it is combined with Noctua's computational capabilities. We illustrate this with an example. Imagine there is some particularly sensitive data being published to the broker, such as a person's GPS location. That person may not want others to know where he or she is, so access to that topic may be severely restricted. However, if there is an application that is not directly interested in that person's location, but rather the distance between that person and someone else, then that use case may be acceptable. That application could be restricted from directly accessing the person's location topic, but may be given permission to access the result it needs through Noctua's macro capability as depicted in Figure 6.

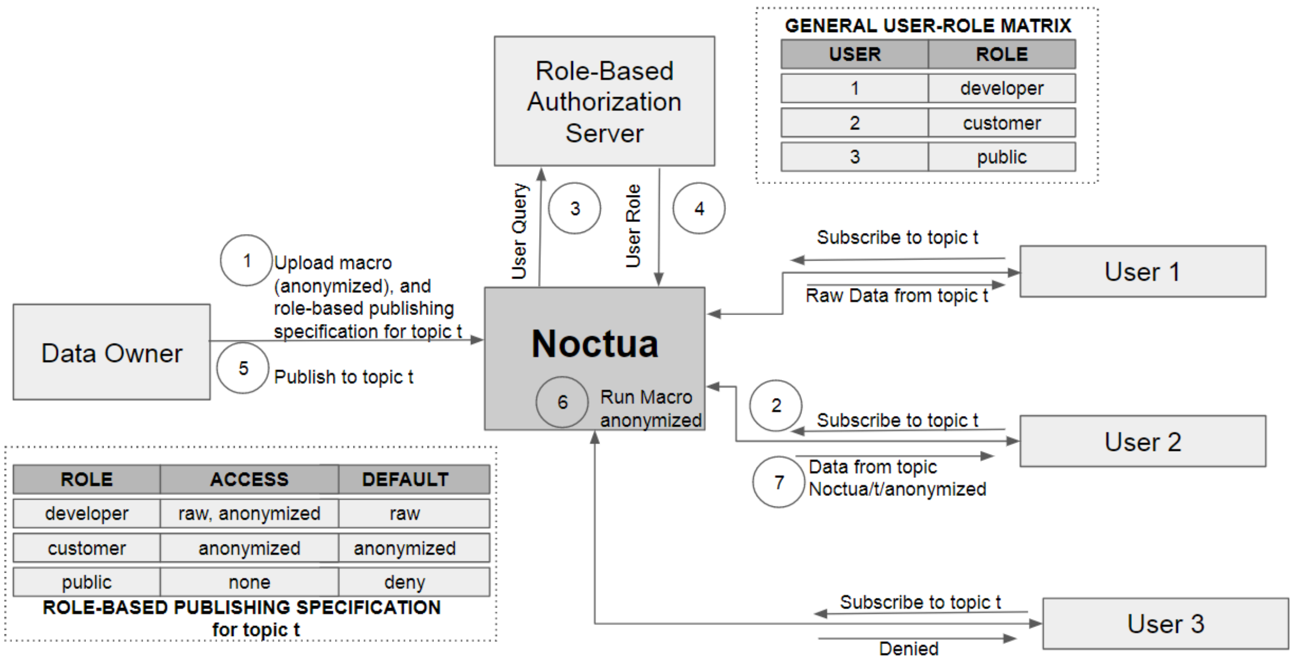
Essentially, what this approach to privacy allows is indirect access to sensitive information in some aggregated form. This protects sensitive information without severely limiting flexibility in application development. The kind of aggregation or filtering needed to make data anonymous is left to the discretion of the data owner/administrator as it is expected to be application specific.

Going beyond simple allow/deny permissions, Noctua can also automate the assignment of the "right stream of data" to each user based on their access role, which we refer to as role-based publishing. We describe this general framework in the following section.

IV. ROLE-BASED PUBLISHING

Noctua's ability to process real-time data streams can be used to provide differently processed streams to different subscribing users, as a function of their role, greatly facilitating the use of role-based access control for IoT applications. We refer to this feature of Noctua as role-based publishing. At the outset, we note that role based publishing is an optional functionality in Noctua that can be activated and instantiated for each topic.

Figure 7 shows how Noctua's role-based publishing works.The role-based authorization service could be implemented on the same system as Noctua or it could be an external



GENERAL USER-ROLE MATRIX

USER	ROLE
1	developer
2	customer
3	public

ROLE	ACCESS	DEFAULT
developer	raw, anonymized	raw
customer	anonymized	anonymized
public	none	deny

ROLE-BASED PUBLISHING SPECIFICATION for topic t

Fig. 7. Role-Based Publishing with Noctua

authorization/role-based access control server (e.g. a service built using OpenIAM [8]— or OAuth [9]). The service is able to provide Noctua with the role associated with a given user.

The data owner can upload one or more macros to Noctua to process the raw data stream, and also provides a role-based publishing specification (RBPS) that Noctua can use to determine which macros/processed streams can be accessed by which user (once the user's role has been determined). The RBPS may specify multiple streams that a role is allowed to access, but must also specify one of these streams as the default.

When a user subscribes to an original topic, Noctua makes a call to the role-based authorization service to determine that user's role, then sends data to the user corresponding to the default processed stream that is specified for the role, by effectively subscribing the user to the permitted topic (original or processed). Some roles may even be denied access to any data from the stream. If a role is allowed access to multiple streams, the user may further directly subscribe to other topics corresponding to any of the permitted streams.

Figure 7 illustrates a possible flow for a given topic and user.

1) The owner of data for topic t provides a macro called "t/anonymize" for anonymizing that data stream. This could be implemented, for example by adding noise to the data, removing certain labels, or applying a threshold to generate a coarse-grained version of the data. The owner also uploads to the Noctua server (as a JSON object posted using HTTP) a role-based publishing specification (RBPS) for this topic. This is a table, as shown in the bottom left of the Figure 7, that specifies which data streams each possible role is permitted to access, and for each role also specifies a default stream.
2) User 2 sends a subscription request for topic t.
3) Noctua queries the authorization service about User 2. The authorization service uses the user-role matrix to determine this user's role.
4) The service informs Noctua that the role for User 2 is "customer"
5) The data owner publishes data to topic t on a streaming basis.
6) The data from the publisher is available as topic t on Noctua and also processed into a new macro-based topic, called "Noctua/t/anonymized" (using the macro provided by the data owner in Step 1).
7) Since the second row of the RBPS for topic t specifies that the default stream for customers is anonymized, User 2 will receive data on the topic "Noctua/t/anonymized" from Noctua whenever it is available.

In this example, we see that User 2's subscription request for topic t is essentially automatically translated by Noctua to another stream of anonymized data. User 1 would get access to the raw stream by default due to his or her role as a "developer." However, User 1 can also directly subscribe to and receive data on "Noctua/t/anonymized" because the "'developer" role is permitted access to the anonymized stream as well. If a user with a role that is not authorized to access either the raw or anonymized streams subscribes, it is denied the subscription entirely (and cannot access "Noctua/t/anonymized" directly either).

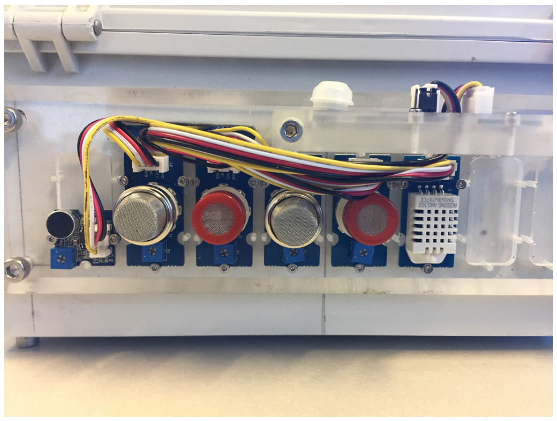


Fig. 8. Various sensors on a CCI Testbed node

V. Evaluation

We have devised a set of experiments to analyze the capabilities and performance of Noctua as compared to traditional methods of implementation of IoT systems. In our first experiment, we evaluate Noctua on the campus-wide CCI IoT Testbed currently under development at the University Southern California [10]. Next we apply Noctua to localize a person walking in an indoor environment. Finally, we take a look at the implications of Noctua's privacy features.

A. Hardware

One of the CCI Testbed nodes is shown in Figure 8. At the core of each node is a Raspberry Pi 3 computer [11], which handles data collection from all locally attached analog and digital Grove sensors [12]. Each node is equipped with several sensors, including temperature, humidity, light, noise, and a variety of gas sensors as shown in the figure. The nodes are connected to USC's campus network through WiFi.

B. Weighted Moving Average

In this experiment, we look at creating a weighted moving average of data from three CCI testbed nodes, specifically their temperature readings. A moving average is a simple technique for smoothing out time series data. Let $f(t)$ represent the temperature at some time index t. For each node, we seek to perform the following computation, shown in Equation 1, over its past three values. This moving average gives more weight to more recent values.

$$WMA = \frac{3}{6}f(t) + \frac{2}{6}f(t-1) + \frac{1}{6}f(t-2) \tag{1}$$

The testbed is configured such that each node is publishing its temperature on an individual MQTT topic, one based on its hostname. The nodes are configured to transmit their temperature value once every second. QoS 0 is used for all messages so that we can determine a lower bound on messages required. Once we've obtained the weighted moving average for each topic, we then average the three resulting values together using equal weights. Figure 9 shows the macro used for this experiment.

As shown in Figure 9, we are using the `required` option to specify that we need the last three values for each topic we've referenced. It should be noted that whenever the `required` or `limit` option specified is greater than 1, the value returned for a topic reference will be in array form. We are therefore able to iterate over the elements as we have done in the macro.

For comparison purposes, we've implemented this moving average calculation in three different ways, all based on publish-subscribe messaging. We refer to these implementations as: a) local processing, where the subscriber itself performs the calculation; b) application service, where a single standalone service performs the calculation on behalf of any subscribers; and c) Noctua, where the broker itself performs the calculation. The topology for these implementations are shown in Figure 12.

1) Message Complexity: Figure 12 shows the message complexity in terms of the number of MQTT `PUBLISH` messages required between devices until the first output of the calculation is available, assuming the system starts from scratch (no topic history). In all cases, we need at least three values from each sensor, resulting in at least nine `PUBLISH` messages before any calculation can take place, regardless of the implementation. For the first case (Figure 12a), since the subscriber performs the calculation itself, it needs all nine messages forwarded to it resulting in a total of 18 messages, as shown in Figure 10. Second, in Figure 12b, the application service needs to receive all of the sensor data. The service then publishes the result which gets forwarded to the subscriber. This results in a total of 20 messages. Lastly, we can see the results for the Noctua implementation in Figure 12c. The calculation is performed inside the broker itself, so only a single messsage, the result, needs to be transmitted to the subscriber. This results in a total of 10 messages, which is also the theoretical minimum for a broker-based system.

2) Delay: To determine the overhead associated with using the Noctua framework, we measured the end-to-end delay of the three implementations. Specifically, we measured the amount of time it takes to get a calculated result once the first MQTT `PUBLISH` message is sent. For each measurement, all three nodes were triggered to start sending their values simultaneously so that the results are comparable. Figure 11 shows the outcome of this experiment. The results are the averaged over a 100 iterations for each implementation.

The maximum difference between the delays of the implementations is small, $< 0.2s$. The local implementation performed the best with an average delay of $5.36s$, followed closely by Noctua at $5.47s$. With a delay of $5.54s$, the application service took the longest, which makes sense as there is an additional link traversal involved with that implementation relative to the others, as can be seen in Figure 12b. For this moving average example, we can see that there is no significant cost to using Noctua.

3) Scalability: To get a better sense of how Noctua may perform for different applications, we investigate its scalability in terms of message complexity. In Figures 13 and 14, we

```
function(){
  var temps = [];
  temps.push(Noctua.topic('ee499_7/temp',
     { required: 3 }));
  temps.push(Noctua.topic('ee499_8/temp',
     { required: 3 }));
  temps.push(Noctua.topic('ee499_9/temp',
     { required: 3 }));

  var weights = [3, 2, 1];
  var denom = weights.reduce((a, b) => a + b, 0);

  var avg = 0;
  for (var i = 0; i < temps.length; ++i) {
    var weighted_sum = 0;

    for (var j = 0; j < weights.length; ++j) {
      weighted_sum += temps[i][j] * weights[j];
    }
  avg += weighted_sum/denom;
}
return avg/temps.length;
}()
```

Fig. 9. Macro for a weighted moving average

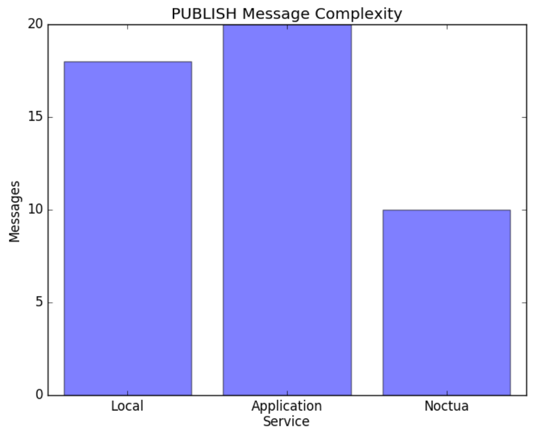


Fig. 10. Minimuim MQTT `PUBLISH` messages required until first result

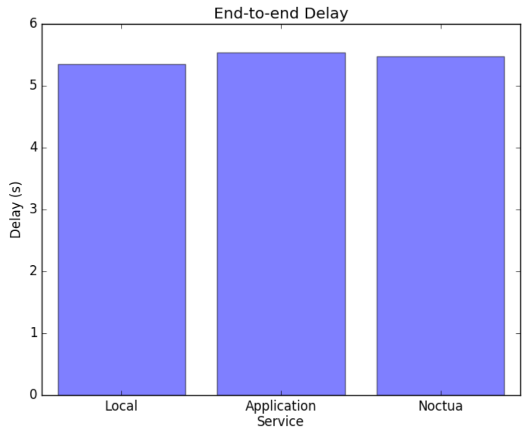


Fig. 11. Delay between first `PUBLISH` message and first result

calculated the number of `PUBLISH` messages required for a varying number of sensors and a varying number of subscribers, respectively, for the different implementations. For Figure 13, we imagine that the moving average example was extended to cover an increasing number of sensors. As before, we still require three values from each sensor for the formula shown in Equation 1. In this figure we assume there is still a single subscriber. We can see that the message growth rate is linear for all three implementations, but that the Noctua implementation grows much more slowly than the other two, demonstrating Noctua's superiority in terms of scalability.

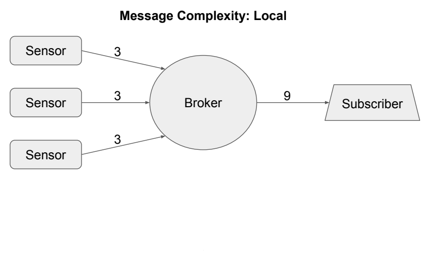
In Figure 14, we take a look at the opposite case. Instead of varying the number of sensors, we vary the number of subscribers interested in the output of the moving average. The number of sensors is fixed at 50. Here we can see that the local implementation performs much worse than the other two as we would expect. It is simply not practical to send every published message to every subscriber. The application service and Noctua implementations have an almost horizontal curve, although they are growing. Once again Noctua is able to achieve the best performance in terms of `PUBLISH` messages.

C. Application Example: Localization as a Service

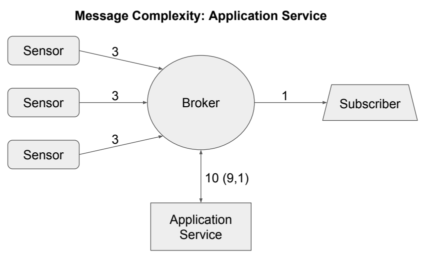
To demonstrate that Noctua is robust and can support applications involving calculations that are much more complicated than a moving average, we created a macro that can process RSSI values and make predictions as to where a person is located. The localization macro is shown in Figure 15. The idea is to have a user device publish its RSSI readings from multiple beacons to the broker, and have the Noctua broker use the macro to estimate the location and send back the estimated location stream back to the user. This demonstrates how Noctua can be used to rapidly build and deploy "localization as a service" (and by extension, many other similar applications where data analytics or machine learning algorithms can be used to refine and transform raw data into more meaningful insights in real-time).

To show the macro functioning and evaluate its computational cost, we perform experiments with simulated data. In this experiment, we simulate a person walking through an indoor environment with a wireless device. This device measures the received signal strength, or RSSI, from multiple beacons located within the space. The blue line in Figure 16 shows the path that this person takes, while the hollow diamonds indicate the location of the beacons. We apply the log-distance path loss radio propagation model [13] to simulate the signal strengths the person would receive as they move about the space. This model is shown in Equation 2. P_{RX} and P_{TX} represent the received and transmitted signal powers in dBm, respectively; K represents the path loss in dB at the reference distance d_0; and X_g is a zero mean Gaussian random variable that represents fading.

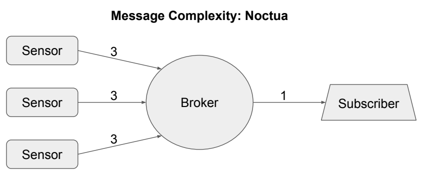
$$P_{RX} = P_{TX} - K + 10\eta \; log_{10}\frac{d}{d_0} + X_g \tag{2}$$



(a) Local processing



(b) Processing through application service



(c) Processing through Noctua

Fig. 12. Various implementations of a three-sensor system

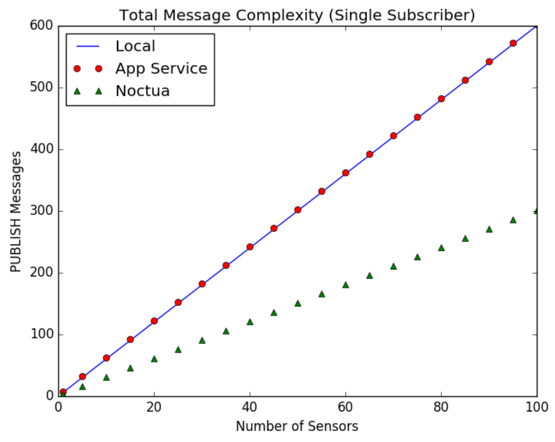


Fig. 13. MQTT `PUBLISH` messages required for varying number of sensors

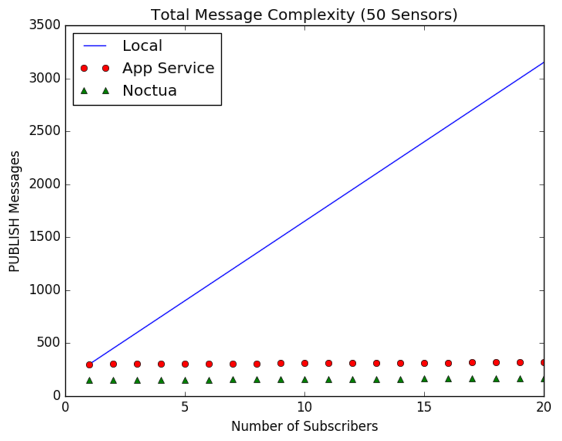


Fig. 14. MQTT `PUBLISH` messages required for varying number of subscribers

For localization we apply the well-known maximum-likelihood (ML) estimation technique [14]. Our implementation of this technique evaluates the probability of observing the received RSSI values at different locations within the space and chooses the location with the highest probability as the prediction. We use the center point of each square in the grid as our search space. Figure 17 shows the predictions made

```
function () {
  //omitted: definitions of beacons, ETA, SIG, K, TX
  var readings = Noctua.topic('person/rssi')

  result = new Array(readings.length)
  for (var i=0; i < readings.length; ++i) {
    result[i] = new Array(10);
    for (var j=0; j < 10; ++j) {
      result[i][j] = new Array(10)
    }
  }

  for (var b=0; b < readings.length; ++b) {
    rx = readings[b]

    for (var j=0; j < 10; ++j) {
      for (var i=0; i < 10; ++i) {
        y = 5 + 10*j
        x = 5 + 10*i

        d = Math.sqrt(Math.pow(beacons[b][0] - x, 2)
             + Math.pow(beacons[b][1] - y, 2))
        fade = TX - rx - K - 10*ETA * Math.log10(d)
        pdf = (1/Math.sqrt(2 * Math.PI * Math.pow(
            SIG, 2))) * Math.exp(-Math.pow(fade, 2)
            /(2 * Math.pow(SIG, 2)))

        result[b][j][i] = pdf
      }
    }
  }

  max = 0
  max_ind = []

  for (var j=0; j < 10; ++j) {
    for (var i=0; i <  10; ++i) {
      combined = 1

      for (var b=0; b < readings.length; ++b){
        combined *= result[b][j][i]
      }

      if (combined > max) {
        max = combined
        max_ind = [i, j]
      }
    }
  }

  max_pos = [5 + 10*max_ind[0], 5 + 10*max_ind[1]]
  return max_pos
}()
```

Fig. 15. Macro for localization using maximum-likelihood estimation

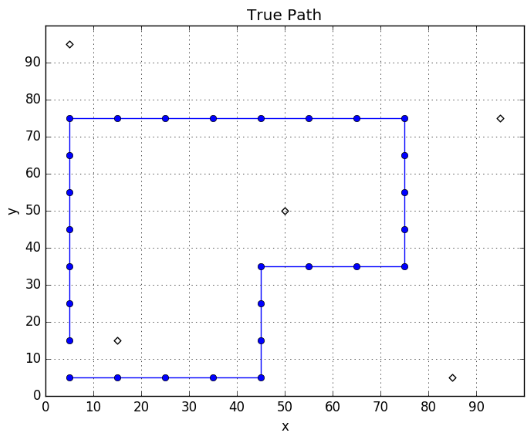


Fig. 16. A person's path through an indoor environment

by Noctua. As the figure shows, it is generally the case that using more beacons improves the prediction accuracy. Figure 18 shows the execution time of the Noctua macro when it uses a varying number of beacons in its calculation. The execution times are averaged over 100 iterations.

D. Impact of Role-based Publishing

We next consider the performance impact of role-based publishing in Noctua, when activated for a topic. When a user makes a subscription to a topic that has a role-based publishing specification (RBPS) associated with it, Noctua incurs additional processing and communication time before data can be sent on that topic. This pertains to the communication and processing needed to contact and hear back from the authorization server (which may be external to Noctua) and the processing incurred to determine and set up publication from the default data stream for the role that the given user corresponds to. During this additional time before the publication stream is set up, however, it is possible that the publisher sent data items to the broker that Noctua doesn't deliver to the user.

We present a brief mathematical analysis of the expected number of lost data items due to the latency associated with role-based publishing. We model the role-based publishing set up latency, the sum of the query latency for user roles plus the local processing incurred to determine and set up the default topic for the user, as being a random variable T_{RBP} that is exponentially distributed with mean (τ). We also assume that the data for the topic stream (raw or processed) is periodically sent at a deterministic frequency of ρ times per second. Then the number of lost packets $L = \lfloor T_{RBP} \cdot \rho \rfloor - 1$. It can be shown that $L+1$ is a geometric random variable, with success parameter $p = 1 - e^{-\frac{1}{\tau \cdot \rho}}$, and hence L has the following expected value:

$$E[L] = \frac{1}{1 - e^{-\frac{1}{\tau \cdot \rho}}} - 1 \tag{3}$$

This is shown numerically in Figure 19. It can be seen that the data loss increases essentially linearly in the product of τ and ρ, which can be significant under some circumstances. Such data loss occurs to a significant extent only if a) a role-based publication specification is provided for a data stream, b) querying for user roles incurs non-trivial latency (perhaps because it involves calls to a cloud-based server), and c) the published data stream has a very high frequency.

VI. Related Works

From the days of wireless sensor networks (WSNs), there has been an effort towards developing effective programming abstractions that has carried over into the IoT space. These efforts have resulted in various solutions ranging from new programming languages to macroprogramming frameworks. Here we discuss some of these works that bear similarity to our work on Noctua.

The authors of T-Res [15] present a programming abstraction that facilitates in-network processing in IoT-based WSNs. In T-Res, the input, output, and processing components of Python tasks are decoupled and presented as network resources using the CoAP protocol [16], allowing them to be reconfigured dynamically. One key difference between T-Res and Noctua, is that T-Res places the burden of managing asynchronicity of input data on the developer, who is required to maintain state between the executions of their tasks. With Noctua, developers can explicitly specify the constraints on the data they need and rely on the framework to trigger their macro only when these constraints are satisfied. Noctua also enables users to access arbitrary amounts of historical data rather than just the last value.

PyoT [17] is a macroprogramming framework that, like T-Res, makes use of CoAP and Python. However, this framework assumes the existence of metadata on devices to enable searches and avoids the need to address devices directly. While this framework is designed for more interactive use, through a web interface that it provides, Noctua is focused on data-driven applications and doesn't expect much user involvement beyond its initial setup.

Flask [1] presents a language for data-driven sensor network applications. This language can be used to “wire” up data flow graphs, which consist of the operations that need to be performed to compute the desired output. A limitation of Flask is that communication and processing all get statically defined at compile time, leading to a strong coupling between data sources and sinks. It is also not possible to change the processing at runtime as all participating devices would need to be reprogrammed.

There is a commercial product called PubNub that provides publish-subscribe messaging infrastructure as a service. One of the features supported by this service is known as PubNub Functions [18] and it appears to implement a form of in-networking processing. It allows users to manipulate their data in various ways as it is flowing through the network. These Functions are written in JavaScript. There are also third-party Functions, referred to as BLOCKS, that can be used as

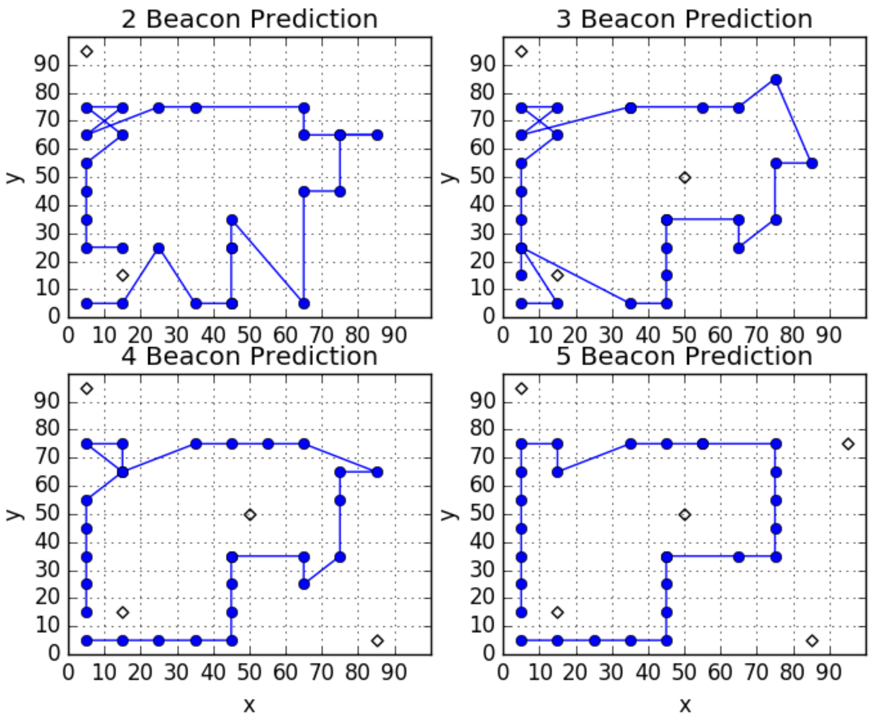


Fig. 17. Localization using maximum-likelihood (ML) estimation using macro on Noctua

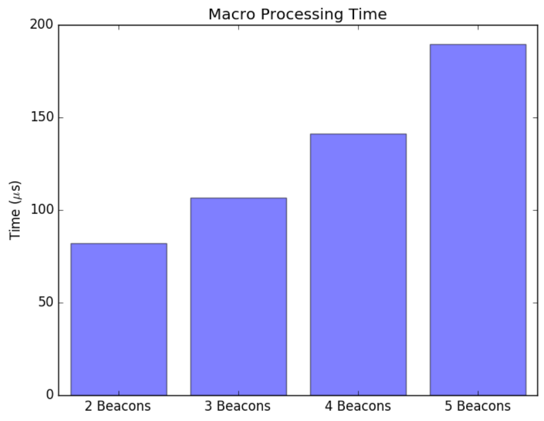


Fig. 18. Average calculation times for location prediction for macro on Noctua

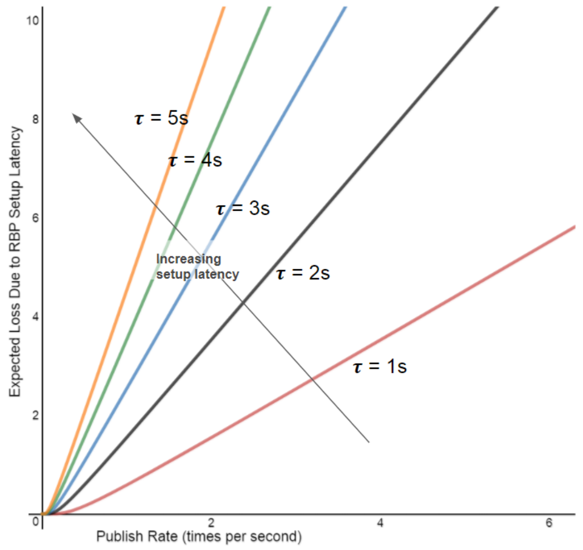


Fig. 19. Average data loss due to setup latency associated with role-based publishing on Noctua

well. Since PubNub Functions is a proprietary product, it is difficult to speculate on what is going on behind the scenes, but it demonstrates that there is a real demand for this type of functionality.

Node-RED [19] is a web-based dataflow tool that enables users to quickly create interactions among supported devices and web-based services through a visual editor. While it is built on similar technology, i.e. Node.js, Node-RED is not focused on publish-subscribe messaging and does not tackle the privacy issues we address with Noctua. Node-RED's Flows also cannot be dynamically reconfigured at runtime like Noctua's macros.

With regards to publish-subscribe messaging, there are a variety of software products that provide such capability. One such product is Apache Kafka [20], an open-source stream processing platform. While Kafka provides publish-subscribe semantics, it is more of an enterprise-grade messaging queue than it is a replacement for a lightweight protocol such as MQTT. Kafka is suited for operation in data center clouds and not designed for use at the network edge or in otherwise constrained environments, which we target with Noctua.

VII. Conclusion

This paper presented Noctua, a framework enabling a new messaging paradigm we refer to as publish-process-subscribe. This paradigm addresses the observation that many IoT applications actuate on processed forms of data rather than just the raw data itself. This leads to a waste of network resources as raw data is shipped across the network unnecessarily. Noctua provides a mechanism, which we refer to as macros, by which application developers can address this issue without being burdened with managing low-level communication details.

In summary, the goals of Noctua are to ease application development, while reducing network congestion, improving network lifetime, and protecting data privacy. Noctua accomplishes these goals through the use of macros and flexible access controls. Macros are portions of JavaScipt code that are offloaded to the Noctua broker. We have demonstrated that these macros are simultaneously effective at reducing the computational strain on edge devices and improving network congestion. And we have discussed how topic-level role-based permissions and role-based privacy-oriented real time data processing allow Noctua to support a diverse set of applications.

We plan to make our reference implementation of Noctua publicly available as open source upon publication of this work. In future work, we would like to consider further enhancements of Noctua including distributed processing, and a more comprehensive evaluation through a large-scale deployment in a project focused on IoT technologies for smart cities.

References

[1] G. Mainland, M. Welsh, and G. Morrisett, “Flask: A language for data-driven sensor network programs,” *Harvard Univ., Cambridge, MA, Tech. Rep. TR-13-06*, 2006.

[2] A. Awan, S. Jagannathan, and A. Grama, “Macroprogramming heterogeneous sensor networks using cosmos,” in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 159–172.

[3] A. Banks and R. Gupta, “Mqtt version 3.1.1,” *OASIS standard*, vol. 29, 2014.

[4] 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.

[5] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, “Mqtt-s: A 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.

[6] S. Tilkov and S. Vinoski, “Node. js: Using javascript to build high-performance network programs,” *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.

[7] “Mongodb,” https://www.mongodb.com/.

[8] “Openiam,” http://www.openiam.com/.

[9] P. Fremantle, B. Aziz, J. Kopecký, and P. Scott, “Federated identity and access management for the internet of things,” in *Secure Internet of Things (SIoT), 2014 International Workshop on*. IEEE, 2014, pp. 10–17.

[10] “Testbeds — cci,” https://cci.usc.edu/index.php/research/testbeds/.

[11] “Raspberry pi 3 model b,” https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[12] “Grove - seeed studio,” https://www.seeedstudio.com/category/Grove-c-1003.html.

[13] A. F. Molisch, *Wireless communications*. John Wiley & Sons, 2012, vol. 34.

[14] N. Patwari, R. J. O’Dea, and Y. Wang, “Relative location in wireless networks,” in *Vehicular Technology Conference, 2001. VTC 2001 Spring. IEEE VTS 53rd*, vol. 2. IEEE, 2001, pp. 1149–1153.

[15] D. Alessandrelli, M. Petraccay, and P. Pagano, “T-res: Enabling reconfigurable in-network processing in iot-based wsns,” in *Distributed Computing in Sensor Systems (DCOSS), 2013 IEEE International Conference on*. IEEE, 2013, pp. 337–344.

[16] Z. Shelby, K. Hartke, and C. Bormann, “The constrained application protocol (coap),” *IETF*, 2014.

[17] A. Azzara, D. Alessandrelli, S. Bocchino, M. Petracca, and P. Pagano, “Pyot, a macroprogramming framework for the internet of things,” in *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*. IEEE, 2014, pp. 96–103.

[18] “Pubnub functions for serverless compute,” https://www.pubnub.com/products/functions/.

[19] “Node-red,” https://nodered.org/.

[20] “Apache kafka,” https://kafka.apache.org/.