IoT Middleware: A Survey on Issues and Enabling Technologies

Anne H. Ngu, Mario Gutierrez, Vangelis Metsis, Surya Nepal,
and Quan Z. Sheng, *Member, IEEE*

***Abstract*—The Internet of Things (IoT) provides the ability for humans and computers to learn and interact from billions of things that include sensors, actuators, services, and other Internet-connected objects. The realization of IoT systems will enable seamless integration of the cyber world with our physical world and will fundamentally change and empower human interaction with the world. A key technology in the realization of IoT systems is middleware, which is usually described as a software system designed to be the intermediary between IoT devices and applications. In this paper, we first motivate the need for an IoT middleware via an IoT application designed for real-time prediction of blood alcohol content using smartwatch sensor data. This is then followed by a survey on the capabilities of the existing IoT middleware. We further conduct a thorough analysis of the challenges and the enabling technologies in developing an IoT middleware that embraces the heterogeneity of IoT devices and also supports the essential ingredients of composition, adaptability, and security aspects of an IoT system.**

***Index Terms*—Internet of Things (IoT), IoT middleware, IoT service discovery, security and privacy.**

I. Introduction

THE INTERNET of Things (IoT) is a domain that represents the next most exciting technological revolution since the Internet [1]–[4]. IoT will bring endless opportunities and impact in every corner of our planet. With IoT, we can build smart cities where parking spaces, urban noise, traffic congestion, street lighting, irrigation, and waste can be monitored in real time and managed more effectively. We can build smart homes that are safe and energy-efficient. We can build smart environments that automatically monitor air and water pollution and enable early detection of earthquakes, forest fires, and many other devastating disasters. IoT can transform manufacturing, making it leaner and smarter. According to CBS news, there have been nearly 600 bridge failures in the USA since 1989. A large number of bridges in every state are really a danger to the traveling public. IoT-enabled sensors can monitor the vibrations and material conditions in bridges (as well as buildings and historical monuments) and provide early warning that would save numerous human lives. IoT is going to create massive disruption and innovation in just about every industry segment imaginable.

While the IoT offers numerous exciting potentials and opportunities, it remains challenging to effectively manage things to achieve seamless integration of the physical world and the cyber one [1], [5], [6]. Many IoT middleware and connectivity protocols are being developed and the number is increasing each day. For example, message queuing telemetry transport (MQTT), constrained application protocol (CoAP), and Bluetooth low energy (BLE) are popular connectivity protocols designed specifically for IoT devices. However, the plethora of IoT connectivity protocols and middleware are not facilitating the ease of connecting IoT devices and interpreting collected data from them. This is compounded by the fact that each IoT middleware advocates different programming abstraction and architecture for accessing and connecting to IoT devices. For example, in the global sensor network (GSN) [7] project, the concept of *virtual sensor*, which is specified in XML and implemented with a corresponding wrapper, is provided as the main abstraction for developing and connecting a new IoT device. In the TerraSwarm project [8], an *accessor* design pattern implemented in Javascript is proposed as the main abstraction. In the Google Fit project [9], no particular high level abstraction is provided for encapsulating a new device type. The system is preprogrammed to support a fixed set of IoT devices, which can be accessed by representational state transfer (REST) APIs [10]. Adding an IoT device not already supported requires expert Java programming experience in extending Google Fit's `FitnessSensorService` class. The current state-of-the-art support for IoT application development is application specific which is equivalent to the scenario where every IoT device requires a different Web browser for connection to the Internet as echoed by Zachariah *et al.* [11] in the paper entitled "The Internet of Things Has a Gateway Problem."

In this paper, we survey the state-of-the-art middleware solutions in realizing IoT applications. Several surveys on IoT middleware have been published such as [12]–[15]. To the best of our knowledge, these surveys overview IoT middleware only from specific perspectives and none of them address the more recent trend of light-weight plug-and-play or cloud-based IoT middleware. The aim of this paper is to provide a

Manuscript received June 27, 2016; accepted September 15, 2016. Date of publication October 4, 2016; date of current version February 8, 2017. This work was supported in part by the National Science Foundation under the Research Experiences for Undergraduates Program (CNS-1358939) at Texas State University and in part by NSF-CRI under Award 1305302.

A. H. Ngu, M. Gutierrez, and V. Metsis are with the Department of Computer Science, Texas State University, San Marcos, TX 78666 USA (e-mail: angu@txstate.edu; mag262@txstate.edu; vmetsis@txstate.edu).

S. Nepal is with the Data61, CSIRO, Sydney, NSW 2122, Australia (e-mail: surya.nepal@data61.csiro.au).

Q. Z. Sheng is with the Department of Computing, Macquarie University, Sydney, NSW 2109, Australia (e-mail: michael.sheng@mq.edu.au).

Digital Object Identifier 10.1109/JIOT.2016.2615180

better understanding of current research and challenges of IoT middleware systems. The main contributions of this paper are as follows.

1) A classification of the different types of architecture of IoT middleware.
2) A comparative analysis of emerging IoT middleware systems for each architecture type.
3) An assessment of the key research challenges, such as composition, adaptability and security, in building the next generation IoT middleware for rapid composition of IoT applications.

The remainder of this paper is organized as follows. We first motivate the need for an IoT middleware based on our experiences in building a real-time blood alcohol content (BAC) predictor using smartwatch sensor data (Section II). We then discuss our observation of the three key software architectures of IoT middleware and present a description of the similarities and differences among the three architectures (Section III). In Section IV, we provide a detailed analysis of the different IoT middleware architectures by surveying eight existing IoT middleware systems with respect to fulfilling the key functionalities required by BAC-like IoT applications which are device abstraction for data collection, composition for visualization and analysis, service discovery for opportunistic integration, security and privacy, and data persistency. We compare these three types of IoT middleware by providing an example implementation of data collection of a Phidgets sensor in GSN, Google Fit, and Ptolemy accessor host. We then outline the key research challenges in developing an IoT middleware that will enable a scientist or a health professional to configure/compose a BAC-like IoT application which is adaptable, open, and secure (Section V). Finally, we overview and discuss the relevant work in Section VI and provide some concluding remarks in Section VII.

II. Use Case for IoT Middleware

IoT applications fall into two general categories: 1) ambient data collection and analytics and 2) real-time reactive applications. The first category of applications generally involves collecting sensor data from a variety of sensors (e.g., wearable devices), process them offline to gain actionable information (e.g., a model) and then run the model as a predictor for new data collected from the sensor in the future. The second category of applications involves real-time reactive systems such as autonomous vehicle or manufacturing processes where the systems make real-time decisions based on observed sensor values. The former category of applications is growing rapidly especially in the healthcare domain where personalized health tracking and monitoring has become vital to improved and affordable healthcare.

In this section, we motivate the need for an open, lightweight, secure, IoT middleware based on our experience in implementing an ambient data collection and analytics IoT application that can predict BAC using smartwatch sensor data [16]. Below, we briefly outline the motivation for creating this IoT application and show how this application, and in fact all IoT applications in this category, can benefit from an IoT middleware.

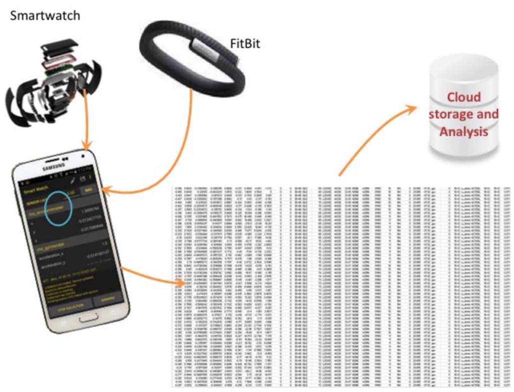


Fig. 1. Infrastructure for data collection and analysis.

Drunk driving is a dangerous, worldwide problem. This problem is not only a hazard to the drunk drivers, but also to pedestrians and other drivers. At dangerous levels of intoxication, it can be difficult to judge one's own drunkenness. Instead, it would be better to get a definitive measurement of the BAC, or simply a binary response: "drunk" or "not drunk". Compact breathalyzers are probably the best option at the moment, but these are not discreet and require deliberate actions by the user. The other option is to use a smartphone application to manually calculate BAC, but this demands a greater deal of involvement from the user (remembering how many drinks they have taken in a social setting). To be practical, it would be useful to have some sort of noninvasive and accurate monitoring system that will warn its users if they become too intoxicated. This system can also be used to warn friends and family, or prevent the operation of the drinker's car by integrating with car's ignition device.

We investigated the prediction of intoxication level from smartwatch sensor data via building a secure IoT application from the ground up. The collected data need to be stored locally and transmitted to cloud storage for analysis. The availability of local storage is important to avoid the unpredictable latencies from wireless transmission of data to the cloud. The data needs to be secured not only at rest, both at local and cloud storage, but also in transit. This demands the end to end security from the edge/sensor to the cloud data center.

After data collection, data is processed in the cloud to determine whether there is any relationship between each type of sensor values and the recorded BAC values. To achieve that, the collected data must be preprocessed and visualized. Then, various machine learning algorithms are applied to the collected sensor data to obtain the most accurate predictor. Fig. 1 shows the main infrastructure used for a generic data collection and analysis system. The data collection application running on the smartphone (which is also known as the gateway) implements a set of Java classes to deal with the low-level details of data collection process such as managing the various threads for collecting the various sensor values from the Microsoft Band smartwatch (known as the edge device) or other devices such as Fitbit. The data collection application also performs some aggregation on the collected data before sending them for archival on the cloud. The data analytics is performed entirely in the cloud which has the high performing computation engine and various big data analytics and visualization tools. Once a model is derived from the data analysis, it is saved and integrated with the BAC application as a predictor.

The following are the key computation units we observed from implementing the BAC application.

1) The Microsoft Band smartwatch must first be virtualized as a software component to the BAC smartphone application in order to connect and obtain data from all available sensors on the Microsoft Band smartwatch. A device abstraction component is thus needed to hide users from the low level implementation details of the networking protocols and communication capabilities of different physical sensors.
2) The real-time interaction between the BAC application and the physical device must be supported. Data is usually delivered as infinite streams in time-stamped order from the devices. A stream, event processing or aggregation service is thus an important component. Stream processing provides complex event detection that turns the collected data (usually in large quantity) into useful actionable information. Aggregation can provide more meaningful data for analysis. For example, in the case of collecting accelerometer data, the three most recent values were averaged with linear weighting rather than just using the last value.
3) A monitoring or visualization service is needed to allow users to monitor/control the state of the physical devices as well as to manage when and how often the collected data should be archived to the cloud for further analysis or processing. This component should also provide notification and subscription services for the timely delivery of IoT status to users, in this case an alert for being intoxicated.
4) An IoT application can generate a large amount of data that need to be processed and archived, so the ubiquitous connection to a cloud infrastructure is needed for data analytics and archiving.
5) The security and privacy component is needed to provide the integrity of the collected data (stream) and to ensure that the user's privacy is not violated. Users should have the option to archive the collected data in a storage medium of choice and only be able to connect to authenticated/certified IoT devices.
6) A composition engine (also called a rule engine in some systems) is necessary to enable users to combine analytics services from the cloud, from data services in other gateways (PhidgetInterfaceKit and Arduino) or other IoT devices (car's ignition device) without low-level programming.

A data collection and analytics system for tracking environment pollution in a building would involve a similar set of computation units as illustrated in the BAC prediction application; albeit the type of sensors at the edge will be Mica motes with desktop or laptop serving as the gateway. The data collected will also be pushed to the cloud storage or a backend database. A similar set of analysis and visualization tools are needed for the analytics. In summary, the logical requirement for environment monitoring and BAC monitoring are the same. Having to develop two separate applications for each of the above applications with dedicated set of resources not only increases the cost and the time for development but also leads developers with the additional challenges of security and privacy surrounding data and ToT devices access. There is thus a need to develop an open, lightweight, adaptable, secure IoT middleware that serves as a bridge across a variety of IoT devices and applications. Such a middleware will enable a scientist or a health professional to configure/compose a new secure IoT application for performing the data collection and analysis relevant to his/her context without any low-level programming.

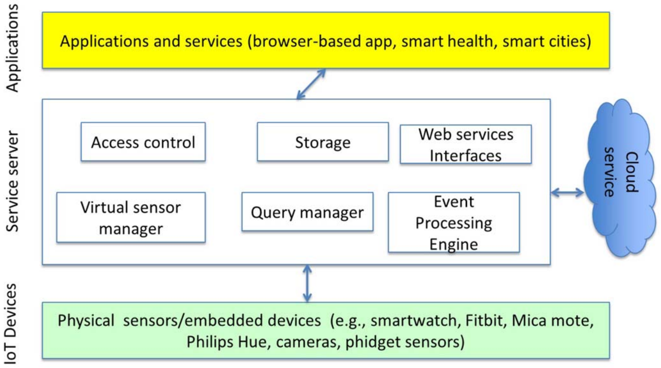

Applications
Applications and services (browser-based app, smart health, smart cities)
Service server
Access control
Storage
Web services Interfaces
Virtual sensor manager
Query manager
Event Processing Engine
Cloud service
IoT Devices
Physical sensors/embedded devices (e.g., smartwatch, Fitbit, Mica mote, Philips Hue, cameras, phidget sensors)

Fig. 2. Service-based IoT middleware.

III. Architecture of IoT Middleware

Existing architectures for IoT middleware form three class types, from our observation. The first type, which we refer to it as a *service-based* solution, generally adopts the service-oriented architecture (SOA) [17] and allows developers or users to add or deploy a diverse range of IoT devices as *services*. The second type, which is known as *cloud-based* solution, limits the users on the type and the number of IoT devices that they can deploy, but enables users to connect, collect, and interpret the collected data with ease since possible use cases can be determined and programmed *a priori*. The third type is the *actor-based* framework that emphasizes on the open, plug and play IoT architecture. A variety of IoT devices can be exposed as reusable actors and distributed in the network.

Fig. 2 depicts a service-based IoT middleware. It is a three-layered architecture adopted by the OpenIoT [18], an European Union project to standardize IoT platforms. This architecture consists of a physical plane (sensors and actuators), a virtualized plane (server or cloud infrastructure), and an application plane (utility). The main computational units are available in the middle layer or the virtualized plane. The generic services available in the middle layer range from access control, storage management to an event processing engine. These services support the data collection part of the BAC-like IoT applications, but not the analytic part.

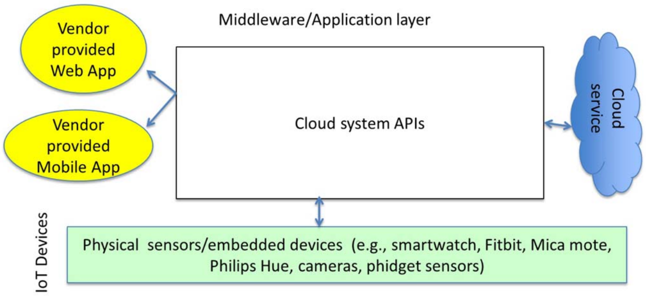

Vendor provided Web App
Vendor provided Mobile App
Middleware/Application layer
Cloud system APIs
Cloud service
IoT Devices
Physical sensors/embedded devices (e.g., smartwatch, Fitbit, Mica mote, Philips Hue, cameras, phidget sensors)

Fig. 3. Cloud-based IoT middleware.

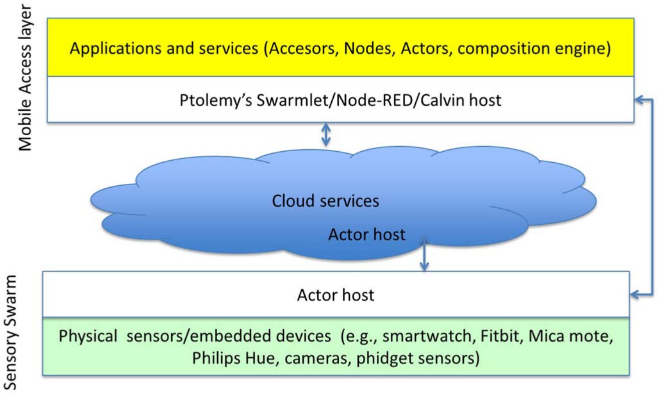

Mobile Access layer
Applications and services (Accesors, Nodes, Actors, composition engine)
Ptolemy's Swarmlet/Node-RED/Calvin host
Cloud services
Actor host
Sensory Swarm
Actor host
Physical sensors/embedded devices (e.g., smartwatch, Fitbit, Mica mote, Philips Hue, cameras, phidget sensors)

Fig. 4. Actor-based IoT middleware.

The service-based architecture is a high-performing heavyweight middleware generally deployed on multiple nodes running in the cloud or on powerful gateways between IoT devices and the applications. It is not designed to be deployed in resource-constrained IoT devices (e.g., smart phones) and does not support device-to-device communication.

A cloud-based IoT middleware architecture is shown in Fig. 3. The functional components (white box in the diagram) of the middleware are limited to what is available on the cloud and it varies widely among cloud-based platforms. Typically, those functionalities are exposed as a set of APIs. The provided functionalities could be as simple as a very high performance storage system or a very powerful computation engine with predefined monitoring and analysis tools. The services of IoT devices available in the cloud can only be accessed or controlled via either vendor's provided application or cloud supported RESTful APIs.

An actor-based IoT middleware architecture is first presented in TerraSwam [19], a joint research project between universities, government and private companies in USA. A three-concentric circles visual is used to depict the architecture of the actor-based IoT middleware. The outermost circle is the sensory swarm (sensors and actuators), the middle circle is the mobile access (gateways such as smartphone, Raspberry Pi, Swarmbox, and laptop) and the inner most circle is the cloud. To facilitate the comparison with other types, we map the three circles into a three layered architectural view. Fig. 4 depicts this architecture. As shown in the figure, the middleware (also named as actor host) is designed to be light-weight that can be embedded in all the layers (sensory layer, mobile access layer, and the cloud). The basic middleware computation units are thus distributed in the network (the white box). For example, an actor-based middleware deployed on a smartwatch might not include a storage service. However, an actor that provides a storage service can be downloaded from the cloud repository when needed.

The key differences in the three architectures are the openness of the architecture in supporting a new IoT device type, the type of middleware services, or computational units they support, and where the IoT middleware can be embedded or deployed. The service-based IoT middleware is deployed in servers or in cloud. This middleware provides users with simple tools such as Web applications to view the raw data that the IoT devices are collecting, but usually provides limited functionalities to users when it comes to composition or integration with other applications or in interpreting the data. The service-based IoT middleware can be set up for restricted access to protect private and sensitive data. The computational units in the service-based architecture are not designed to be extendable or configurable by users.

The actor-based style of architecture provides the best latency and scalability for large-scale connected IoT devices because the middleware can be deployed in all layers and IoT devices can perform computation where it is most beneficial. Users can extend the computational units of the actor-based IoT middleware by developing a pluggable actor or download that from a central repository. Both service and actor-based IoT middleware architectures do not dictate a particular standard such as RESTful API or BLE for interoperability among IoT devices. They both embrace the heterogeneity of IoT devices by supporting a particular programming model or device abstraction. In contrast, in the cloud-based architecture, interoperability is achieved by adopting specific standards. Cloud-based style of middleware can stop working completely when the cloud provider ends the service. Google Nest [20] is a prime example of that.

While all three architectures support security and privacy to some extent, cloud-based architecture requires users to trust the cloud provider to uphold the privacy and integrity of their data. Users are not given alternatives other than those prescribed by the cloud. In service and actor-based architectures, users have a choice of how and where data can be stored. In both service and cloud-based architectures, there is a weak security link between physical devices and the middleware; because the middleware cannot be embedded within the physical device, the data transmitted between physical devices, and the middleware can be compromised.

IoT applications typically operate in a dynamic and uncertain environment. For example, IoT devices can run out of battery power and stop working, and the connectivity between the devices and gateways might be lost at any time. The middleware must provide a *service discovery* component so that new services can be made available opportunistically and failed services can be replaced dynamically to guarantee a certain quality of service. For example, the physical devices can connect to a different gateway of similar quality if the

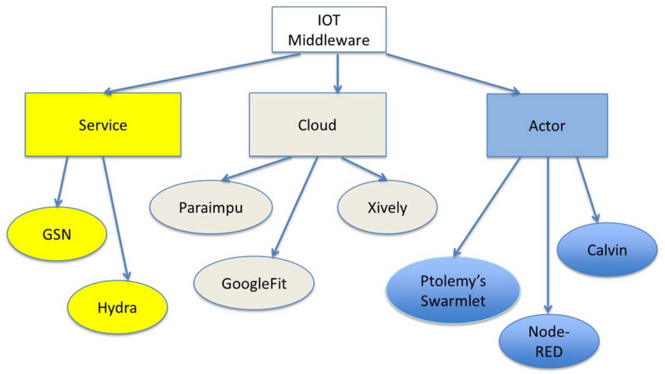


Fig. 5. Overview of various IoT middleware.

current gateway is about to lose connection. Only service-based middleware currently supports a limited form of service discovery.

IV. Existing IoT Middleware Systems

Here, we provide a more in depth analysis of the capabilities of all the three types of IoT middleware in terms of: 1) the abstraction they provide for connecting and accessing the physical devices; 2) the services that they provide for flexible composition of IoT devices and services; 3) the support for service discovery; 4) the handling of privacy and security of the collected data; and 5) the support they provide for creating and provisioning BAC-like IoT applications. Fig. 5 gives an overview of the various IoT middleware that we studied. Many IoT systems, frameworks, or middleware are constantly being developed. Exhaustive coverage of all of them is impossible. We aim to survey at least two IoT middleware systems in each architecture style, and restrict this paper to IoT middleware systems that can support data collection and analysis as in BAC-like IoT application. We target systems that are not just a framework or a standardization effort. We also avoid closed and subscription-based IoT middleware products. A comprehensive list of IoT related systems can be found in [21].

A. Service-Based IoT Middleware

1) Hydra: The Hydra system [22], [23] is a four-year project funded by the European Union to develop a service-oriented middleware for networked embedded systems. The project has been renamed to LinkSmart since 2014. Web service is provided as the main abstraction for incorporating heterogeneous physical devices into applications and for controlling any type of physical devices irrespective of their network technologies such as Bluetooth, RF, ZigBee, RFID, WiFi, etc. Hydra-enabled devices and services are secure and trustworthy through distributed security and a social trust computation unit provided by the middleware.

A major novelty in Hydra is that it provides support for using devices as services by embedding services in devices. Furthermore, the capabilities of IoT devices can be semantically described using ontologies in OWL and semantic annotation for Web service description language (SAWSDL) so that they can be discovered. The framework is intended for three specific application domains: 1) home automation; 2) healthcare; and 3) agriculture. Hydra supports service discovery and self-configuration of services/devices via context (i.e., location and time). It recruits new devices or resources dynamically via peer-to-peer network technologies. Hydra provides an SDK augmented with semantic model driven architecture to create applications in supported domains.

The SDK tool is too low level for use as a composition engine for end users. Hydra advocates the use of a homogeneous standard (Web services) for addressing the heterogeneity of IoT devices. While it is possible to create a smartwatch Web service using the supplied SDK tool kit with training, wrapping IoT devices as Web services might limit the types of IoT devices that can be deployed and managed in this platform since a Web service is a heavy weight protocol to run on energy and capability constrained IoT devices. All collected data are transmitted to the Hydra middleware for processing and archiving. No local processing or aggregation of collected data is possible on IoT devices. This is impractical for some BAC-like applications where detection of critical events (e.g., a fall of an elderly person) from the collected data must be analyzed in real-time). A Hydra IoT application must be handcrafted by a programmer and it is not a platform that can empower the consumer to quickly search, create, and deploy a BAC-like data collection and analysis application. Hydra is thus more suitable for enterprise-level IoT applications that form long-term and tight coupling with a static set of IoT devices that are already supported by the platform.

2) Global Sensor Networks: GSN [7], [24] is a service-based IoT middleware (see Fig. 2) that aims to provide a uniform platform for flexible integration, sharing, and deployment of heterogeneous IoT devices. The central concept is the virtual sensor abstraction, which enables users/developers to declaratively specify XML-based deployment descriptors to deploy a sensor. This is similar to the concept of deployment descriptors used in the deployment of enterprise beans in J2EE server [25]. The architecture of GSN follows the same container architecture as in J2EE where each container can host multiple virtual sensors and the container provides functionalities for lifecycle management of the sensors which includes persistency, security, notification, resource pooling, and event processing. The input to the virtual sensor is one or more data streams which are processed according to the XML specification. These include the sampling rate of the data, the type, and location of the data stream, the persistency of the data, the output structure of the data, and the SQL processing logic for the data stream. Each input stream is associated with a wrapper. The wrapper program specifies the following:

1) which network protocol to be used to connect, interact, and communicate with the physical sensor when first initialized;
2) what to do in order to read data from the sensor;
3) what to do with the data when it is received from the sensor.

GSN provides an SQL-based database that stores all the raw sensor data if the permanent storage attribute of the virtual sensor is specified as “true” in the XML specification.

```
<virtual-sensor name="PhidgetInterfaceKit" priority="10">
  <processing-class>
    <class-name>gsn.vsensor.BridgeVirtualSensor</class-name>
   <output-structure>
     <field name="sound" type="double" />
     <field name="light" type="double" />
    </output-structure>
  </processing-class>
  <description>This phidget sensor holds up to eight other sensors that can hold various data
  </description>
  <life-cycle pool-size="10" />
  <streams>
    <stream name="input1">
      <source alias="source1" sampling-rate="1" storage-size="1">
        <address wrapper="phidgetinterfacekit">
         <predicate key="wireless">true</predicate>
         <predicate key="wired_ip">169.254.247.137</predicate>
         <predicate key="wireless_ip">169.254.162.71</predicate>
         <predicate key="sound">1</predicate>
         <predicate key="sound_id">1B98F4E7E8DD4966926F58C669EEB3FF</predicate>
         <predicate key="sound_device">Microwave</predicate>
         <predicate key="sound_off_ceiling">20</predicate>
         <predicate key="sound_off_floor">-1</predicate>
         <predicate key="light">3</predicate>
         <predicate key="light_id">6F1A90A2740E43A0A4D9C86622D39815</predicate>
         <predicate key="light_device">Microwave</predicate>
         <predicate key="light_off_ceiling">40</predicate>
         <predicate key="light_off_floor">-1</predicate>
         <predicate key="light_on_floor">110</predicate>
         <predicate key="light_on_ceiling">-1</predicate>
       </address>
     <query>SELECT sound, light, timed FROM wrapper</query>
    </source>
   <query>SELECT sound, light, timed FROM source1</query>
    </stream>
  </streams>
</virtual-sensor>
```

Fig. 6. Phidgets sensor XML deployment file in GSN.

In addition, each virtual sensor contains a key-value pair which can be registered and discovered in GSN.

The ability to add a platform specific wrapper enables the system to integrate with sensors of heterogeneous types. To add a new type of sensor to the platform, the user has to know how to write an XML descriptor for the physical sensor and provide an implementation of the wrapper in Java if it is not already available. In the following paragraph, we illustrate the creation of Phidgets sensors in GSN to demonstrate the capability provided by its device abstraction. We choose to show Phidgets sensor implementations for the rest of this paper because we have prototype implementations of Phidgets sensors in all the three types of middleware that we observed. The light and sound sensor data we collected in Phidgets are comparable in characteristics to sensor data that can be collected from a smartwatch.

Adding a Phidgets sensor (IoT device) as a new virtual sensor in GSN requires the creation of a deployment file as shown in Fig. 6 and creating a wrapper class that can run as a thread for consuming the stream data according to the properties specified in the XML deployment file. The `virtual-sensor-name` tag in the deployment file specifies the storage medium for the collected data. The `processing-class` tag specifies the Java class of this virtual sensor, in this case, the `BridgeVirtualSensor`. The `output-structure` tag specifies the structure of the data to be collected. In this case, it is the sound and the light and they are both of *double* type. The `stream` tag specifies how the real-time interaction between the physical device and the application must be supported. For example, the sampling rate, the processing logic on the collected data are specified via attribute `sampling-rate` and the `query` tag. Fig. 7 is the partial implementation of the wrapper class which extends the AbstractWrapper class provided by GSN and gets input as specified by this XML descriptor.

While GSN provides scalable servers for collection and storage of sensor data, it does not provide tools to compose or interpret the data other than display it on a supplied Web application. It also does not support composition of multivendor devices via the XML descriptor. However, the extended GSN [26], which is part of the OpenIoT project, does provide a limited composition capability. In GSN, a domain specific application must be created by a programmer when data need to be collected and integrated from various IoT devices. External applications can access virtual sensors hosted on GSN via RESTful or Web service APIs. A limited form of service discovery, based on dictionary look up, is supported. User data is protected by a login account. Similar to Hydra, all collected

```
public class PhidgetInterfaceKitWrapper extends AbstractWrapper {

   public initialize() {
      connect to the physical sensors;
      read sensor values;
      detect new sensor values;
   }

   public void run() {with certain sampling rate}
   public DataField[] getOutputFormat()
   public String getWrapperName()
   public void dispose()
}
```

Fig. 7. Phidgets sensor wrapper class in GSN.

data are transmitted to the middleware for processing and archiving. GSN is not designed for embedding in power and computation constrained IoT gateways such as smartphones or Rasperry Pi, thus no local processing or aggregation of data is performed. The declarative specification of sensor capabilities via XML descriptor file is a step in the right direction to rapid creation of BAC-like applications by automatic generation of the wrapper class from the descriptor file.

B. Cloud-Based IoT Middleware

1) Google Fit: Google Fit [9] is an open IoT ecosystem. It is a cloud-based IoT middleware that lets users control their fitness data and build fitness apps from one central location. It has a similar goal to Apple's HealthKit. The system contains a fitness store, which is a cloud storage service (similar to Firebase, a JSON-based document server) that stores data from a variety of devices and apps. A sensor framework consists of a set APIs for connecting third-party IoT devices to its store. For example, it provides APIs for subscribing to a particular fitness data type or a particular fitness source (e.g., Fitbit or Smartwatch), APIs for querying of historical data or persistent recording of the sensor data from a particular source (e.g., a smartwatch). In addition, there is a permission and user control module that handles the privacy and security of data by prompting for user consent before Google Fit's apps can read or store collected data. Google Fit is an IoT middleware designed for ease of composing a preconceived type of application, in this case, self-tracking data from wearable fitness devices.

Google Fit provides native support for IoT devices that communicate in BLE. Adding a new fitness sensor type that does not communicate in BLE requires a developer to provide an implementation of `FitnessSensorService` class as well as the supported data type if it is not available. In Fig. 8, we provide a sample implementation of a Google Fit software sensor for a Phidgets sensor to demonstrate the complexity of creating a BAC-like application for IoT devices that are not yet supported by Google Fit.

To collect and store sensor data from a Phidgets sensor which communicates via a WiFi network, there is a need to implement a sensing service as an Android app to communicate with Google Fit's cloud storage server. Fig. 8 shows the implementation of the `PhidgetsSensorService` which extends `FitnessSensorService`. The programming for that service involves understanding Java's Event programming as well as mastering the various callback methods in `FitnessSensorService` class which are non trivial. However, for IoT devices (e.g., Fitbit and Garmin) that are already supported by Google Fit, it is just a matter of downloading and installing a Google Fit app on Android compatible gateways such as smartphone or tablet, creating an account, setting up personal details, and permission. Note that by choosing to use Google Fit, the user is tied to storing his/her sensor data in Google Fit's cloud storage, in the format provided by Google Fit and in the size limit dictated by Google Fit. It is not possible to preprocess the collected data before storing them such as normalizing the data as in the BAC prediction application.

Google Fit thus has a narrow application scope and does not provide a framework for general IoT application's data collection, composition, and analysis. Service discovery is limited to a scan for nearby BLE devices. There are significant problems with privacy, security, and unpredictable latency when using cloud-based architecture. Moreover, users must entrust Google Fit to manage their private data.

2) Xively: Xively [27], along with LogMeIn, is a public cloud-based IoT middleware's platform as a service. Xively's overall mission is to help developers and companies to turn physical sensors into software sensors and connect them to Xively's IoT cloud platform quickly and simply. The IoT middleware provides a Web-based application for quickly connecting IoT devices to its cloud and collecting data from the devices. Once data is in the cloud, Xively allows users to pull data from them easily anytime and anywhere using their tools. The main use for Xively is for connecting IoT devices of the users' choice and store the collected data on their cloud in a scalable way. Xively also provides a persistency service that includes a time-series database for fast data storage and retrieval.

The company has many tools and resources that developers can use to connect their sensors and collect data from those sensors. For example, it provides adaptors for integration of collected data with commercial enterprise softwares such as customer relationship management, enterprise resource planning, and business intelligence. A directory service is provided for finding appropriate services also known as service discovery.

```
import com.google.android.gms.fitness.*;
import com.phidgets.InterfaceKitPhidget;

public PhidgetsSensorService extends FitnessSensorService {

  private Phidget pClient = null;
  private Map<String, FitnessSensorServiceRequest> mRegisteredSensors = new HashMap<>();

   // Listens for and publishes the light and sound sensor data.
   private final SensorChangeListener mListener = new SensorChangeListener() {
   @Override
    public void onPhidgetChanged(PhidgetInterfaceKit e) {
         FitnessSensorServiceRequest associatedRequest = mRegisteredSensors.get("Phidgets");
         if (associatedRequest != null) {
            List<DataPoint> data = new ArrayList<>();
            data.add(DataPoint.create(associatedRequest.getDataSource())
                        .setFloatValues((float) e.getSensorValue(1);
                        .setFloatValues((float) e.getSensorValue(3);
                        .setTimestamp(System.currentTimeMillis(), TimeUnit.MILLISECONDS));

            try {
               associatedRequest.getDispatcher().publish(data);
            } catch (PhidgetException re) {}
         }};
   @Override
   //find the right data source
   protected List<DataSource> onFindDataSources (List<DataType> list) {
     List<DataSource>matchingSources = new ArrayList<>();
     matchingSources.add (new DataSource.Builder()
                  .setDevice(new Device("PhidgetInterfaceKit", "Sensors", "--",
                      Device.TYPE_Phidgets))
                  .setName("Phidgets Sensor")
                  .setType(DataSource.TYPE_RAW)
                  .setDataType(DataType.TYPE_PHIDGET_LIGHT_SOUND)
                  .build()
      );
      return matchingSources;
   }
  @Override
  //Register sensor with GoogelFit to deliver data
  protected boolean onRegister(FitnessSensorServiceRequest request) {
      if (pClient != null || attemptConnectingPhidgetClient()) {
         try {
            if (mRegisteredSensors.get("Phidgets") == null) {
               pClient.getSource().addAttachListener(mListener);
               mRegisteredSensors.put("Phidgets", request);
            }
         } catch (PhidgetException e) { return false;}
         return true; }
      return false;
   }
  @Override
  //configure the sensor to stop delivering data
   protected boolean onUnregister (DataSource dataSource) {}
   }
```

Fig. 8. Google Fit sensor service.

Even though adding sensors or IoT devices is advertised to be simple in Xively, the provided APIs require nontrivial programming skill especially for IoT devices that are not already supported. It is impossible to compose an IoT application in Xively that correlates data from multiple IoT sources. For example, no composition engine is provided such that it is possible to correlate data from a sensor hosted in GSN with a sensor hosted in Google Fit without extensive programming. Fundamentally, Xively is providing similar capabilities as Google Fit albeit it covers a more diverse set of devices and includes integration with enterprise-level services. The system is targeted for business users. As a cloud-based architecture, Xively has the same security, privacy, and latency issues as experienced by Google Fit.

3) *Paraimpu:* Paraimpu is a social aware IoT middleware [28], [29] that allows consumers to add, use, share, and interconnect their RESTful IoT services whether physical or virtual. Things are mapped to either the abstract concept

of sensors or actuators in Paraimpu. The former characterizes anything capable of producing data of a related type (text, numeric, JSON, XML, etc.) and the latter characterizes any thing that is able to perform actions by consuming data produced by the sensors. Paraimpu also provides the connection abstraction between things. This allows users to compose simple IoT applications via Javascript. All sensors and actuators in Paraimpu are represented as RESTful resources and JSON is used for internal interchange of data between devices. Paraimpu is implemented using a scalable architecture leveraging a nonblocking Tornado Web sever [30], an NGINX [31] load balancer, and a MongoDB [32] which provides persistency, replication, and fail-over data management support. In other words, Paraimpu aims to provide a scalable cloud infrastructure.

The key advantage of Paraimpu over other IoT middleware is the ability for consumers to reuse and share IoT services with others in their social networks. Paraimpu provides a limited set of configurable sensors, actuators, and connections that can be reused across applications via filtering and mapping between inputs and outputs among sensors and actuators. A simple JavaScript-based rule engine is provided for specifying composition logic between one sensor and one actuator. If the developer wants to connect two sensors to the same actuator, he/she will have to make two separate connections and make a replica of the actuator. Paraimpu does not support service discovery. Security is handled via supporting https protocol and authentication via an access token. Privacy is handled by giving users the control of how they want to share their data (private versus public). As in Google Fit and Xively, users have no control over the total ownership of their data. Paraimpu does not provide device to device communication and thus entails the usual latency problem of a cloud-based architecture. Tutorial materials available on Paraimpu seem to indicate that Paraimpu is good for IoT applications that only involve one sensor and one actuator.

C. Actor-Based IoT Middleware

1) Calvin: Calvin [33] is an open-source IoT middleware from Ericsson that aims to provide a unified programming model which is light-weight and portable for capability and energy constrained IoT devices. It is a hybrid framework combining concepts from the actor-oriented model and flow-based computing for composing and managing IoT applications. The main abstraction for building IoT applications in Calvin is an actor which is a reusable software component that can represent a device, a computation, or a service. An actor's interface is defined by its input and output ports. An actor reacts to inputs by producing outputs, rather than reacting to method calls by returning values as in the traditional object-oriented model. This paradigm of actor model follows an asynchronous atomic callbacks pattern where short atomic actions are interleaved with atomic invocation of response handlers for high performing real-time interaction. Calvin's actor model also hides the low-level communication protocols of things since actors connect and communicate through ports no matter how the physical connectivity is done. Calvin provides its own version of scripting language for ease of programming of an actor. It advocates a prescriptive application development process called describe, connect, deploy, and manage to improve the process of developing an IoT application. Calvin is a light-weight IoT middleware that can run on devices on the edge to minimize the latency and also utilize the full computing power available on the cloud when the need arises.

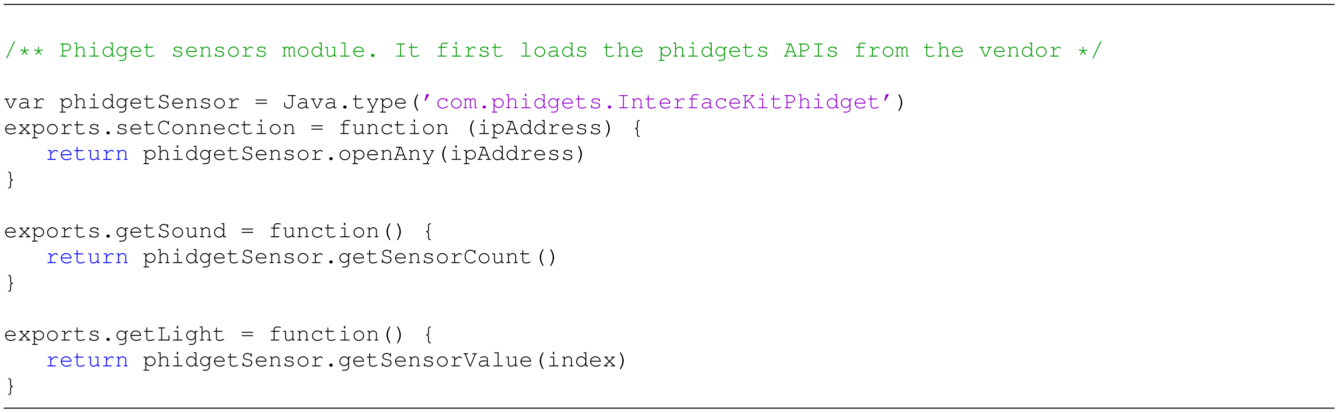
One key advantage of Calvin's actor is its ability to migrate from one runtime environment to another to provide a robust distributed IoT computation platform. The platform also comes with a predefined set of actors that perform common unique tasks. This includes actors for popular communication protocols and parallel processing. Calvin's developer may extend the capability of this middleware by creating a new actor using CalvinScript and add the new actor to the library. Actors in Calvin can be composed using CalvinScript.

Calvin does not yet support a graphical user interface (GUI)-based composition of IoT applications. In the context of BAC-like IoT applications, a smartwatch actor, a analysis and visualization actors must be developed to compose such an application. The real-time interaction between physical sensors and the application is managed by Calvin's runtime system which is light-weight and can be deployed in any layer of the IoT middleware. Calvin advocates a simple programming model for actors and unless this programming model is adopted by the IoT developers' community and IoT vendors, creating BAC-like IoT application will not be trivial. Both security and service discovery are not addressed in Calvin.

2) Node-RED: Node-RED [34] is an open-source IoT middleware platform from IBM. It is based on `node.js`, a server-side Javascript platform that uses an event-driven, nonblocking I/O module in a distributed computing environment. Similar to Calvin, it is an IoT middleware that can be run at the edge of the network because of its light footprint. The main abstraction is *Node* which is a visual representation of a block of Javascript codes designed to perform a specific function on an IoT device (e.g., reading a particular value). In other words, each node can be viewed as an actor.

Node-RED's key advantage is a visual tool that simplifies the job of composing IoT devices especially if the node for the IoT device is already developed and published by others. The visual tool in Node-RED allows users to drag-and-drop blocks that represent components of a larger system and wire them up to form an IoT application. Node-RED thus supports composition of IoT applications. The composition engine acts as a glue to IoT devices that can be abstracted as nodes.

For a device or service to work with Node-RED, the APIs for communicating with the device must be available as `node.js` library or a module accessible by Node-RED. A limited form of security is provided via password authentication. The Node-RED team envisions that modules or `node.js` libraries for heterogeneous IoT devices can be crowdsourced via creating a social network of Node-RED developers. Node-RED does not provide service discovery. It is built on `node.js`, which is a new platform with limited libraries or modules. Node-RED is good for rapid prototyping of IoT applications whose functionalities can be encoded



```
/** Phidget sensors module. It first loads the phidgets APIs from the vendor */

var phidgetSensor = Java.type('com.phidgets.InterfaceKitPhidget')
exports.setConnection = function (ipAddress) {
   return phidgetSensor.openAny(ipAddress)
}

exports.getSound = function() {
   return phidgetSensor.getSensorCount()
}

exports.getLight = function() {
   return phidgetSensor.getSensorValue(index)
}
```

Fig. 9. Phidgets sensor module in Ptolemy accessor host.

in simple Javascript and executed with Node-RED's in-built event-driven computation model for real-time interaction with physical devices.

3) Ptolemy Accessor Host: Ptolemy [35] is an actor-oriented framework for the modeling, simulating, and designing of concurrent, real-time, embedded devices. It is an open-source system developed by Prof. E. Lee at the University of California, Berkeley, since 1996. The well-known Kepler scientific workflow system [36] is built on top of the Ptolemy system. The central modeling concept in Ptolemy is actors which are software components that execute concurrently and communicate through messages sent via interconnected ports. A novel feature of Ptolemy is that the overall execution and actor interaction semantics is not defined by the actors, but is factored out into a separate component called *director*. Director plays the same role as the orchestration engine in a workflow. Ptolemy accessor host extends the actor-oriented framework with an accessor abstraction. An accessor encapsulates an IoT device and can expose an actor interface [8]. Each accessor has an interface and an implementation. The interface serves as the local proxy for the remote IoT device and is specified in Javascript which is light weight and portable. The implementation (similar to a wrapper) encapsulates the APIs of the physical devices which can be implemented in any language. Accessor leverages the disciplined model of computation in Ptolemy and embraces the heterogeneity in IoT devices rather than proposing a single standard to homogenize them. Ptolemy accessor host also provides a GUI for the ease of composing IoT applications. The GUI contains drag-and-drop features (an easy way to connect actors and accessors to each other) as well as a comprehensive framework for managing the composed IoT applications. This includes various kinds of displays for the execution status and the ability to pause and resume the execution at anytime. Accessor is designed to run on the edge devices, the gateways, as well as on the cloud. This ensures that the IoT application is executed where it is most beneficial. Accessors can be downloaded and shared over the Web as in Paraimpu.

To demonstrate the light-weight programming model used in actor-based architecture, we illustrate an implementation of an accessor to collect data from a Phidgets sensor in Ptolemy accessor host. Note that the communication module for Phidgets sensor's hardware must be available on the Ptolemy accessor host. The communication module can be implemented in any programming language, however, to make the module executable on other accessor hosts beyond Ptolemy (a host that has a JavaScript engine), the module needs to conform to the CommonJSModule standard. Fig. 9 is a partial implementation of the `phidgetsSensor.js` module, which encapsulates the phidgets' communication APIs. Fig. 10 shows the implementation of a Phidgets accessor that makes use of the functionalities provided by the module to collect sound and light sensor data. When compared with the wrapper codes for Phidgets in GSN or the phidgets sensor codes in Google Fit, Ptolemy's accessor script and its module are simpler because they do not require knowledge of threads or concurrent programming. An accessor script is designed to be easily written with the description automatically generated for other applications to consume. One of the built-in computation models of the Ptolemy, i.e., the discrete event director handles the real-time semantics of accessors execution and coordination with the physical devices and other actors.

The key advantage of Ptolemy accessor host is the ability to compose multivendor IoT devices grounded on rigorous *multiple* computation models with time semantics. For example, it guarantees correct execution of a real-time distributed IoT application. Other actor-oriented platforms (Node-RED and Calvin) only support a single model of computation typically based on data/event flow without rigorous time semantics. In addition, in Ptolemy accessor host, developers can leverage extensive existing actor libraries (computation, storage, or visualization actors) for composing IoT applications. Ptolemy accessor host is good for both the dynamic reactive IoT applications (e.g., control of autonomous vehicles) as well as the ambient data collection and analytics BAC-like IoT applications because of the multiple computation models inherited from Ptolemy.

Ptolemy accessor host does not yet support service discovery. As the number of actors and accessors grow, the ability to discover compatible actors or accessors will be valuable for composing adaptive distributed IoT applications. Various access controls can be made available via specialized actors

```
/** Load the specified module by name. */
var phidgets= require('phidgetsSensor')

/** Define inputs and outputs of the accessor */
exports.setup = function() {
   input('trigger');
   parameter('sampling rate', {'type':'int', 'value': 1});
   parameter('ipAddress', {'type':'string', 'value': ' '});
   output('sound', {'type':'number'});
   output('light', {'type':'number'});
}

 /** Whenever a new input arrives at the input port, in this case the 'trigger' input port,
    the new light (id 3) and sound (id 1) data will be send to the respective output ports */
exports.initialize = function() {
   phidgets.setConnecion(get('ipAddress'))
   addInputHandler('trigger', function() {
         var lightValue = phidgets.getSensorValue(3);
         var soundValue = phidgets.getSensorValue(1);
         send('light', lightValue);
         send('sound', soundValue);
      });
}
```

Fig. 10. Ptolemy phidgets accessor.

in Ptolemy accessor host. However, when an IoT application involves multivendor devices, being able to guarantee privacy of data from each device and yet to share the data effectively for actionable knowledge is a significant challenge.

D. Summary of IoT Middleware Systems

Existing IoT middleware systems, which focus on supporting the efficient and secure processing of streaming data from a large number of homogeneous sensors, tend to be service-based while those that focus on consumer usability tend to be cloud-based. We summarize and discuss the main characteristics of existing IoT middleware systems in this section and identify the key challenges and research gaps related to providing a light-weight, open, and secure IoT middleware in next section.

The horizontal column heading in Table I lists the key functionalities for IoT middleware that will empower consumers to create BAC-like IoT applications. Each row of the table enumerates the functionalities provided by the IoT middleware system we surveyed for composing BAC-like IoT applications. The device abstraction column refers to the construct used to encapsulate the heterogeneity of physical devices. The network connectivity column refers to the types of communication protocols supported (e.g., HTTP, UDP, and BLE). Composition column refers to the ability to connect and mashup cross vendors IoT devices regardless of its connectivity protocol or message format. The monitoring and visualization column refers to the ability to check the state of the devices and visualize the collected data at any time and anywhere without having to write a complex third-party application. The service discovery column refers to the ability to dynamically recruit the best device or service with certain quality level for composition. The security and privacy column refers to how personal data are stored and managed. Consumers must be confident that sensitive data from their devices do not propagate beyond authorized entities or tempered with before they use the application. The persistency column refers to how the IoT middleware provides and handles the storage of the varied types and amount of data.

Table I shows that device abstraction is provided by all surveyed systems. However, the type and granularity varies quite a bit. In cloud-based middleware, vendor supplied low level APIs are used for adding IoT devices not natively supported. In service and actor-based middleware, high-level programming models such as accessors, node, or virtual sensors are provided. The ease of deploying a new IoT device on the middleware is directly related to how device abstraction is supported. We have demonstrated in Fig. 8 that adding a Phidgets sensor by subclassing the `FitnessSensorService` class in Google Fit is non trivial. The network connectivity is well supported across all three type of architectures. The actor-based middleware supports the most varied and biggest selection of network protocols. All systems provide persistency service, the difference is in the types of data that can be stored and the scalability of the service. Google Fit currently only supports data related to fitness. Multimedia data are not supported.

As cloud-based middleware is targeted to a large number of IoT devices and users, a high performance persistency service is a necessity. Xively and Google Fit do not review the technical details of their persistency service and thus we list them as "cloud storage" in the table. This can be interpreted as having a combination of high performing Web servers and storage systems. For actor-based middleware such as Ptolemy accessor host, persistency service defaults to the local file system which might not be scalable. The composition and service discovery are the least supported among the surveyed systems. Paraimpu provides limited composition capability between a sensor and an actuator. Only Ptolemy accessor host currently

TABLE I
IoT MIDDLEWARE VERSUS FUNCTIONALITIES

Middleware	**Functionalities**							
	Device abstraction	**Network connectivity**	**Composition**	**Monitoring and visualization**	**Service discovery**	**Security and privacy**	**Persistency**	**Stream processing**
Hydra	Web service	ZigBee, Bluetooth, RFID, Wi-Fi, HTTP, SOAP	SDK took kit with Semantic Model Driven Architecture	Vendor GUI application	OWL and SAWSDL for device description	Distributed security and social trust	Unknown	Event manager
GSN	Virtual sensor/XML deployment descriptor	Push/pull, SOAP, HTTP	Specific application created by programmer	Vendor Web application	Key-value store registry for devices	Login account, electronic signature	SQL database	SQL query on data stream
Google Fit	Fit Programmable API or RESTful resource	Bluetooth, Wi-Fi. HTTP	Fitness based devices and data	Vendor Web application	Not supported	Permission and user control module	Cloud storage	Sensors API
Xively	RESTful resource	Bluetooth, Wi-Fi, Socket, MQTT, HTTP	Not supported	Vendor Web application	Not supported	User control on how data is shared	Cloud storage	RDF stream processing
Paraimpu	Sensor, Actuator, Connection	Pull/push, HTTP	Compose one sensor with one actuator using Connection	Paraimpu workspace	New sensor can announce its presence	Access token needed for access, user data shared according to the stated privacy law	Mongo DB	JSON query
Calvin	Actor	Wi-Fi, BlueTooth, i2c	CalvinScript	Vendor Web application	Provide search via Calvin Control APIs	Not supported	Distributed Hash Table (DHT)	Data flow processing
Node-RED	Node	MQTT	Drag and drop visual tool	Browser-based flow editing tool	Key word based search provided for node	User permission with password securely hashed using the bcrypt algorithm	Local file system	Node streams
Ptolemy Accessor Host	Accessor	MQTT, Bluetooth, Wi-Fi. HTTP, UDP, CoAP, Websockets	Drag and drop visual tool based on Virgil	Java-based GUI application	Import and export facility	Key management and Taint analysis	Local file system	Discrete event director

provides a comprehensive composition engine. The stream processing engine in the middleware provides an abstraction for the real-time interaction between the application and the physical devices. Composition of IoT applications is made much simpler when the middleware provides a scalable stream processing engine. Among the systems, Ptolemy accessor host provides the most advanced stream processing engine via various directors. None of the systems currently provides security solutions targeted to the unique characteristics of IoT devices.

V. KEY CHALLENGES AND ISSUES

While IoT middleware holds promise for developing innovative IoT applications and services that we might not even be able to imagine right now, it also poses a number of daunting challenges [1], [5], [37]. The first challenge is in developing an IoT middleware that must be available in the cloud as well as on the edge (IoT devices and gateways) for supporting all types of IoT applications, for better privacy control and latency. This requires the system to be portable and lightweight. Among the IoT middleware we studied, only Calvin, Node-RED, and Ptolemy accessor host are designed to be portable and light-weight. There is a tradeoff between having powerful services such as semantic-based discovery, fraud-resilient security enforcement, and stream processing versus the ability to deploy an instance of the IoT middleware in constrained devices.

The second challenge is to empower consumers to create IoT applications targeted to their context. In the cloud-based IoT middleware, no composition engine is provided for consumers. This limits consumers to the preprogrammed IoT applications and prevents consumers from creating their own innovative applications. For the service-based architecture, an SDK tool is provided for crafting an IoT application. This requires low level programming knowledge and does not empower consumers to create their IoT applications targeted to their needs. Currently, only the actor-based IoT middleware and the OpenIoT project provide visual composition tools. The visual tools provided by Node-RED and Ptolemy accessor host are early composition tools in this research direction. For example in Ptolemy accessor host, the accessors, which are the fundamental elements of the composition tool, must be designed and implemented according to a specific programming model. This requires end users to master JavaScript or other scripting languages in order to create an IoT application if the desired accessors have not already been provided. Moreover, an accessor in an IoT application is "designed to fit" a particular usage. Each new usage requires the development of a new accessor. A higher-order accessor or a context-aware accessor needs to be developed for flexible composition of IoT applications. For example, by gathering some contextual information from consumers (location, time, URL, and communication protocol), the desired accessor can be configured automatically as a subclass of an existing accessor for the desired IoT application. In addition, currently, no composition tool supports transactional properties. It assumes IoT applications will run from the beginning to the end successfully. There is no provision of rollback or restart from a certain point in a composed IoT application when there is a failure.

The third challenge is to provide semantic service discovery that goes beyond discovery of IP addresses of the nearby IoT devices. Given that the environment that the IoT application interacts with evolves continuously, new services or devices could come on-line anytime and existing devices might become unavailable. It is essential to be able to discover or query for compatible services at the right time and at the right place both at design time as well as at runtime. For example, in some critical health monitoring IoT applications, failed IoT services must be replaced without causing any disruption.

The fourth challenge is to guarantee the security of IoT applications and also protect the privacy of users. Many applications from a variety of domains, ranging from smart healthcare [38] to digital agriculture [39], are utilizing the IoT infrastructure. Critical decisions are going to be made in these applications by analyzing data collected from the IoT devices. This has raised the issues of security, privacy, and trustworthiness of IoT generated data [40]. These issues are not limited to data alone, but also the underlying networks and devices. Hence, supporting security, privacy, and trust mechanisms within IoT middleware has been recognized as a critical and important issue for the successful deployment of IoT applications, and is deemed as one of the major challenges in both industry and academic communities. Security is generally supported via some kind of authentication and encryption protocols and privacy is addressed by giving the end user the ability to specify different level of access controls without the guarantee of data ownership.

In the rest of this section, we will focus our discussions on the challenges and solutions to provide IoT service discovery and security, which are the two most prominent gaps in the eight systems we surveyed.

A. IoT Service Discovery

We first review the existing service discovery mechanisms and then outline a few strategies for providing service discovery in IoT middleware.

1) Existing Service Discovery Approaches in IoT: A common approach to support adaptability in IoT middleware is to adopt a uniform service description and a mechanism to reason over that description for service discovery in different contexts. For example, at design time, a user should be able to query a registry of service descriptions and with the click of a few buttons start to interact with a device that can guarantee the desired capability and quality. Within the OpenIoT project [18], the W3C Incubator Activity XG group has proposed an semantic sensor network (SSN) ontology for sensor discovery and dynamic integration of sensors. SSN ontology captures both the properties or attributes of the sensors and the observations derived from the sensors. The key concepts in SSN are *Sensor*, *Observation*, *Property*, *FeatureOfInterest*, *SensorOutput*, *MeasurementCapability*, *ObservationValue*, and *Condition*. Using this ontology, a wind sensor for Mount Washington hosted in the Paraimpu IoT middleware, which is capable of measuring wind speed every ten minutes, will have FeatureOfInterest having a value of "wind" and the Property with a value of "speed." SSN enables IoT devices and the observed data to be annotated in a way that can be searched and understood by others.

In XGSN [26], the extended GSN project, both the IoT devices and the observed data are semantically annotated using SSN. The semantic annotation of the sensors consists of the metadata associated with the physical sensor such as its location, the type of observation it produces, the responsible person or organization for the sensor, and the source type. The other semantic annotations are associated with the observed data, which include the time and context when the observation happen, the observed property, the measurement unit and the values. The annotated data is converted to resource description framework (RDF) streams and sent to a cloud-enabled linked sensor middleware (LSM) [41] which is a Linked Data RDF store. LSM can either store the RDF stream data in a triple store or perform continuous queries over the stream data using SPARQL queries to derive aggregated observation in real-time. Using this setup, each GSN virtual sensor has a corresponding sensor instance in the LSM's RDF store. XGSN provides an RESTful service for registration of the annotated virtual sensors. However, the annotated semantic of the virtual sensors must be provided in an associated RDF file, which has to be created first by an expert who knows RDF.

SensorML [42] is an approved open geospatial consortium standard (OGC). SensorML supports semantic description of IoT devices by providing a way to describe a sensor using standardized XML tags such as `PhysicalComponent` for describing metadata associated with the sensor, `Output` and `DataStream` which describe the observed properties of the sensor and `Position` which describes the location of the sensor. The main objective is to provide a semantic annotation of sensors, actuators and processors so that they are interoperable and can be understood by machine and utilized automatically in complex workflows. SensorML is a part of the OGC sensor Web enablement suite of standards.

The Hydra [23] project adopts OWL and SAWSDL to semantically annotate their IoT devices for discovery. OWL is an ontology language for semantic Web. OWL ontologies are usually stored as RDF documents. The SAWSDL allows a Web service description language (WSDL) file to be enriched with semantic description. This assumes that IoT devices can be automatically exposed as WSDL files.

Other considerable efforts made in IoT service discovery include Snoogle [43], Microsearch [44], and MAX [45]. These systems assume that sensor nodes attached to physical IoT devices carry textual descriptions of the devices in terms of keywords. Users then have the opportunity to find those devices by matching a query consisting of a list of keywords. The results from these systems are still experimental and preliminary. Ostermaier *et al.* [46], reported a system, named Dyser, for real-time discovery of things on the Web. Dyser uses statistical models to predict the state of its registered resources when a user submits a query. The predictions are used to establish a ranking that determines the order in which resources are contacted to find out whether their current state matches the query. However, contacting registered resources for every query implies serious performance overhead, particularly given that the number of queries and things could be enormous. It is not trivial to build an efficient index for such a big data set.

2) Open Problems and Suggested Solutions for IoT Service Discovery: The semantic annotation of sensors and sensor data enables sensor data from different sources to be shared and queried in a uniform way. While emerging standards like SSN and linked data are gaining adoption, competing standards like SensorML and SAWSDL cannot be ignored. Although there are open source ontology tools such as Protege[1] that can be used to provide semantic annotation of IoT devices, it is still nontrivial to annotate IoT devices with such descriptions without expert knowledge of RDF. To realize a uniform IoT service description system, a translator or an annotation tool should be provided for setting up a registry of IoT devices and access mechanisms. In the OpenIoT project [47], a visual semantic annotation tool is integrated with XGSN to hide the details of creating RDF file from users. This annotation tool is very specific for XGSN middleware. Further research is needed to develop a generic annotation tool that can be used by any IoT middleware. With an ontological approach, unless the community as a whole agrees on a single standard, it is impossible to develop a generic annotation tool. Given the rapid technological advances in this field, it might be impossible to maintain a single global ontology for describing IoT devices.

There is also the *scalability* issue when using ontological approach for service discovery in IoT. When there are large number of sensors streaming large quantities of data over a long period of time, RDF can be too verbose for capturing the semantics of the data especially for energy and computation constrained IoT devices. Furthermore, SPARQL query engine used for querying the RDF store might not scale with the large number of data streams from IoT devices.

We propose two nonontological approaches for service discovery. The first approach is to create a similarity-based search engine for heterogeneous IoT devices or services. This approach has been used successfully to create similarity-based Web service search engine such as Woogle [48], WSExpress [49], and Titan systems [50]. Web services are self-describing interfaces that can be programmatically accessed and manipulated through the Web. We have created a Web service similarity search engine called ServiceXplorer [51]. A user can upload an existing Web service description file, the system will parse the description and create records of the key functionalities of the Web services that can be searched using the earth movement distance-based similarity search algorithm. This approach assumes that the interface for an IoT device is available in the form of a document and there is some regularity in the structure of the document. For example, documentation created by JSDoc for an accessor in Ptolemy has a regular structure that highlights the functionalities of the encapsulated physical device. An example of the documentation of an accessor for Philips Hue lightbulb in Ptolemy can be found online.[2] A query can be expressed as a combination of keywords or key phrases. This approach is restricted for discovery of meta data of sensors, not the observed data from sensors. For example, a query to discover all temperature sensor services that have readings above 90° in the last 2 h cannot be answered using this approach.

The second nonontological approach toward service discovery is based on performing data analytics on the usage log of the devices or services. Instead of having to label the IoT devices or services using ontology, this approach aims to collect the usage history of the devices and services and use that to infer the capabilities and relationships between devices. By analyzing when, where, who is using the device (e.g., a device used frequently around lunch time, located in the kitchen, and used by someone who has the overlapping scheduled lunch time has a high probability to be a kitchen appliance), it is possible to derive relationships between devices and services. This allows recommendation of devices or services for composition as well as optimizing the time and cost of using an IoT device in a particular situation. For example, when multiple people want to use a particular IoT device which is in use, it makes sense to recommend other similar devices by looking at devices that are the nearest neighbor of this device without having to manually set up the ontological description of all devices in the system. Recommending relevant things by discovering implicit relationships between things via usage logs has been described in a recent research work [52], [53]. A key advantage of this approach is that the discovery of devices and services takes place behind the scene. As devices come on line and get used, the usage log is created and data analytics can be used to classify this device. The key research problem that needs to be addressed in this approach is ensuring that privacy of users can be preserved when mining the usage log of the devices or services.

B. Security and Privacy

In the previous sections, we have discussed the security and privacy issues in the selected IoT middleware systems. In this section, we first define the security, privacy, and trust in the context of the IoT environment. We then review existing mechanisms on supporting security, privacy, and trust, with a specific focus on the existing IoT middleware, and finally discuss some open problems and suggested solutions.

1) Security, Privacy, and Trust Properties: Building resilient, trusted IoT systems requires that key essential properties for security, privacy, and trust are met. What are these properties? There are neither consensus definitions, nor exact boundaries on which properties fall into security or privacy or trust. However, it is clear from the published literature that there has often been a focus on specific properties rather than taking the issue of security, privacy, and trust in IoT systems holistically. For the purpose of this discussion, we capture some key properties discussed in the literature in Fig. 11. It is important to note that they do not represent the comprehensive list of all relevant properties.

[1] [Online]. Available: protege.stanford.edu

[2] [Online]. Available: https://www.terraswarm.org/accessors/doc/jsdoc/accessor-devices_Hue.html

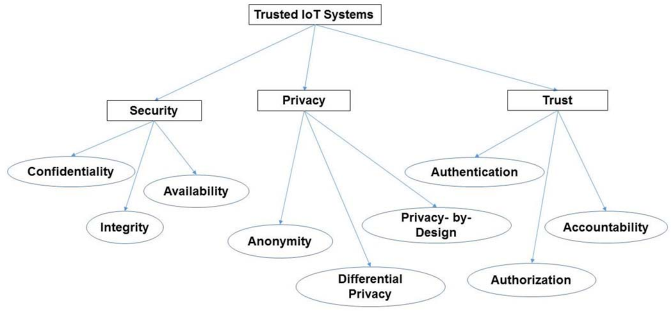


Fig. 11. Key security, privacy, and trust properties in IoT system.

Security is often defined using three properties: *confidentiality*, *integrity*, and *availability* [54]. Confidentiality means that a given message must not be understood by anyone other than the desired recipients. The integrity property ensures that the data produced and consumed in the IoT system are not maliciously altered. The availability property means the system is robust enough to be able to operate in adverse situations. In addition to these three key properties, there are more security properties defined in the literature, such as forward and backward secrecy [55], that are equally important in certain context within IoT systems.

The treatment of privacy in the literature is varied, ranging from being defined as one of the security properties [55] to referring to particular privacy enhancing technologies [56]. For our discussion, we define privacy with regard to entities in an IoT system that can have control over how information is shared and distributed. The entities here could be the end users of the IoT system or IoT devices [57]. We capture the privacy needs through three key popular and emerging techniques: 1) anonymity; 2) differential privacy; and 3) privacy-by-design. We refer readers to [58]–[60] for further information.

Trust is often defined through three properties: 1) *authentication*; 2) *authorization*; and 3) *accountability*. The authentication property ensures that the data is sent by the claimed sender [61], whereas the authorization property guarantees that only authorized entities are able to perform certain operations [62]. The accountability refers to the property that enables auditing of IoT systems for significant events and analyze log to associate the behaviors of entities to such events [63]. Audit trails and logs are important for detecting security violations and recreating security incidents. In addition to the above three properties, there are other properties in the literature that are related to trust management such as freshness, reputation, access control, and nonrepudiation. We refer readers to [55] for further details.

In the following, we review how existing IoT middleware systems are designed to support different security, privacy, and trust properties.

2) Existing Approaches for Supporting Security, Privacy, and Trust: The problem of security and privacy in the IoT systems has started to receive greater attention from researchers recently, resulting in a number of publications in this area [64]–[66]. However, the focus of the majority of these publications is on the sensors, networking, and application layers in the IoT architecture, rather than the middleware. Bandyopadhyay *et al.* [67] have surveyed ten popular IoT middleware including Hydra [68], GSN [69], SOCRADES [70], and SMEPP [71], and reported that only six of them support security and privacy features. Fremantle and Scott [72] are the first who surveyed twenty two IoT middleware systems with the focus on security and privacy. They also found that only ten of the surveyed systems have a security model, but many of those models are not well defined and developed. This clearly demonstrates the lack of security support in the existing IoT middleware systems. When we consider the security, privacy, and trust properties described earlier, only eight of the twenty two support a form of confidentiality, integrity, authentication, federated identity, access control, and attestation mechanism. However, none support privacy-by-design and trust management. In the following, we review the support of security, privacy, and trust in the middleware architecture discussed earlier.

a) Service-based IoT middleware: Hydra [68] (also known as LinkSmart [73]) is a service-based IoT middleware implemented using Web services and utilizes the XML security model [74]. One of the drawbacks of the standard XML security model is that it is costly in both time and memory due to the XML canonicalization needed for digital signature. The data encryption in Hydra/LinkSmart is supported through the use of symmetric keys, as they are more efficient than the public key infrastructure (PKI). However, it suffers from a unique challenge in key management, i.e., in creating and distributing keys. To alleviate this problem, the Hydra/LinkSmart system offers a service, called TrustManager, to support PKI with a central certificate authority (CA). The CA issues a signed certificate to each identifiable device in Hydra/LinkSmart that is used to identify the communicating parties in the system. Hydra middleware offers limited support to the security, privacy, and trust properties described earlier. It does not offer any policy-based access control for IoT data, and does not address the secure storage of data for users. It also fails to offer any user-controlled models of access control to user's data [72]. Hence, the underlying security, privacy, and trust support in Hydra is limited to the cryptographic support through TrustManager.

GSN [69], another service-based middleware, explicitly includes a security model through its integrity service and access control. The security model sits on top of the data and query layers and supports authorization property by controlling access to data and query. Only authorized users can have access to them. The integrity service is designed to support the integrity and confidentiality properties through encryption and electronic signature. In addition, GSN is designed to support access control and integrity supports at different levels of granularity; from whole GSN containers to individual virtual sensors. Though the architecture is designed with security considerations, the details on the underlying implementation models are difficult to access through two key papers on GSN [69], [75] and XGSN [76]. Hence, we conclude that the support for security, privacy, and trust is limited in GSN as well as its extended version, XGSN.

The OpenIoT middleware is also a service-based middleware as it is built with GSN as the core. The OpenIoT security and privacy framework describes a number of essential features that an IoT middleware is required to provide support in security, privacy, and trust [77]. For example, the support of data integrity through message digest, confidentiality through private key cryptography, key exchange through public key cryptography, authentication through digital signature, identity management through digital certificates, and key management through keytool and keystore. In addition, it has outlined a number of security protocols that are essential for IoT such as IEEE802.15.4, IPSec, TSL, and HTTPS. Furthermore, access control mechanisms such as mandatory access control, rule-based access control, lattice-based access control, and information flow-based access control, are also discussed as fundamental requirements for the IoT middleware. However, the implementation details of these security mechanisms in OpenIoT platform are not available. There are two specific issues with the OpenIoT security and privacy framework: 1) most underlying techniques described in the model are generic and not specific to the resource-constrained IoT environment and 2) no public implementations of the security framework are available.

In Virtus [78], a service-based middleware system, security is integrated with the system by design. Virtus uses the XMPP protocol to provide secure event driven communication. Authentication is supported by the transport layer security (TLS) protocol and encryption by the simple authentication and security layer (SASL) protocol. TLS also ensures the confidentiality and data integrity, whereas SASL guarantees server validation through an XMPP-specific profile, by means of authentication. In addition, the Virtus architecture is designed to allow the isolation of one or more instances from the Internet so that highly sensitive data can be kept private.

b) Cloud-based IoT middleware: The details on the support of security, privacy, and trust in the cloud-based systems like Google Fit, Paraimpu, and Xively are not available to make a fair assessment. As these systems rely on cloud-based architecture, a number of security and privacy issues related to cloud systems are applicable to them [79]. One of the major concerns is the control of sensitive information in the cloud-based system, as the data is controlled by the service provider. In addition, the limited security support provided by these systems are not designed for the resource-constrained IoT environment.

In Webinos [80], another cloud-based IoT middleware developed under the European FP7 project, the security and privacy is treated as a first class citizen and included while designing the system rather than an after-thought solution as in OpenIoT. The system uses the concept of *Personal Zone*, a secure virtual overlay network that groups a user's personal devices and services. Each user in the system has a trusted personal zone hub (PZH) instance running in the cloud. The PZH acts as a centralized authority to enforce policies. The policies are defined using XACML [81], a general-purpose access control language. The privacy is enforced within Webinos by disclosing the minimum amount of data via disabling the collection of contextualized data, and automatic filtering of data related to personal information.

c) Actor-based IoT middleware: The security support for the actor-based IoT middleware, e.g., Ptolemy accessor host [82], is still in the very early stage of development. Terraswam has identified the importance of security, privacy, and trust issues in their whitepaper in 2012 [83]. However, the solutions have not been yet built into the systems. Calvin [33] also identified security as a major issue, but it is still working toward it. Another actor-based middleware, Node-RED, is by default not secured; anyone who can access the IP address and port can access the editor and deploy the changes. IBM is developing a more secure solution by building the Watson IoT Platform in Bluemix which allows devices to connect securely using a unique combination of client ID and authentication token over TLS V1.2.

In summary, we concur with the observations made by Fremantle and Scott [72] that the existing middleware platforms can be categorized into three groups as follows:

1) middleware platforms that do not address security, privacy, and trust such as Calvin and Node-RED;
2) middleware platforms that address the security issues to some extent such as Hydra, Virtus and Webinos;
3) middleware platforms that offer theoretical models or white paper, but have not demonstrated any real-world implementation or concrete approaches such as OpenIoT and Ptolemy accessor host.

Furthermore, the security mechanisms supported by the middleware heavily rely on the underlying protocols. For example, LinkSmart uses SOAP/Web services model of security, VIRTUS uses XMPP standards and Webinos utilizes policy-based access control (XACML) and Federated Identity tokens (OpenID). It is important to note that these protocols are not designed for a resource-constrained environment (like IoT) and hence could be too costly both in memory and time.

3) Open Problems and Suggested Solutions:

Fremantle and Scott [72] have identified the following gaps in the existing middleware platforms: 1) lack of privacy-by-design in the middleware systems; 2) context-based security; 3) user-centric model of access control; and 4) support for federated identity at the device level. In addition to these, there are a number of other issues related to security, privacy, and trust that need to be addressed. In the following, we discuss some open issues and possible solutions.

a) Lightweight device authentication: IoT typically operates in a resource-constrained environment such as low-bandwidth communication between nodes, low energy, small memory, and slow CPU cycles. These characteristics directly impact the security, privacy, and trust solutions designed for such an environment. For example, the use of small packets in IEEE 802.15.4 may result in fragmentation of security protocol and may open new attack vectors. Similarly, limitations of CPU and memory put severe restrictions on the utility of the resource-demanding public-key cryptography as used in the Internet security. There are a number of lightweight public key cryptographic techniques that have been developed to overcome this problem. NTRU [84], ECC [85], and AE [86] are examples of such cryptographic techniques. Further research is needed in this area. The question from the middleware perspective is how these light-weight security mechanisms can be incorporated into the IoT middleware.

b) Denial-of-service attacks: The memory and processing constraints also lead to new form of denial-of-service attacks through resource exhaustion. The attacks may go unnoticed until the services become unavailable. A number of defense mechanisms have been developed for such attacks such as DTLS [87], HIP [88], and their variants. Despite these solutions proposed to address the problem, the issue remains open for further research. None of the middleware solutions reported in the literature considers this problem and potential solutions in their design.

c) End-to-end security: The gap between the Internet protocols (IPV6) and the light-weight IoT protocols (6LoWPAN) makes it difficult to achieve end-to-end security in the IoT environment. Though 6LoWPAN [89] and CoAP [90] progress toward reducing the gap by compressing IPV6 packets, there still remains the subtle difference. The protocol translations at gateways might not work with the existing security solutions. First, the relevant information for translation is encrypted and might not be available to gateways. Second, the changes made at the gateways might invalidate the end-to-end data integrity protection. A number of solutions have been proposed toward addressing this such as sharing keys with gateways, selective protection, transformation independent message authentication, etc. However, it still remains as a challenging problem, and needs further work. The middleware can play a significant role to alleviate this problem. Furthermore, achieving security at the IoT edges is challenging for two reasons. First, the traditional perimeter defense does not work when the edges are distributed in the unprotected environment. Second, the cloud data center cannot expose itself to the end devices. Some of the techniques that could potentially solve these problems include software defined perimeter, zero trust, and deperimeterization: though they are at the early stage of the development.

In summary, IoT middleware needs to be designed with security, privacy, and trust as first class citizens rather than after-thought. It should satisfy key security properties in all layers in the IoT architecture, for all entities including end users and devices. Furthermore, the underlying techniques also need to satisfy the constrains imposed by the IoT environment including big data, low powered devices, smaller processors, etc. To date, existing approaches tend to adapt available solutions for existing middleware as an after-thought. Such a patchwork approach is not going to satisfy the needs of ever-growing IoT applications. Hence, building trustworthy IoT middleware is still an open research and development problem. We believe emerging techniques such as privacy-by-design, differential privacy, and light-weight public-key cryptography will form the building blocks of such IoT middleware solutions.

VI. Related Work

The IoT has become a vibrant and rapidly expanding area of research and development over the last few years. A number of surveys have been conducted on IoT middleware. The types of middleware covered in these surveys vary considerably, and rapid technological advancements and new products bring new challenges and opportunities. The majority of existing survey papers cover only the wireless sensor network domain, while others concentrate on semantic and service-based frameworks, and a small minority covers only the hardware or protocol level middleware solutions. In this section, we briefly overview these surveys.

Bandyopadhyay *et al.* [13] proposed key functional blocks for service-based IoT middleware, including interoperation, context detection, device discovery and management, security and privacy, and persistency management. Bandyopadhyay *et al.* [13] provided a classification of ten existing IoT middleware systems based on the proposed functional blocks and concentrate on how those middleware systems provide adaptations. The main conclusion drawn from the survey is that context-awareness, security and privacy, and interoperation are the least supported functional blocks among the surveyed middleware systems. Chaqfeh and Mohamed [14] investigated the major challenges and solutions in IoT middleware. The identified list of challenges include interoperability, scalability, device abstraction, spontaneous interaction, unfixed infrastructure, multiplicity, and security and privacy. Chaqfeh and Mohamed [14] classified the existing solutions into three domains: 1) semantics Web and Web services; 2) RFID and sensors networks; and 3) robotics-based systems. Chaqfeh and Mohamed [14] argued that Semantic Web and Web service IoT middleware could address all the listed challenges except the scalability of persistence service. We concur that semantic Web is a viable solution for addressing service discovery and composition in IoT. However, it is a heavy-weight solution for resource-constrained IoT devices.

Teixeria *et al.* [15] focused on the challenges associated with service-oriented IoT middleware and propose a solution for service discovery and data composition using a probabilistic approach to deal with the problems of scale, unknown service availability and data inaccuracies. The work is part of the EC-funded project large scale choreographies for the future Internet (CHOReOS). Finally, a very recent survey by Razzaque *et al.* [12] provided a comprehensive overview of IoT middleware, focusing on the requirements of a variety of middleware types, and exploring some of the challenges related to these requirements. The requirements are grouped into two general categories: 1) middleware service requirements (functional and nonfunctional) and 2) architectural requirements. Each of the challenges is discussed at a high-level without delving on how they have been addressed by some of the most popular middleware used in practice today.

To the best of our knowledge, none of these existing surveys addresses the more recent trend of light-weight plug-and-play or cloud-based IoT middleware. This paper is the first IoT middleware survey that emphasizes on the architectural patterns with emphasis on empowering users to create innovative IoT applications targeted for ambient data collection and analysis. We attempt to explore in depth those architectural characteristics that can make each middleware type suitable or unsuitable for a variety of applications, and the features that might enable or hinder wide user adoption, especially, in the case of users with low programming experience.

VII. Conclusion

The World Wide Web has gone through many transformations, from traditional linking and sharing of computers and documents, to a platform for conducting businesses and connecting people via social media, and now the emerging paradigm of connecting billions of physical objects (IoT) to empower human interaction with both the physical and virtual worlds in an unprecedented way. In this survey paper, we have analyzed three key IoT middleware architectures ranging from consumer-centric cloud-based architectures, light-weight actor-based architectures, and heavy-weight service-based architectures. We outlined four key challenges in developing an IoT middleware which are as follows.

1) A light-weight middleware platform that can provide similar services when deployed on power constrained IoT devices as well as in desktop computers and cloud infrastructure.
2) A composition engine that is intuitive and not application specific.
3) A security mechanism that can operate in a resource constrained environment and yet can achieve similar guarantee as Internet security.
4) A semantic-based IoT device/service discovery that goes beyond discovery of domain names and IP addresses.

We elaborate on two nonontological solutions for addressing key challenges in IoT service discovery. The first approach is adapted from existing works in Web service search engines and the second approach is based on machine learning and recommendation techniques. Finally, in the IoT security domain, we believe emerging techniques such as privacy-by-design, differential privacy, and light-weight public-key cryptography will form the building blocks for security in IoT middleware.

Acknowledgment

The majority of this paper was performed while the first author was visiting CSIRO, Sydney, Australia. The work on accessors was performed while the first author was visiting Prof. E. Lee and C. Brooks at the University of California, Berkeley.

References

[1] D. Raggett, “The Web of Things: Challenges and opportunities,” *IEEE Comput.*, vol. 48, no. 5, pp. 26–32, May 2015.

[2] R. Want, B. N. Schilit, and S. Jenson, “Enabling the Internet of Things,” *IEEE Comput.*, vol. 48, no. 1, pp. 28–35, Jan. 2015.

[3] L. Baresi, L. Mottola, and S. Dustdar, “Building software for the Internet of Things,” *IEEE Internet Comput.*, vol. 19, no. 2, pp. 6–8, Mar./Apr. 2015.

[4] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A survey,” *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, 2010.

[5] L. Yao, Q. Z. Sheng, and S. Dustdar, “Web-based management of the Internet of Things,” *IEEE Internet Comput.*, vol. 19, no. 4, pp. 60–67, Jul./Aug. 2015.

[6] Y. Qin *et al.*, “When things matter: A survey on data-centric Internet of Things,” *J. Netw. Comput. Appl.*, vol. 64, pp. 137–153, Feb. 2016.

[7] K. Aberer, M. Hauswirth, and A. Salehi, “A middleware for fast and flexible sensor network deployment,” in *Proc. 32nd Int. Conf. Very Large Data Bases*, Seoul, South Korea, 2006, pp. 1199–1202. [Online]. Available: http://dl.acm.org/citation.cfm?id=1182635.1164243

[8] E. Latronico *et al.*, “A vision of swarmlets,” *IEEE Internet Comput.*, vol. 19, no. 2, pp. 20–28, Mar./Apr. 2015.

[9] *Google Fit*. (2015.) [Online]. Available: https://developers.google.com/fit/

[10] *Architectural Styles and Design of Network-Based Software Architectures*. (2000.) [Online]. Available: www.ics.uci.edu/∼fielding/pubs/dissertation/top.htm

[11] T. Zachariah *et al.*, “The Internet of Things has a gateway problem,” in *Proc. HotMobile*, Santa Fe, NM, USA, 2015, pp. 27–32.

[12] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, “Middleware for Internet of Things: A survey,” *IEEE Internet Things J.*, vol. 3, no. 1, pp. 70–95, Feb. 2016.

[13] S. Bandyopadhyay, M. Sengupta, S. Maiti, and S. Dutta, “Recent trends in wireless and mobile networks,” in *Proc. 3rd Int. Conf. WiMo CoNeCo*, Ankara, Turkey, 2011, pp. 288–296, doi: http://dx.doi.org/10.1007/978-3-642-21937-5_27

[14] M. A. Chaqfeh and N. Mohamed, “Challenges in middleware solutions for the Internet of Things,” in *Proc. Int. Conf. Collaboration Technol. Syst. (CTS)*, Denver, CO, USA, 2012, pp. 21–26.

[15] T. Teixeira, S. Hachem, V. Issarny, and N. Georgantas, “Service oriented middleware for the Internet of Things: A perspective,” in *Proceedings of the 4th European Conference on Towards a Service-Based Internet, ServiceWave*, Poznan, Poland: Springer-Verlag, 2011, pp. 220–229.

[16] M. A. Gutierrez, M. L. Fast, A. H. Ngu, and B. J. Gao, “Real-time prediction of blood alcohol content using smartwatch sensor data,” in *Proc. IEEE Int. Conf. Smart Health*, Phoenix, AZ, USA, 2015, pp. 175–186.

[17] M. P. Papazoglou and D. Georgakopoulos, “Introduction: Service-oriented computing,” *Commun. ACM*, vol. 46, no. 10, pp. 24–28, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/944217.944233

[18] J. Soldatos *et al.*, “OpenIoT: Open source Internet-of-Things in the cloud,” in *Interoperability and Open-Source Solutions for the Internet of Things* (LNCS 9001). I. P. Žarko, K. Pripužić, and M. Serrano, Eds. Cham, Switzerland: Springer, 2015, pp. 13–25, doi: http://dx.doi.org/10.1007/978-3-319-16546-2_3

[19] *The TerraSwarm Research Center*. (2013.) [Online]. Available: https://terraswarm.org

[20] *Nest*. (2014.) [Online]. Available: https://developer.nest.com

[21] *IoT Related Projects*. (2015.) [Online]. Available: https://www.terraswarm.org/terraswarm/wiki/Main/RelatedProjects

[22] M. Eisenhauer, P. Rosengren, and P. Antolin, “A development platform for integrating wireless devices and sensors into ambient intelligence systems,” in *Proc. Sensor Mesh Ah Hoc Commun. Netw. Workshops (SECON)*, Rome, Italy, 2009, pp. 1–3.

[23] *HYDRA*. (2010.) [Online]. Available: http://hydramiddleware.eu

[24] *Global Sensor Networks*. (2004.) [Online]. Available: http://lsir.epfl.ch/research/current/gsn/

[25] B. Burke and R. Monson-Haefel, *Enterprise Java Beans 3.0*. Sebastopol, CA, USA: O’Reilly, 2006.

[26] J.-P. Calbimonte, S. Sarni, J. Eberle, and K. Aberer, “XGSN: An open-source semantic sensing middleware for the Web of Things,” in *Proc. 7th Int. Workshop Semantic Sensor Netw.*, Trentino, Italy, 2014, pp. 51–66.

[27] *Xively*. (2014.) [Online]. Available: http://xively.com

[28] A. Pintus, D. Carboni, and A. Piras, “Paraimpu: A platform for a social Web of Things,” in *Proc. 21st Int. Conf. World Wide Web*, Perth, WA, Australia, 2012, pp. 401–404. [Online]. Available: http://doi.acm.org/10.1145/2187980.2188059

[29] *Paraimpu*. (2014.) [Online]. Available: https://www.paraimpu.com

[30] *Tornado Web Server*. (2015.) [Online]. Available: www.tornadoweb.org/en/stable

[31] *Open Source Web Server Load Balancer*. (2015.) [Online]. Available: https://www.nginx.com/products/

[32] *MongoDB Documentation*. (2015.) [Online]. Available: https://docs.mongodb.com/manual/reference/database-references/

[33] P. Persson and O. Angelsmark, “Calvin—Merging cloud and IoT,” *Proc. Comput. Sci.*, vol. 52, pp. 210–217, Dec. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915008595

[34] *Node-RED, A Visual Tool for Wiring the Internet of Things*. (2015.) [Online]. Available: http://nodered.org

[35] *Ptolemy II*. (1996.) [Online]. Available: http://ptolemy.eecs.berkeley.edu

[36] *Kepler Scientific Workflow Engine*. (2011.) [Online]. Available: https://kepler-project.org

[37] P. Barnaghi, A. Sheth, and C. Henson, "From data to actionable knowledge: Big data challenges in the Web of Things," *IEEE Intell. Syst.*, vol. 28, no. 6, pp. 6–11, Nov./Dec. 2013.

[38] D. Niewolny, "How the Internet of Things is revolutionizing healthcare," White Paper, 2013.

[39] P. P. Jayaraman, D. Palmer, A. Zaslavsky, and D. Georgakopoulos, "Do-it-yourself digital agriculture applications with semantically enhanced IoT platform," in *Proc. IEEE 10th Int. Conf. Intell. Sensors Sensor Netw. Inf. Process. (ISSNIP)*, Singapore, 2015, pp. 1–6.

[40] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini, "Security, privacy and trust in Internet of Things: The road ahead," *Comput. Netw.*, vol. 76, pp. 146–164, Jan. 2015.

[41] *Linked Sensor Middleware (LSM)*. (2014.) [Online]. Available: http://open-platforms.eu/library/deri-lsm/

[42] *Sensor Model Language (SensorML)*. (2012.) [Online]. Available: http://www.opengeospatial.org/standards/sensorml

[43] H. Wang, C. C. Tan, and Q. Li, "Snoogle: A search engine for pervasive environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 8, pp. 1188–1202, Aug. 2010.

[44] C. C. Tan, B. Sheng, H. Wang, and Q. Li, "Microsearch: A search engine for embedded devices used in pervasive computing," *ACM Trans. Embedded Comput. Syst.*, vol. 9, no. 4, 2010, Art. no. 43.

[45] K.-K. Yap, V. Srinivasan, and M. Motani, "MAX: Wide area human-centric search of the physical world," *ACM Trans. Sensor Netw.*, vol. 4, no. 4, 2008, Art. no. 26.

[46] B. Ostermaier, K. Römer, F. Mattern, M. Fahrmair, and W. Kellerer, "A real-time search engine for the Web of Things," in *Proc. Int. Conf. Internet Things (IOT)*, Tokyo, Japan, 2010, pp. 1–8.

[47] P. P. Jayaraman, J.-P. Calbimonte, and H. N. M. Quoc, "The schema editor of OpenIoT for semantic sensor network," in *Proc. Semantic Sensor Netw. Terra Cognita (SSN-TC)*, Bethlehem, PA, USA, 2015, pp. 25–30.

[48] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang, "Similarity search for Web services," in *Proc. VLDB*, Toronto, ON, Canada, 2004, pp. 372–383.

[49] Y. Zhang, Z. Zheng, and M. Lyu, "WSExpress: A QoS-aware search engine for Web services," in *Proc. ICWS*, Miami, FL, USA, 2010, pp. 91–98.

[50] J. Wu, L. Chen, Y. Xie, and Z. Zheng, "Titan: A system for effective Web service discovery," in *Proc. WWW*, Lyon, France, 2012, pp. 441–444.

[51] A. H. H. Ngu, J. Ma, Q. Z. Sheng, L. Yao, and S. Julian, "A similarity-based Web service search engine," in *Proc. ACM Conf. Inf. Retrieval (SIGIR)*, Gold Coast, QLD, Australia, 2014, pp. 1251–1252.

[52] L. Yao, Q. Z. Sheng, A. H. Ngu, and X. Li, "Things of interest recommendation by leveraging heterogeneous relations in the Internet of Things," *ACM Trans. Internet Technol.*, vol. 16, no. 2, 2016, Art. no. 9.

[53] L. Yao, Q. Zheng, N. J. Falkner, and A. H. Ngu, "ThingsNavi: Finding most-related things via multi-diminesional modeling of human-thing interactions," in *Proc. 12th Int. Conf. Mobile Ubiquitous Syst. Comput. Netw. Serv. (MobiQuitous)*, London, U.K., 2014, pp. 20–29.

[54] J. Jang-Jaccard and S. Nepal, "A survey of emerging threats in cybersecurity," *J. Comput. Syst. Sci.*, vol. 80, no. 5, pp. 973–993, 2014.

[55] J. Lopez, R. Roman, and C. Alcaraz, "Analysis of security threats, requirements, technologies and standards in wireless sensor networks," in *Foundations of Security Analysis and Design V*. Heidelberg, Germany: Springer, 2009, pp. 289–338.

[56] R. H. Weber, "Internet of Things—New security and privacy challenges," *Comput. Law Security Rev.*, vol. 26, no. 1, pp. 23–30, 2010.

[57] J. H. Ziegeldorf, O. G. Morchon, and K. Wehrle, "Privacy in the Internet of Things: Threats and challenges," *Security Commun. Netw.*, vol. 7, no. 12, pp. 2728–2742, 2014.

[58] C. Dwork, "Differential privacy: A survey of results," in *Theory and Applications of Models of Computation, 5th International Conference, TAMC 2008*, Xi'an, China: Springer, 2008, pp. 1–19.

[59] M. Langheinrich, "Privacy by design—Principles of privacy-aware ubiquitous systems," in *Ubicomp 2001: Ubiquitous Computing, Third International Conference*, Atlanta, GA, USA: Springer, 2001, pp. 273–291.

[60] B. C. M. Fung, K. Wang, R. Chen, and P. S. Yu, "Privacy-preserving data publishing: A survey of recent developments," *ACM Comput. Surveys*, vol. 42, no. 4, 2010, Art. no. 14.

[61] M. Burrows, M. Abadi, and R. M. Needham, "A logic of authentication," *Proc. Roy. Soc. Math. Phys. Eng. Sci.*, vol. 426, no. 1871, pp. 233–271, 1989.

[62] S. Cirani, M. Picone, P. Gonizzi, L. Veltri, and G. Ferrari, "IoT-OAS: An OAuth-based authorization service architecture for secure services in IoT scenarios," *IEEE Sensors J.*, vol. 15, no. 2, pp. 1224–1234, Feb. 2015.

[63] D. J. Weitzner *et al.*, "Information accountability," *Commun. ACM*, vol. 51, no. 6, pp. 82–87, 2008.

[64] R. Roman, J. Zhou, and J. Lopez, "On the features and challenges of security and privacy in distributed Internet of Things," *Comput. Netw.*, vol. 57, no. 10, pp. 2266–2279, 2013.

[65] D. Kozlov, J. Veijalainen, and Y. Ali, "Security and privacy threats in IoT architectures," in *Proc. 7th Int. Conf. Body Area Netw.*, Oslo, Norway, 2012, pp. 256–262.

[66] M. Abomhara and G. M. Køien, "Security and privacy in the Internet of Things: Current status and open issues," in *Proc. Int. Conf. Privacy Security Mobile Syst. (PRISMS)*, Aalborg, Denmark, 2014, pp. 1–8.

[67] S. Bandyopadhyay, M. Sengupta, S. Maiti, and S. Dutta, "A survey of middleware for Internet of Things," in *Recent Trends in Wireless and Mobile Networks*. Heidelberg, Germany: Springer, 2011, pp. 288–296.

[68] M. Eisenhauer, P. Rosengren, and P. Antolin, "HYDRA: A development platform for integrating wireless devices and sensors into ambient intelligence systems," in *The Internet of Things: 20th Tyrrhenian Workshop on Digital Communications.*, New York, NY, USA: Springer, 2010, pp. 367–373.

[69] K. Aberer, M. Hauswirth, and A. Salehi, "A middleware for fast and flexible sensor network deployment," in *Proc. 32nd Int. Conf. Very Large Data Bases*, Seoul, South Korea, 2006, pp. 1199–1202.

[70] L. M. S. de Souza *et al.*, "SOCRADES: A Web service based shop floor integration infrastructure," in *The Internet of Things: First International Conference, IOT 2008*, Zürich, Switzerland: Springer, 2008, pp. 50–67.

[71] R. J. C. Benito *et al.*, "SMEPP: A secure middleware for embedded P2P," in *Proc. ICT-MobileSummit*, vol. 9. Santander, Spain, 2009, Art. no. 135.

[72] P. Fremantle and P. Scott, "A security survey of middleware for the Internet of Things," *PeerJ PrePrints*, vol. 3, Jul. 2015, Art. no. e1521.

[73] P. Kostelnik, M. Sarnovsk, and K. Furdik, "The semantic middleware for networked embedded systems applied in the Internet of Things and services domain," *Scalable Comput. Pract. Experience*, vol. 12, no. 3, pp. 307–316, 2011.

[74] A. Ekelhart, S. Fenz, G. Goluch, M. Steinkellner, and E. Weippl, "XML security—A comparative literature review," *J. Syst. Softw.*, vol. 81, no. 10, pp. 1715–1724, 2008.

[75] O. Corcho, J.-P. Calbimonte, H. Y. Jeung, and K. Aberer, "Enabling query technologies for the semantic sensor Web," *Int. J. Semantic Web Inf. Syst.*, vol. 8, no. 1, pp. 43–63, 2012.

[76] J.-P. Calbimonte, S. Sarni, J. Eberle, and K. Aberer, "XGSN: An open-source semantic sensing middleware for the Web of Things," in *Proc. Terra Cognita Semantic Sensor Netw.*, Trento, Italy, 2014, pp. 51–66.

[77] J.-P. Calbimonte, M. Riahi, and A. Zaslavsky, "Privacy and security framework. OpenIoT deliverable D522," Distrib. Inf. Syst. Lab., École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, Tech. Rep. 210925, 2014.

[78] D. Conzon *et al.*, "The virtus middleware: An XMPP based architecture for secure IoT communications," in *Proc. 21st Int. Conf. Comput. Commun. Netw. (ICCCN)*, Munich, Germany, 2012, pp. 1–6.

[79] S. Nepal and M. Pathan, *Security, Privacy and Trust in Cloud Systems*. Heidelberg, Germany: Springer, 2014.

[80] H. Desruelle, J. Lyle, S. Isenberg, and F. Gielen, "On the challenges of building a Web-based ubiquitous application platform," in *Proc. ACM Conf. Ubiquit. Comput.*, Pittsburgh, PA, USA, 2012, pp. 733–736.

[81] T. Moses *et al.*, *Extensible Access Control Markup Language (XACML) Version 2.0*, Oasis Standard 200502, 2005.

[82] E. Latronico *et al.*, "A vision of swarmlets," *IEEE Internet Comput.*, vol. 19, no. 2, pp. 20–28, Mar./Apr. 2015.

[83] D. Blaauw *et al.*, "The terraswarm research center (TSRC)," White Paper, 2012.

[84] J. Hoffstein, J. Pipher, and J. H. Silverman, "NSS: An NTRU lattice-based signature scheme," in *Advances in Cryptology—EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques*, Innsbruck, Austria: Springer, 2001, pp. 211–228.

[85] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab, "NanoECC: Testing the limits of elliptic curve cryptography in sensor networks," in *Wireless Sensor Networks: 5th European Conference, EWSN 2008*, Bologna, Italy: Springer, 2008, pp. 305–320.

[86] I. Anshel, M. Anshel, D. Goldfeld, and S. Lemieux, "Key agreement, the algebraic eraser and lightweight cryptography," *Contemp. Math.*, vol. 418, pp. 1–34, 2006.

[87] D. McGrew and E. Rescorla, *Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-Time Transport Protocol (SRTP)*, Internet Eng. Task Force, Fremont, CA, USA, 2010.

[88] A. Gurtov, *Host Identity Protocol (HIP): Towards the Secure Mobile Internet*, vol. 21. Chichester, U.K.: Wiley, 2008.

[89] Z. Shelby and C. Bormann, *6LoWPAN: The Wireless Embedded Internet*. vol. 43. Somerset, U.K.: Wiley, 2011.

[90] Z. Shelby, K. Hartke, and C. Bormann, *The Constrained Application Protocol (COAP)*, Internet Eng. Task Force, Fremont, CA, USA, 2014.

Anne H. Ngu is currently a Full Professor with the Department of Computer Science, Texas State University, San Marcos, TX, USA. From 1992 to 2000, she was a Senior Lecturer with the School of Computer Science and Engineering, University of New South Wales, NSW, Australia. She had held research scientist positions with Telcordia Technologies, Piscataway, NJ, USA, Microelectonics and Computer Technology, Austin, TX, USA, the University of California, Berkeley, CA, USA, CSIRO, Sydney, NSW, Australia, and Tilburg University, Tilburg, The Netherlands. She was a Summer Faculty Scholar with Lawrence Livermore National Laboratory, Livermore, CA, USA, from 2003 to 2006. She has published over 100 technical papers in journals and refereed conferences in computer science. Her current research interests include large-scale service and information discovery and integration, service platforms for Internet of Things, business and scientific workflows, and databases.

Prof. Ngu was a recipient of the professional service features key leadership roles in three international conferences. She was bestowed with a special Outstanding Contribution Award for one of them. She was also a recipient of the 2013 NCWIT Undergraduate Research Mentoring Award.

Mario Gutierrez received the B.Sc. degree in computer science from Texas State University, San Marcos, TX, USA, in 2015.

He is currently an independent video game developer. He has authored two papers on the applications of machine learning. His current research interests include artificial intelligence, game mechanics, and computer graphics.

Vangelis Metsis received the B.S. degree in computer science from the Department of Informatics, Athens University of Economics and Business, Athens, Greece, in 2005, and the Ph.D. degree from the Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX, USA, in 2011.

He is an Assistant Professor with the Department of Computer Science, Texas State University, San Marcos, TX, USA. He has federal support and has a strong publication record in related research areas, such as human activity and behavior analysis, human–computer interaction and motion tracking using sensors, sleep monitoring, and human biosignal analysis. His current research interests include machine learning, data mining, and computer vision with special focus on applications of smart health and wellbeing, and pervasive computing.

Surya Nepal received the B.E. degree from the National Institute of Technology at Surat, Surat, India, the M.E. degree from the Asian Institute of Technology, Bangkok, Thailand, and the Ph.D. degree from RMIT University, Melbourne, VIC, Australia.

He is a Principal Research Scientist with Data61, CSIRO, Sydney, NSW, Australia, where he currently leads the Distributed Systems Security Group. He has over 150 peer-reviewed publications to his credit; his papers are published in international journals, such as the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON SERVICE COMPUTING, *ACM Transactions on Internet Technologies*, and the IEEE TRANSACTIONS ON COMPUTERS. He has coedited three books including security, privacy, and trust in cloud systems by Springer. His current research interests include development and implementation of technologies in the area of distributed systems and social networks, with a specific focus on security, privacy, and trust.

Quan Z. Sheng (M'05) received the Ph.D. degree in computer science from the University of New South Wales, Sydney, NSW, Australia, in 2006.

He is a Full Professor and the Head of the Department of Computing, Macquarie University, Sydney. He has authored and coauthored over 270 publications His current research interests include Web of Things, Internet of Things, big data analytics, service computing, distributed computing, and Internet technologies.

Prof. Sheng is a member of the ACM. He was a recipient of the ARC Future Fellowship in 2014, the Chris Wallace Award for Outstanding Research Contribution in Computer Science in 2012, and the Microsoft Research Fellowship in 2003.