An IoT Edge Computing Framework Using Cordova Accessor Host

Anne H.H. Ngu, Jesuloluwa Eyitayo, Guowei Yang, Colin Campbell, Quan Z. Sheng, and Jianyuan Ni

***Abstract*—The Internet of Things (IoT) is a rapidly growing system of physical sensors and connected devices, enabling advanced information gathering, interpretation, and monitoring. The realization of a versatile IoT edge computing framework will accelerate seamless integration of the cyber-world with new physical IoT devices, and will fundamentally change and empower the way humans interact with the world. While there are many cloud-based IoT computing frameworks, they cannot support the needs of IoT applications that require local processing and guarantee of consumer's privacy. This paper presents experimentation with the open source plug and play IoT middleware called Cordova Accesor Host. We demonstrated that Cordova Accessor Host supports the essential ingredients of the composition and re-usability of IoT services using accessor as the basic building block and adopting an accessor-module-plugin design pattern. The portability is demonstrated by using the same accessor for collecting sensor data from radically different IoT devices such as wearables (e.g., smartwatches) and micro-controllers (e.g., Arduino). Our energy profiling experiments show that IoT services deployed using Cordova Accessor Host consume around 35% less battery power than the same IoT services deployed in the native Android operating system.**

***Index Terms*—Service Middleware and Platform, Open Service Platform, Edge Computing**

I. Introduction

The Internet of Things (IoT) is a domain that represents the next most exciting technological revolution since the Internet [1], [2], [3], [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 earthquake, forest fire and many other devastating disasters. IoT can transform manufacturing, making it leaner and smarter. In recent years, IoT is increasingly seen as the technology that will transform healthcare by providing unobtrusive and passive monitoring of a patient's vitals via sensors [5].

While 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 ones [6], [7], [8], [2], [9]. Many IoT middleware and connectivity protocols are being developed and the number is still increasing each day. For example, Message Oriented Telemetry Transport (MQTT), Constrained Application Protocol (CoAP) and BLE (Bluetooth Low Energy) 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 building applications to collect and interpret the data to gain wisdom from them. This is compounded by the fact that each IoT middleware advocates a different programming abstraction and architecture for accessing and connecting to IoT devices [1]. For example, in the Global Sensor Network (GSN) [10] 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. One of the challenges of using virtual sensor is that it can incur additional latency due to the overhead of mapping virtual sensors to physical sensors. All virtual sensors must be hosted in a powerful cloud server. In the Node-RED project at IBM (http://nodered.org), a node is proposed as the main abstraction. In the Google Fit project[1], no particular high level abstraction is provided for encapsulating a new device type. The system is pre-programmed to support a fixed set of IoT devices, which can be accessed by Representational State Transfer (REST) APIs [11]. Adding an IoT device that is not supported requires expert Java programming experience in extending Google Fit's `FitnessSensorService` class. In addition, data are collected and stored solely in the cloud in GoogleFit, which might not be acceptable for privacy conscious consumers. The Kubernetes Open Programmable Acceleration Engine project[2] is a recent player tackling this problem emphasizing on robustness of IoT edge computing via their orchestration engine. The importance of a light-weight IoT edge computing is also highlighted in [12].

The current state-of-the-art support for IoT service development is application specific, similar to the scenario where every IoT device requires a different Web browser for connection to the Internet as echoed by Zachariah et al. [13]. In order for consumers to tap into the next IoT revolution, there is an urgent need to launch an IoT computing framework in the same spirit as the launch of the Web frameworks such as XAMPP, Laravel, ASP.NET, Django and Express.js that revolutionized how Web applications can be engineered and built for on-line businesses

A. Ngu, J. Eyitayo, G. Yang, C. Campbell, J. Ni are with the Department of Computer Science, Texas State University, San Marcos, Texas, USA. Email: {angu, Jesuloluwa, gyang, c_c953 j_n317}@txstate.edu

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

[1]https://developers.google.com/fit/.

[2]https://opae.github.io/latest/index.html.

using ensemble of micro services, as well as the launch of the mobile operating systems (iOS and Android) that brought disruptive applications such as AirBnB and Uber. We believe the launch of an IoT edge computing framework, which can be deployed on the computation constrained edge devices, can open up an astounding range of IoT services that have impacts beyond our imagination and fundamentally change and empower human interaction with the physical world [14].

This paper demonstrates the opportunities and challenges of using the open source Cordova Accessor Host[3] as an IoT edge computing framework. Cordova host leverages the cross platform design of Apache Cordova tools, its plugins, and the integration with accessor host [15] for instantiating and executing network of accessors as IoT services running on edge devices. An experimental Cordova host has been developed and listed on TerraSwarm's Accessor project website[4] since 2016, but there has never been a development of a real-world IoT service/application using that experimental host. There is no empirical evaluation of the benefits or opportunities of using Cordova host for IoT services development in terms of composibilty and re-usability of accessors, reduction of programming and deployment barriers, ease of plugins development for accessors, and the impact of battery power of the edge device with running an Cordova host.

We first demonstrated in this paper how Cordova host is used for the development and deployment of a real-world Fall Detection App (an IoT service that will improve the health of the older population) as a composition of accessors running on a WearOS or a MSBAND commodity-based smartwatch that paired with a smartphone. This involves development of Cordova plugins for WearOS and MSBAND. We then demonstrated the re-usability of accessors, by composing a different IoT service that can perform real-time heart rate monitoring. We also showed that IoT service composed based on accessors can be ported to smartwatches from different vendors with minimal programming via the accessor-module-plugin design pattern. We further demonstrated that a radically different hardware source e.g., the micro-controller based device such as Arduino can be used for data collection without any additional programming so long as the Cordova's plugin for that micro-controller is available or has been developed. Finally, we demonstrated the platform portability of Cordova based IoT services by showing the cross-platform deployment of the same IoT service to a different platform (iOS, Android, or Browser) using a simple *add platform* command available in Apache Cordova's tool. The main contributions of this paper are:

- Providing a methodology for using the Accessor-Module-Plugin design pattern for the creation of IoT applications in the Cordova host that facilitates the context-aware dynamic binding with native implementations.
- Investigating the effectiveness of the Cordova host for the rapid IoT service development by composing two different IoT services using accessors as basic components and measures the reusability and code changes required.
- Demonstrating Cordova host's support of the seamless integration with various type of IoT devices and facilitating sensor data collection across multi-vendors IoT devices (smartwatches from Google, Microsoft, Huawei and micro-kernel devices) with minimal additional programming.
- Demonstrating the energy and computation efficiency of accessor-based edge IoT services as compared with native implementation.

We are the first to show that Cordova Accessor host is a true edge-based IoT middleware that can be deployed on popular mobile phones and micro-controllers with no dependence on cloud technology. The novelty of this work comes from the implementation of the original conceptual and skeleton platform to demonstrate the practicality of a true edge-based IoT computing framework for the composition of more than a simple "Hello World" IoT application across radically different IoT devices and operating systems.

The remainder of this paper is organized as follows. We first present the related IoT service frameworks in Section II. This is followed by the background of the accessor design pattern and accessor hosts in Section III. In Section IV, we present the architecture of an edge-based IoT computing framework, Cordova host, and its capabilities. We present two IoT services developed and deployed on the Cordova host and document the reusability of accessors and reduction in deployment barriers across heterogeneous IoT devices. In Section V, we present the performance study in battery power consumption and memory utilization of IoT services implemented on the Cordova host verses the one on the native Android environment. Finally, in Section VI, we provide some concluding remarks and discuss the future work.

II. Related Work

We explored four classical IoT frameworks: *Service-oriented*, *Cloud based*, *Actor oriented*, and *Container-based*. We explored these four options in search of a framework that can allow seamless integration of heterogeneous IoT devices from multiple vendors to build real-time IoT applications with on the edge analysis and data storage yet without dictating a particular communication protocol or be dependent on cloud. The availability of edge/local storage is important to avoid the unpredictable latency from wireless transmission of data to the cloud or server for real-time analytic [16]. In addition, to ensure that users' privacy is not violated, users should have the option to archive data generated from their personal IoT devices in a secure local storage of their own choice.

The service oriented framework that we explored was Global Sensor Network (GSN) in [10]. GSN 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 developers to declaratively specify XML-based deployment descriptors to describe how to connect to a virtual sensor. This is similar to the concept of deployment descriptors used in the deployment of enterprise Java beans in J2EE server. The architecture of GSN follows the same container architecture as in J2EE where

[3]We use the term Cordova Accessor Host and Cordova host interchangeably.
[4]https://wiki.eecs.berkeley.edu/accessors/Main/CordovaHost.

each container can host multiple virtual sensors. The container provides functionalities/capabilities for lifecycle management of the sensors including persistence, security, communication, resource pooling and event processing. GSN servers can fulfill our local data storage requirement, however, GSN is a heavy weight system to run on an edge device like a smartwatch or smartphone. To date, there is no working edge-based GSN framework. Another service oriented platform for IoT is presented in [17]. The main architecture is similar to GSN. The key contribution of this service framework is its scalability and robust scheduling that has shown to support more than 1,000 services. However, it can only be deployed on high-end servers or cloud.

Another service-oriented IoT framework was proposed in [18]. This is a server/cloud level system where IoT services are composed based on task-oriented computing approach. This framework requires the development of a common IoT service ontology and adoption of a common standard.

We examined various cloud based frameworks such as AWS IoT from Amazon[5], Watson IoT from IBM[6], ThingSpeak IoT[7], and Google IoT Cloud[8] (e.g., GoogleFit). These cloud-based frameworks usually provide the following four fundamental services: 1) Web-based administrative console for managing device connection; 2) cloud-based data storage; 3) cloud-based analytic services, and 4) advanced reporting or visualization. We examined Google's GoogleFit cloud service in details for IoT application development. GoogleFit provides a set of APIs for connecting third-party IoT devices to its cloud storage. For example, it provides APIs for subscribing to a particular fitness data type or a particular fitness source (e.g., Fitbit or Samsung Smartwatch) and APIs for querying of historical data or persistent recording of the sensor data from a particular source (e.g., a smartwatch). With GoogleFit, the user is tied to storing his/her sensor data in GoogleFit's cloud storage, in the format dictated by GoogleFit and in the size limit enforced by GoogleFit. It is not possible to get access to the collected raw data and pre-process them for analysis and visualization, which is a critical component for many IoT applications. Moreover, GoogleFit requires all collected data to be stored remotely in the Google cloud. GoogleFit is not suitable for IoT services that must be performed quickly in real-time on board the edge device.

To date, there are several commercial edge-based computing frameworks that leverage container technology. For instance, Microsoft Azure IoT Edge [19] consists of three components: IoT Edge modules, IoT Edge runtime and a cloud-based interface. The first two components run on edge devices and the cloud-based interface allows remote monitoring of edge devices. The Azure IoT Edge runtime leverages Docker to run IoT Edge modules on the device with the embedded instructions on what modules to download and run via a connection to Microsoft Azure IoT Hub. The current Azure IoT Edge runtime engine only supports Windows and Linux systems, which means an IoT application developed using Azure's cloud cannot be deployed on popular edge devices that run Android, IOS or WearOS. Azure IoT Edge is thus a cloud based IoT computing framework like GoodgleFit. The developer is constrained by the cost of subscribing to the cloud and the type of APIs provided by the cloud's vendor. For example, the Azure services like Azure Machine Learning and Azure Stream are costly to use [20]. If an IoT device is not supported by Azure IoT Edge runtime, then it is not possible to use that as a sensing device in this framework. Moreover, the high end-to-end latency of Azure IoT Edge poses an issue for latency sensitive applications as discussed in [20].

Similarly, Huawei Intelligent EdgeFabric (IEF) allows IoT applications to be deployed on edge nodes as containers [21]. IoT applications are packaged using the Software Repository for Container (SWR) on Huawei Cloud leveraging the edge application template created on IEF for deployment to edge nodes. IEF is based on the open-source KubeEdge software built upon Kubernetes which aims to provide orchestration infrastructure that supports robust collaboration between cloud and edge. Similar to Azure IoT Edge, IEF requires a fee-based subscription and the developer is burdened by the cost of invoking supported services and the lack of SDKs for the integration with new IoT devices. IEF is not designed to faciliate the ease of composing IoT applications. Currently IEF is only available on the Linux systems.

ThingWorx is a commercial IoT platform designed for the industrial Internet of Things which highlights a fast and robust two-way communication between the industrial Things and the cloud server [22]. ThingWorx platform consists of four layers: device, client, platform, and database layers. The device layer consists of things or IoT devices that send data to the platform layer. The client layer contains the web-based interface that users can use to access the ThingWorx platform. The client offers several tools for building IoT applications including the Composer and the Mashup Builder. While the Composer helps the designer to maintain a uniform modeling environment, the Mashup Builder offers a dashboard for building new functions through common components, for example, buttons, lists, and gauges. The platform layer is where ThingWorx foundation resides, which serves as the hub of the ThingWorx's runtime environment and provides common services as well as monitoring the behavior of remote devices. The database layer provides the persistence services for ThingWrox's runtime data. Similarly, ThingWorx leverages Docker to deploy IoT applications as containerized applications on edge nodes. Same as Azure IoT Edge, ThingWorx is only available in Windows and Linux systems and is targeted for industrial IoT.

KubeEdge is the most well known open source container based edge computing framework [23]. It is part of the recent new generation of Cloud-to-Edge infrastructure that is known for its ability to scale out and the support for security and fault tolerance. KubEdge leverages container technology to bring native cloud capability to the edge. It consists of a cloud part and an edge part. While the cloud part interfaces with Kubernetes API and takes care of node management, the edge part has control of container deployment on the edge and provides an infrastructure for storage as well as event-based communication based on MQTT [24]. However, it has been

[5]https://aws.amazon.com/iot
[6]https://www.ibm.com/internet-of-things
[7]https://thingspeak.com/
[8]https://cloud.google.com/solutions/iot

reported in [23] that Kubernetes deployment leads to several performance and stability problems on some low memory edge devices (e.g., Raspberry Pi 3).

In summary, all the aforementioned container-based frameworks are not true IoT Edge Computing frameworks where IoT applications can be composed and shared and run independent of a cloud. The container-based approach will incur high computation resources at the edge. For example, an image of one Docker container is at least 500 MB. Docker container requires a high-end edge device that must have at least 5 to 6 GB RAM and multiple cores. Container-based IoT computing frameworks mainly address the efficient coordination across large number of edge nodes with the cloud. They do not address the ease of integration with heterogeneous IoT devices and the privacy concern of IoT applications. Cordova host is a cross platform IoT Edge Computing framework which allows sharing of plugins and accessors among the community of developers and composition of custom IoT applications using accessors as the building block. Accessors are based on the light-weight Javascript programming model. Acessor design pattern is designed to address the heterogeneity of IoT devices without dictating a particular standard or limiting the use of only certain approved devices.

The last framework that we investigated is the actor based framework from the Accessor project in Terraswarm Research Center in Berkeley[9]. The advantage of an actor-based framework is that it is light-weight and portable for capability and energy constrained IoT devices. The actor-based framework (accessor host) was first presented in the paper entitled "A Vision of Swarmlets" by Latronico et al. [15]. As stated in the accessor homepage[10]:

"Accessors are a technology for making the Internet of Things accessible to a broader community of citizens, inventors, and service providers through open interfaces, an open community of developers, and an open repository of technology. Developed by the TerraSwarm Research Center, accessors enable composing heterogeneous devices and services in the Internet of Things (IoT)".

An accessor is designed with the actor model of computation that embraces concurrency, atomicity and asynchrony. In other words, an accessor can be viewed as an actor that wraps a sensor, an actuator, or a service and hide the different implementations from developers. An *accessor host* is a service or application running on the client platform that can provide execution environment for accessors. The client platform can be a server (e.g., a high end desktop computer), a gateway (e.g., smartphone) or an edge device (e.g., a wearable device). In the context of the Terraswarm project, accessor host is also known as Swarmlet host.

In the iCyPhy (industrial Cyber-Physical Systems) project[11], a sequel to the Terraswarm's accessor project, a semantic accessor framework is proposed [25]. This framework attempts to combine Semantic Web with accessor technologies to create a platform that can dynamically discover and instantiate context-relevant accessors for dynamic real-time service provisioning such as the connected cars applications.

Our final choice of using the actor-based framework with accessor and accessor host came from the fact that it is light-weight and gives us the flexibility to use IoT devices from multiple vendors without dictating a specific standard. Moreover, an accessor host can be deployed in any of the three layer architecture of an IoT ecosystem. We describe the functionalities of accessors and accessor hosts in more details in Section III. Note that the sole accessor host from the Terraswarm project that can be deployed on edge devices is a conceptual and experimental Cordova host with no practical demonstration as a viable edge computing framework.

III. Background: Accessor and Accessor Host

An IoT middleware/computing framework typically exhibits a three-layer architecture (i.e., edge, gateway, cloud) [1]. IoT Middleware usually refers to a software system designed to be the intermediary between physical IoT devices and IoT applications. Terraswarm's accessor project focuses on the open, plug and play component architecture [26] for an IoT middleware. A unique feature of that IoT framework is a lightweight host endowed with standardized capabilities for running and deploying IoT services in any layer of a three-layer IoT architecture. The main concept in this framework for seamless interaction with a physical IoT device is *accessor*.

A. Accessor

Accessors provide the abstraction for smart "Things" across different hardware or software platforms to interact, bridging the heterogeneity among IoT systems and allowing for smarter interactions, sharing and portability. Accessors are implemented using lightweight JavaScript programming model, which is ubiquitous and allows things to communicate and share information in a message-oriented fashion. Figure 1 shows an accessor pattern taken from [15]. The horizontal contract governs interactions among accessors using ports and the vertical contract governs the asynchronous interaction with physical devices on the edge. As shown in Figure 1, the blue box (lower box) represents the proxy of an IoT device or a service on the cloud. An accessor can send a request to and receive a response from the blue box. An Asynchronous Atomic Callbacks (AAC) protocol is used for sending the request. AAC is a non-blocking protocol and enables many concurrent pending requests to be active at once without having the overhead of managing threads. Moreover, AAC invocation is atomic and thus more robust as contrasted to interrupt-driven threads or RPC. An AAC call does not use locks and thus cannot deadlock. The accessor isolates the low-level device specific communication protocol from the high level IoT application.

An accessor is defined by an interface with a number of input and output ports for managing and processing the data transfer between "Things". Ports provide a common paradigm of communication independent from the low level device

[9]https://ptolemy.berkeley.edu/projects/terraswarm/accessors/
[10]https://www.terraswarm.org/accessors
[11]https://github.comiCyPhy/accessor

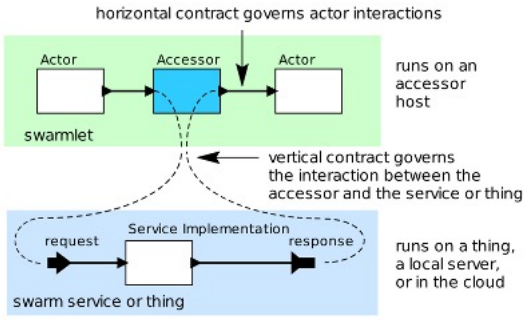


Fig. 1: Accessor Design Pattern from [15]

communication protocols. These ports also connect accessors together to provide a complex service as well as to enable the ease in the change, deletion or addition of capabilities to the service. The JavaScript programming model of accessors also enables accessors to essentially act like web pages on a browser, exchanging information with a variety of other services compliant with the vertical contract as shown in Figure 1.

B. Accessor Host

An Accessor host functions as a middleware which instantiates and executes accessors, similar to Java Virtual Machine (JVM) providing the uniform execution environment for Java programs in different operating systems. All existing accessor hosts share the functionalities defined in the common host[12] so as to provide a standardized set of capabilities for the execution of accessors in different environments. The following are examples of some action functions provided by the common host:

- `require`(): load the required library for the accessor to function correctly.
- `setParameter`(): set a parameter with a reference name and value for the ports.
- `initialize`(): initialize the accessor by the host on startup,
- `fire`(): perform a set action upon receiving the appropriate input signal,
- `setup`(): provide information to establish the names and initial values of all input and output ports,
- `send`(): send an output to another port,
- `setTimeout`(): set the specified function to execute at certain time,
- `setInterval`(): set the specified function to execute after certain time interval, and repeat at interval,
- `wrapup`(): release used resources and terminate the accessor.

The following is an example of a Hello World accessor:

```
exports.setup = function () {
  this.input('name');
  this.output('greeting');
};
exports.initialize = function () {
  this.addInputHandler('name',
   function () {
     this.send('greeting',
     'Hello World,' +this.get('name'));
  });
};
```

This accessor takes a string value in its input port "name", outputs a "Hello World" concatenated with the input name, and sends the result to its output port named "greeting". The function "send" is triggered with the arrival of input events.

The CapeCode host is the most established accessor host[13] but only runs on high end desktop computers. It is built on top of Ptolemy II[14], which is an open actor-oriented software development platform for the modeling, simulating, and designing of concurrent, real-time, embedded devices (a.k.a IoT devices). The central modeling concept in Ptolemy II is an actor, which is a software component that executes concurrently and communicates through messages sent via interconnected ports. CapeCode host extends the Ptolemy's actor-oriented framework with an accessor abstraction. In this context, an accessor encapsulates an IoT device and exposes it as an actor in CapeCode. The interface of an accessor serves as the local proxy for the physical IoT device or other remote services. The IoT device vendor provides the device specific implementation of communication and connectivity. CapeCode host leverages Ptolemy II's graphical-based interface for creating and composing accessors and uses Java's Nashorn V8 Script engine for executing the accessors.

Other accessor hosts are Browser, Node and Cordova. A Browser host supports executing accessors within the web browser. Accessors are instantiated as HTML pages in a Browser host. A Node host is basically a Node.js engine with support for the common host's capabilities. A Cordova host is an extension of Apache Cordova's cross mobile program development platform that is used for building applications using HTML, CSS and JavaScript[15] in one code base and targeted to multiple platforms such as Android, IOS, and Browser with no additional programming. The JavaScript interface of the accessor in Cordova interacts with native languages and APIs of physical or virtual IoT devices through a number of plugins. In essence, the plugin hides the various native code implementations behind a common JavaScript interface.

C. Why Cordova Host?

Although there exist many accessor hosts implemented by the TerraSwarm project team, only Cordova host, Node host, and Browser host have been shown to run on edge devices. The Browser host is light weight and ubiquitous. It can execute accessors using a web browser's native JavaScript engine. However, the Browser host places a lot of limitation on interaction with local hardware and has many security restrictions which restricted the type of accessors that could

[12]https://wiki.eecs.berkeley.edu/accessors/Main/CommonHost

[13]https://ptolemy.berkeley.edu/projects/terraswarm/accessors/.

[14]http://ptolemy.eecs.berkeley.edu

[15]https://cordova.apache.org/.

be implemented. Currently, it can only interacts with IoT devices that have RESTful interfaces. The Node.js host is light weight and has an efficient run-time environment for accessors. However, it has the same problem as the Browser host in having limited interaction with local hardware.

The experimental Cordova host from Terraswarm team has not been used for creating any real-world IoT applications. The version that we downloaded only has a simple TestOnce accessor which sends a single input to a HelloWorld accessor. A very limited working accessor modules in Capecode is available for Cordova host.

On the Cordova Accessor host webpage, Moreover, there is no study that demonstrates that this framework can handle large streaming data from a real world IoT device and serves as an efficient edge computing framework executing and coordinating among multiple accessors. To demonstrate the practicalities and advantages of Cordova host as an edge IoT computing framework, we refactored two monolithic native mobile Apps: 1) Fall Detection and 2) Heart Rate Monitoring into composition of accessors. We analyzed the reusability of accessors, the barrier of programming and deployment for consumers and the power and memory consumption of IoT services running on a Cordova host.

IV. IoT Service Development with Accessor

A. Architecture of Cordova Host

The Cordova Host combines the functionality of the Common Host with the open-source mobile development framework, Cordova by Apache. This framework, which itself is built upon Node.js, utilizes standard Web technologies such as HTML, CSS, and JavaScript for cross-platform application development. Notably, Cordova achieves its cross-platform deployment through the use of platform-targeted wrappers in addition to bindings to native implementations. These bindings, which are commonly referred to as *plugins*, provide a means for developers to access features provided by the platform-specific native codes through a JavaScript interface.

The Cordova Host architecture, which can be seen in Figure 2, primarily consists of both a web view and a Cordova Plugin component. Within the web view's JS directory, there are three separate sub-directories: i) one contains the implementation for the Common Host (provides the foundational functionalities of an accessor host), ii) one contains the implementation for accessor host specific to Cordova, and iii) one houses all the community-developed accessors that can be shared. Those three sub-directories make up the implementation of Cordova Accessor Host. The swarmlet.js is used to programmatically compose a pipeline of accessors to perform a specific task (a.k.a an IoT service). It is analogous to an application launching function such as the Java's main method.

Cordova plugins provide us with a means of utilizing native codes within web view wrapped applications. This is made possible by the plugin's Javascript interfaces. These interfaces consist of function calls that mimic the same behavior to that of the corresponding native methods. These Javascript function calls can be accessed globally throughout the entire project, which includes any specific platforms. An example of this is how a data collection accessor can directly access a Huawei brand smart watch's specific sensor data by utilizing the watch's native WearOS's API accessable via the WearOS plugin. The data returned by the plugin can then be accessed through the accessor's output port by its connected downstream accessor, which could then use another plugin distinctly designed to perform application-specific data analytics on the collected data. This implies that given the native connectivity protocols available via plugins, the data collection accessor has the potential to facilitate the collection of data from a plethora of sensors from a variety of IoT devices with minimal change in the design of the accessor.

B. Cordova Accessor-Module

Cordova Plugin is a very important part of building a functional accessor on the Cordova Host. Apache Cordova project comes with a set of core plugins. These core plugins provide functionalities to access common device capabilities such as battery, camera, contacts, Emails, and SMS. In addition to the core plugins, custom plugins (e.g., HTTP, web sockets) have been developed for communicating with an ever increasing set of IoT devices or services. Plugins require quite a bit of scaffolding. Apache Cordova provides a tool called Plugman to ease the development of plugins. The tool will create the required directory structure and the default configuration files.

It is highly recommended that when designing a Cordova plugin, one should do so with a more general use in mind as it pertains to functionality. This allows for the plugin to be used to develop a multitude of application specific functionalities by arranging and accessing the application specific plugins within certain modules. For example, we could have a data prediction module developed with fusion of both camera data and watch acceleration data. This module would have access to two plugins (i.e., camera, WearOS).

We used the Accessor-Module-Plugin design pattern originally proposed by the Accessor project [27] to structure our IoT applications. Figure 3 depicts an example Accessor-Module-Plugin pattern to achieve the flexibility, reuse, and encapsulation of device specific native codes for IoT application development.

The concept of module has been around for a long time in block structured programming languages to provide encapsulation of a set of related functionalities or sub programs. The goal of modularization in this framework is to enable ease of re-using of existing plugins and to reduce the code density of assessors. The Accessor modules provide accessors with the means to execute cordova host's modules within the accessor's web view contained environment. Modularization promotes the ability to rapidly develop IoT applications by allowing developers to create application specific functionalities through the augmentation of pre-existing open-source Cordova plugins which tend to be very generic. For example, by creating an Arduino-sensor module, we can re-use the Cordova-BLE plugin provided by a third-party, Evothings, to collect Arduino sensor data over bluetooth low energy without modifying the plugin. Through modularization, development time was

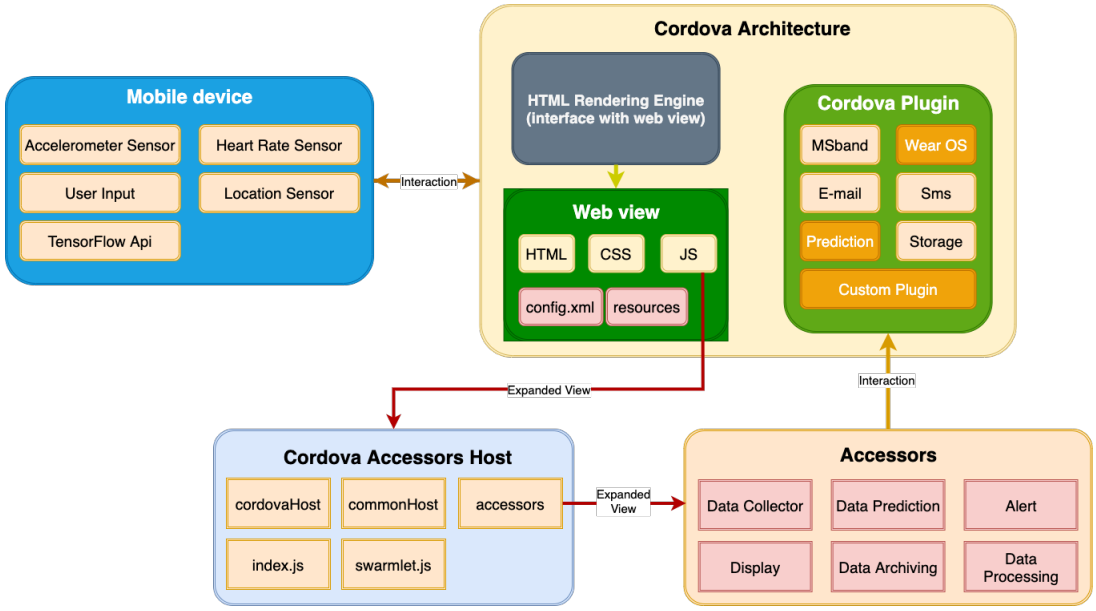


Fig. 2: The architecture of the Cordova Host

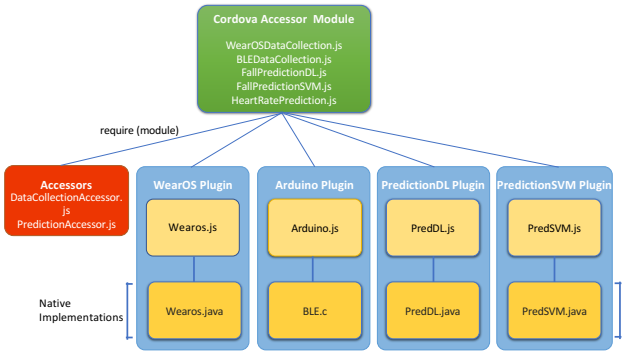


Fig. 3: Accessor-Module-Plugin Design Pattern for Cordova

reduced and the new capability was seamlessly integrated into the accessor without increasing the code density/complexity of accessor making it easy to re-use, as demonstrated in our example of collecting sensor data from either Ardunio or WearOS sensors using the same data collection assessor.

In the design of an accessor, the first statement should be to load the required module for accessing the desired devices via plugin's binding to the native codes knowing the context. The following present some pseudo codes for collecting data from the WearOS watch:

```
...
var context = discoveryContext("WearOS");
var deviceType = require(context);
exports.setup = function() {
  deviceType.initialize(
    function(result) {},
    function(){alert("not Initialized");});};}
exports.initialize = function() {
 var self = this;
 function getSensorData (fn)
    {deviceType.subscribe(...);})}

getSensorData(function(result) {
     self.send('dataOut', result);});};
```

The function "discoveryContext" will return the module that needs to be downloaded to connect to that device. A module groups a set of functionalities for a specific task for ease of discovery. The module to be loaded for WearOS's watch is the "WearOSDataCollection". A module can make use of one or more plugins. The "subscribe()" function in the module will bind to the native codes for reading sensor data via WearOS's plugin in this example. Each plugin is a wrapper for exactly one native implementation. The above design pattern enables sensing data from heterogeneous devices and processing the sensed data using a variety of algorithms with minimal change in the design of the accessors across different devices. New plugin to different native implementation can be added without having to modify the data collection accessor codes. Accessor developers are thus shielded from the low level details of the native implementation. The prediction accessor is designed to bind to different predicting modules using the same design pattern. Each module specializes in a particular prediction task, for example, fall verses blood glucose prediction. For each specific task, developers can choose to use a specific native algorithm via the module.

This Accessor-Module-Plugin design pattern enables composition of IoT applications by reusing accessors and dynamically bind to the required module. The number of plugins and modules for sensing data from different devices can continuously be added without having to modify the original

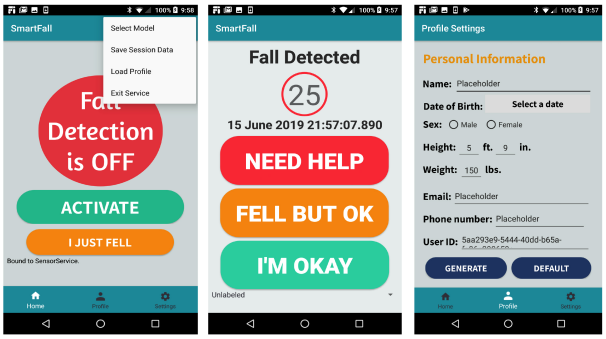


Fig. 4: Fall Detection App User Interface.

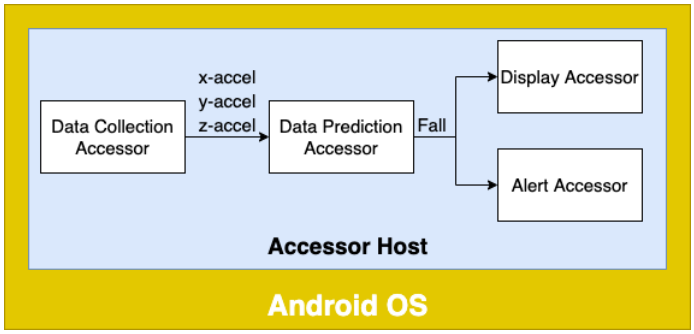


Fig. 5: Composition of Accessors for the Fall Detection App

data collection accessor.

In summary, while the concept of "Accessor-Module-Plugin" is originally proposed by the Terraswarm project in Berkeley, no methodology is provided on how to use it as an effective design pattern. In particular, there are no details on how to design an IoT application into reusable accessors, modules, and plugins in Cordova Accssor host. If a developer installs the experimental Cordova host, a limited number of modules just appear as sub directories. We provide the concrete pattern of Accessor-Module-Plugin in Figure 3 for developers to learn the art of building composable IoT applications in Cordova host using this specific pattern. We also provide the pseudo codes associated with the design pattern.

C. Accessors for Fall Detection

The Fall Detection App senses the streaming accelerometer data from a commodity-based smartwatch and applies a deep learning algorithm over the streaming data to predict falls. The smartwatch is paired with a smart phone that has the fall detection application installed in it. The application on the smartphone performs the necessary computations required for a fall prediction in real time with little or no latency. Figure 4 shows the main user interfaces of the Fall Detection App.

The screen on the left shows the home screen's UI for the application and the second screen shows the UI when a fall is detected. The home screen (leftmost screen in Figure 4) launches the App when the user presses the "ACTIVATE" button. The user must set up a profile and load the profile before the App can be activated.

When a fall is detected, the second screen of Figure 4 pops up on the smartphone, an audible sound is generated, and a timer of 30 seconds is initiated. The user is shown three buttons for interaction. The "NEED HELP" button will send a text message to the caregiver and also save and label the sensed data samples as true positives. The "FELL BUT OK" button will save the sensed data during that prediction interval as a true positive without notifying the caregiver. The "I'M OKAY" button will save these data as false positives. If a fall is detected and the user does not interact with any of these three buttons, after the timer expires, the system assumes that the user might be hurt or unconscious and an alert message is generated and sent to the caregiver automatically. The third UI screen is for the one time initialization of the user profile before the application can be launched. This UI includes setting up the contact details of the caregiver. Note that minimal personal data are collected and all the data are stored locally in the phone.

We refactored the Java implementation of Fall Detection App into four accessors as shown in Figure 5 such that each accessor performs a particular function and interacts with other accessors in the pipeline via message passing:

- *Data Collection Accessor*. When the App is activated, the Data Collection Accessor is triggered to collect accelerometer sensor data from the smartwatch in a set interval/sampling period and send the data as output to the Data Prediction Accessor. The phone and watch must first be paired.
- *Data Prediction Accessor*. This accessor takes a sequence/stream of accelerometer sensor data as input and predicts fall or not fall as output, which is then passed to the Display Accessor. It predicts fall by using a pre-trained deep learning RNN model that we developed in [5], [28], [29].
- *Alert Accessor*. This accessor receives the prediction status as input and sends either an E-mail or SMS to a registered recipient (care giver) if a fall is detected.
- *Display Accessor*. This accessor is responsible for displaying the output and the various options for the user to respond when a fall is detected.

D. Accessors for Heart Rate Monitoring

The Heart Rate Monitoring App utilizes the heart rate data collected from a smartwatch (IoT) device and a threshold algorithm to detect if there is an unusual high heart beat per minute (bpm) given the current context of the user and alert the user. Figure 6 shows a composition of accessors that are made up the Heart Rate Monitoring App:

- *Data Collection Accessor*. The Heart Rate Monitoring App begins with the Data Collection accessor, which senses heart rate data (bpm) from the smartwatch and sends the data as output to the Data Processing Accessor.
- *Data Processing Accessor*. This accessor takes the heart beats per minutes (bpm) data as input and gives an output of status of high bpm, which is passed to the Display Accessor. It performs a simple threshold algorithm to determine high bpm. The threshold algorithm currently

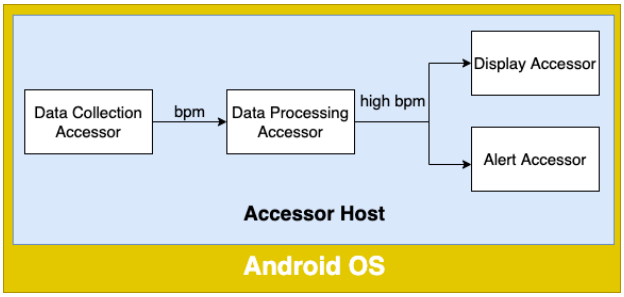


Fig. 6: Composition of Accessors for the Heart Rate Monitoring App

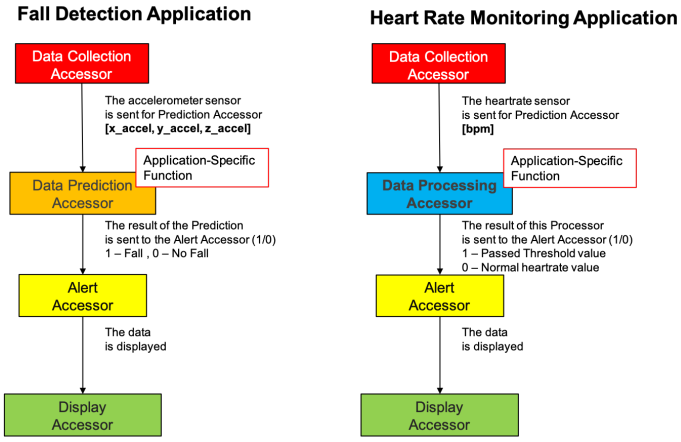


Fig. 7: Reusability of accessors for developing Heart Rate Monitoring App

is set to a simple conditional statement. This simple data processing task does not require access to legacy/native codes via plugin.

- *Alert Accessor*. This accessor receives the status as input and sends either E-mail or text message to a registered recipient if a high bpm is detected.
- *Display Accessor*. This accessor is responsible for displaying the heart rate information to the user as well as the status of monitoring.

E. Reusability of Accessors

The development of the assessor-based Fall Detection App includes four major accessors: the Data Collection, Data Prediction, Display and the Alert as shown on the left side of Figure 7. On the right side of Figure 7 are the accessors for the Heart Rate Monitoring App. Most of the accessors used in the Fall Detection App can be reused when building the Heart Rate Monitoring App. The only accessor that cannot be reused completely is the application specific accessor which is the second accessor,denoted in different colors in the left and right part of the Figure 7 (i.e., the Data Prediction Accessor and Data Processing Accessor). This is because the intention of heart rate monitoring is to send alert on detection of high or low heart rate. The application is not intended to predict heart rate.

If a more complex heart rate monitoring application is desired, such as using a sequence of heart rate data points

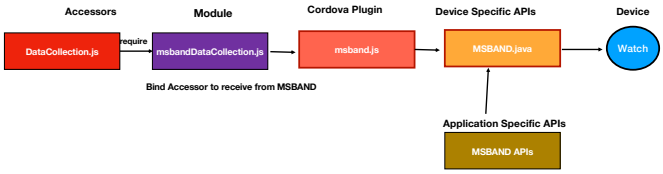


Fig. 8: Association of accessors with module and Cordova plugin for device portability

(time series) in the past to predict the heart rate in the future, then the data processing accessor can be exchanged with the data prediction accessor loaded with the HeartRatePrediction module. The actual prediction is accomplished by the native implementation available via either PredDL or PredSVM plugins given the input data and a pre-trained heart rate prediction model available via the HeartRatePrediction module.

On the other hand, the Data Collection accessor can be reused since its main goal is to retrieve sensor data from an IoT device (Smartwatch in this case). The only difference between the two apps is in the type of sensor data that needs to be collected from the smartwatch. They both use the watch for sensing the data and require the same module for connection. Only one line of code needs to be changed to reuse the data collection accessor for the Heart Rate Monitoring App as shown in the following JavaScript code snippet (line 3):

```
..
1 var sensors=["HEARTRATE","ACCELEROMETER"];
2 // set sensor to Heart rate
3 var sensor_type=sensors[1];
...
```

The Alert accessor for the Fall Detection App can be completely reused without any change in code since it includes the same functions responsible for alerting a caregiver as in the case of the Heart Rate Monitoring App when the heart rate goes beyond a defined threshold. The Display accessor for the Fall Detection App can also be reused with minor modification in building the Heart Rate Monitoring App. This is because both apps present the user with a list of options for interaction albeit the specific text in the options might be different.

The change in code for reusability is a single line of JavaScript. This implies that we can easily build a graphical user interface for non programmers (health professionals or consumers) to compose their IoT applications using Cordova Accessor Host.

F. Portability of Accessors

1) Across different vendors of watches:

The portability of services when using Cordova host refers to the support provided by the host for different devices and how easy it is to port applications from one device to another. As previously discussed, the only accessor that interacts with IoT devices is the Data Collection accessor. We demonstrate in this section how easy it is to switch between devices. Different smartphones (Huawei Mate 9 and Google Nexus 5)

and different smartwatches (Huawei Watch 2, TicWatch Pro and Microsoft Band 2) were used and tested for portability.

Figure 8 shows how the Data Collection accessor communicates with Microsoft Band watch using the MSBAND SDK libraries via MSBand module and plugin. The MSBAND plugin in the figure associates the accessor with the MSBAND SDK via the MSBAND.java program. To port the IoT service from one device vendor to another, in this case from Microsoft watch to Huawei watch, we only need to make one line of code change (line 1) to the Data Collection accessor as shown below. Line 1 is for adding the correct context to download the correct module. In this case the discoveryContext will figure out that Huawei's watch requires the WearOSDataCollection module.

```
...
1 var context=discoveryContext("WearOS");
2 var deviceType=require(context);
...
```

The Cordova plugin for WearOS must first be implemented or downloaded from the Cordova open-source Plugin Database. The development of plugins follows a specific design pattern and it only needs to be done once. The module is a grouping of a set of distinct functions in a plugin that is relevant to an application or task (i.e., data sensing).

2) Across different IoT device types:

Both Fall Detection and Heart Rate Monitoring applications run on smartwatches and smartphones. However, to exhibit Codova Accessor Host's flexibility as an edge IoT computing framework, we want to verify whether the same Data Collection accessor can be reused for sensing data on none-Android based IoT devices such as Adruino, which is a popular micro-controller based device. This device can communicate with the smartphone via Bluetooth Low Energy (BLE). However, the micro-controller such as Adrunio does not come with device specific APIs. To collect sensor data from sensors mounted on Adrunio board and display those data on a smartphone or smartwatch, we need a BLE's plugin. Apache Cordova has a BLE plugin developed by Evothings[16]. By reusing Evothings' BLE plugin and creating a BLE module that specializes in data collection, we demonstrated that the same Data Collection accessor can be used to sense data from an ultrasonic sensor or an infrared temperature sensor mounted on Arduino MKR1010 WiFi Board. Line 1, 3 and 4 are the only lines of codes that must be changed.

```
1 var context=discoveryContext("Adrunio");
2 var deviceType = require(context);
3 var sensors=["HEARTRATE","ACCELEROMETER",
"UNTRASONIC","INFRARED"]
 // set sensor to Untrasonic
4 var sensor_type = sensors[2];
...
```

3) Cross-platform deployment:

To demonstrate the cross-platform capability of Cordova Accessor Host, we deploy the Adrunio's Ultrasonic sensor application and the Heart Rate Monitoring App to iOS plaform by just using the following simple command:

```
cordova platform add ios
```

Cordova will automatically structure the iOS version of the application and utilize the BLE plugin that Evothings has created for iOS. The Heart Rate Monitoring App does not require any plugin. These two applications are deployed to iOS without any modification to the code base targeted for an Android platform.

G. Discussion

The reusability of accessors enables the accessor-based IoT applications to be adapted or re-purposed easily. For example, the accelerometer data collected can be used to analyze the gait of an individual to determine the early onset of Alzheimer. In this case, we can adapt the Fall Detection App by changing the module required for the Prediction accessor to load, i.e., a module that specializes in predicting signs for Alzheimer. Furthermore, if any of the aforementioned accessor-based IoT applications finds a need to archive the sensor data for long term storage or tracking, a Data Archiving accessor can be developed and added as an additional accessor to the overall composition. This will involve development of a plugin for a storage medium such as a database. Currently, SQLite plugin can be downloaded from Cordova plugin database for that purpose. The ease of changing or adding accessors demonstrates that accessor-base IoT applications can be updated more efficiently than current monolithic IoT applications, by replacing outdated accessors independently, without replacing or updating the entire application.

Our proposed Accessor-Module-Plugin design pattern enables streamlined addition of new IoT devices. So long as the device's plugin is available, it is a matter of simply changing a few lines of code to connect and collect data from a new type of device.

In summary, the logical requirement for many IoT applications are similar. Having to develop each separate application with a dedicated set of resources from the ground up increases the cost and the time for the development. An edge-based IoT computing framework like Cordova host can serve as a bridge across a variety of IoT devices and services to accelerate the delivery of IoT services.

V. Performance Evaluation

An edge-based IoT computing framework must be energy and computation efficient since it is deployed on power constrained IoT devices. In this section, we evaluate the performance of Cordova host by comparing the battery power consumption and memory usage of running an accessor-based Fall Detection App versus a native Java-based Fall Detection App on Android. The two versions of Fall Detection App are running on a TicWatch Pro smartwatch paired with Huawei Mate 9 smartphone running Android OS (version 8.0). The memory consumed by the app is recorded using Android's profiler tools.

[16]https://evothings.com/cordova-ble-plugin-updated/

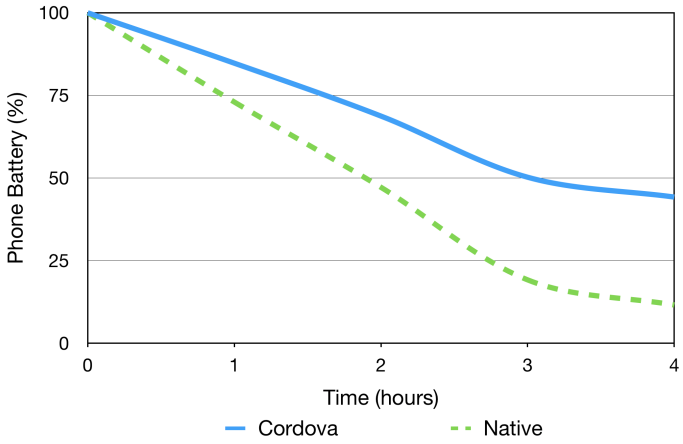


Fig. 9: SmartPhone battery usage of accessor-based App versus native App

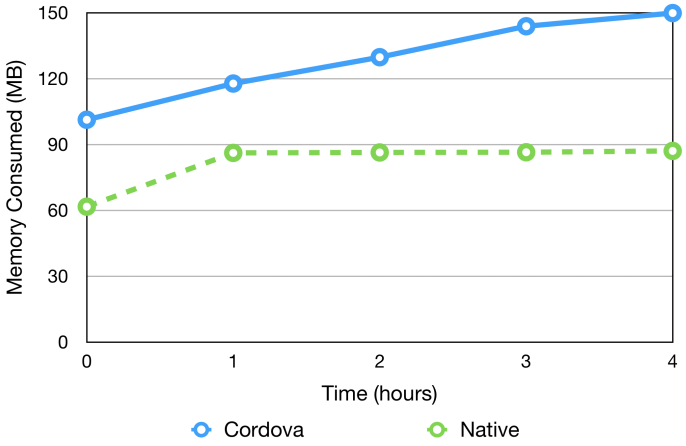


Fig. 10: Memory usage of the accessor-based App versus native App

A. Experimental Setup

Both the phone and watch batteries are fully charged and each application is made to run continuously for four hours for each test. The test is carried out by wearing the watch and activating the Fall Detection App on the phone while carrying out daily activities and recording the battery percentage at various intervals. Five tests are run for each version of the Fall Detection App. The battery percentage of both the smartphone and the smartwatch is recorded at every hour from the start of the experiment having 100%. We report the average battery consumption percentage over the five tests for each application on the smartphone and the smartwatch, respectively.

We also evaluate the memory usage during the whole lifecycle of the applications running on the Cordova host. The memory usage is measured by the Android Profiler tools on Android Studio, which provide real-time data to help developers understand how their application makes use of the CPU, memory, network, and battery resources. The smartphone is connected to Android Studio, then the android app is executed and a session is activated on Android's Profiler tools. In this way, the real-time memory consumption can be monitored and recorded.

B. Results and Analysis

We run the first test for the accessor-based Fall Detection App deployed on the Cordova host and its corresponding test for the native Java Fall Detection App. We observe that the accessor-based version performs a lot better than the native Java version. For example, We notice that the native version has its smartphone's battery at 8% when the smartwatch battery becomes 0% while the accessor-based version has its smartphone's battery still at 35% when the smartwatch battery becomes 0%. We examine the code in the native version and realize that it uses some background services that might consume more battery as compared to the accessor-based application that does not use the background services. In order to compare them fairly, we remove the background services code in the native version that might account for the difference.

After the modification, we perform the second of the five tests and notice that there is a little improvement but the difference is not that significant. The remaining three tests show the similar trend. The average battery usage over the five tests at each one hour interval is recorded. Figure 9 shows the battery usage at each time interval.

The battery usage of the smartwatch is the same because in both experiments, the same service is run on the smartwatch to send the sensed data to the phone periodically. The accessor-based version running on the Cordova host used around 32.65% less battery power than the native version. The better battery utilization in accessor-based IoT applications is attributed to the lightweight JavaScript programming model and the accessor-module-plugin design pattern. The IoT application is decoupled from the device-specific communication code using this design pattern. Moreover, the communication is asynchronous as shown in Figure 3. The device-specific communication is invoked on demand as contrasted to continuous interaction in the native version using Java thread. More experiments with other IoT applications need to be conducted to fully verify the benefits of our design pattern.

In terms of memory consumption, we observe that the native Java version is more efficient than the accessor-based version. Figure 10 shows a visual representation of memory consumption at each time interval. The accessor-based version running on the Cordova host consumes about 40MB more memory than the native Jave version. The higher memory consumption in the accessor-based IoT applications is attributed to the usage of Threadpool inherent in the Cordova host's plugins. The advantage of using the Threadpool is to promote parallel execution. It introduces the possibility of reading sensor data from multiple sensors at once. Different functionalities of the App can be run in parallel and the results are then converged on the Main User Interface thread for display. The usage of Threadpool introduces an overhead in terms of memory usage, making it more memory intensive for simple IoT applications. However, for more complex applications that involve correlating multiple sensor data, Threadpool might be essential.

VI. Conclusion and Future Work

In this paper, we have discussed the need for an edge-base Internet of Things (IoT) computing framework and specifically explored the Cordova Accessor host. We presented the architecture of the Cordova Accessor host and analyzed the advantages of using it as an edge computing framework. Our analysis is based on the first-hand development of three real-world IoT applications using that framework. We demonstrated that the Cordova host supports the essential ingredients of composition, re-usability, and portability of IoT applications. We outlined a methodology to create IoT applications using the Accessor-Module-Plugin design pattern that provides access to the device's native codes in a modular and scalable fashion.

We showed in this paper that IoT applications deployed on Android compatible devices using this framework consume around 35% less battery power than the same IoT application deployed in native Java language. We also showed that accessor-based IoT application uses around 40MB more memory than native app due to the support of parallel execution of functions inherent in the Cordova host implementation. We further demonstrated that our design pattern can support sensing and collection of data from a non Android based device such as Arduino with just one line of code change in the Data Collection accessor codes and no change to the Cordova Accessor host beside adding the third party BLE's plugin.

The container-based edge computing framework that tied to various commercial cloud services is gaining traction. However, the portability of an edge container is not proven yet. Currently, there are no Docker compatible containers that can run on an edge device like Android phone. There are suggestions that Android's kernel can be modified to support containers. It is impractical to have to modify Android's kernel to support container's run-time because different Android devices are using different kernels and each modification and compilation is highly specific. One possible solution is to install an emulation layer on top of an Android OS and run the container on the emulator. However, the resulting container image would be very slow. Until a light weight container's run-time is released and Android kernel is modified to support container like Docker, container based approach is not practical. Moreover, performance will be an issue with running containers on the edge with reasonably priced phones that have only 4 GB RAM.

Designed to be cross platforms, the Cordova Accessor host is a light-weight IoT Edge Computing framework, where Accessor is coded solely in Javascript, which is a dynamically typed, interpreted language with rising popularity in our networked world. We have demonstrated that the accessor-based applications,the data collection accessor, can be deployed to heterogeneous IoT devices in this paper.

In the future, there is a need for more in-depth studies on other performance advantages of the Cordova host such as latency and reliability in data collection. There is also a need to evaluate the performance of the Cordova Accessor host in terms of energy, memory and portability compared with other similar edge-based computing frameworks such as the container-based approach.

References

[1] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Z. Sheng, "IoT Middleware: A Survey on Issues and Enabling Technologies," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 1–20, Feb 2017.

[2] N. K. Tran, Q. Z. Sheng, M. A. Babar, and L. Yao, "Searching the Web of Things: State of the Art, Challenges, and Solutions," *ACM Computing Surveys*, vol. 50, no. 4, pp. 55:1–55:34, 2017.

[3] R. Want, B. N. Schilit, and S. Jenson, "Enabling the Internet of Things," *IEEE Computer*, vol. 48, no. 1, pp. 28–35, 2015.

[4] L. Baresi, L. Mottola, and S. Dustdar, "Building Software for the Internet of Things," *IEEE Internet Computing*, vol. 19, no. 2, pp. 6–8, 2015.

[5] T. R. Mauldin, M. E. Canby, V. Metsis, A. H. Ngu, and C. C. Rivera, "Smartfall: A smartwatch-based fall detection system using deep learning," *Sensors*, vol. 18, no. 10, 2018.

[6] D. Raggett, "The Web of Things: Challenges and Opportunities," *IEEE Computer*, vol. 48, no. 5, pp. 26–32, May 2015.

[7] W. E. Zhang, Q. Z. Sheng, A. Mahmood, D. H. Tran, M. Zaib, S. A. Hamad, A. Aljubairy, A. A. F. Alhazmi, S. Sagar, and C. Ma, "The 10 research topics in the internet of things," in *6th IEEE International Conference on Collaboration and Internet Computing, CIC 2020*, Atlanta, GA, USA, December 1-3, 2020, 2020, pp. 34–43.

[8] L. Yao, Q. Z. Sheng, and S. Dustdar, "Web-based Management of the Internet of Things," *IEEE Internet Computing*, vol. 19, no. 4, pp. 60–67, July/August 2015.

[9] Y. Qin, Q. Z. Sheng, N. J. G. Falkner, S. Dustdar, H. Wang, and A. V. Vasilakos, "When Things Matter: A Survey on Data-Centric Internet of Things," *Journal of Network and Computer Applications*, vol. 64, pp. 137–153, 2016.

[10] K. Aberer, M. Hauswirth, and A. Salehi, "A middleware for fast and flexible sensor network deployment," in *VLDB '06*, 2006, pp. 1199–1202.

[11] R. T. Fielding, "Architectural styles and design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000, www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[12] R. Buyya and S. N. Srirama, *A Lightweight Container Middleware for Edge Cloud Architectures*, 2019, pp. 145–170.

[13] T. Zachariah, M. Klugman, B. Campbell, J. Adkins, N. Jackson, and P. Dutta, "The internet of things has a gateway problem," in *HotMobile '15*, February 2015.

[14] H. Wu, D. Sun, L. Peng, Y. Yao, J. Wu, Q. Z. Sheng, and Y. Yan, "Dynamic Edge Access System in IoT Environment," *IEEE Internet Things Journal*, vol. 7, no. 4, pp. 2509–2520, 2020.

[15] E. Latronico, E. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber, "A vision of swarmlets," *IEEE Internet Computing*, vol. 19, no. 2, pp. 20–28, Mar 2015.

[16] J. Tang, Z. Zhou, X. Xue, and G. Wang, "Using Collaborative Edge-Cloud Cache for Search in Internet of Things," *IEEE Internet Things Journal*, vol. 7, no. 2, pp. 922–936, 2020.

[17] H. Lee, E. Jeong, D. Kang, J. Kim, and S. Ha, "A Novel Service-Oriented Platform for the Internet of Things," in *Proc. of the 7th International Conference on the Internet of Things (IoT'17)*, 2017.

[18] I.-Y. Ko, H.-G. Ko, A. J. Molina, and J.-H. Kwon, "Soiot: Toward a user-centric iot-based service framework," *ACM Trans. Internet Technol.*, vol. 16, no. 2, Apr. 2016. [Online]. Available: https://doi-org.libproxy.txstate.edu/10.1145/2835492

[19] A. Das, S. Patterson, and M. Wittie, "Edgebench: Benchmarking edge computing platforms," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 175–180.

[20] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou, "A survey on edge computing systems and tools," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1537–1562, 2019.

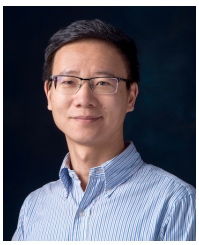
[21] R. Li, X. Liu, X. Zheng, C. Zhang, and H. Liu, "Tdd4fog: A test-driven software development platform for fog computing systems," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 673–676.

[22] M. Rahman and J. Gao, "A reusable automated acceptance testing architecture for microservices in behavior-driven development," in *2015 IEEE Symposium on Service-Oriented System Engineering*, 2015, pp. 321–325.

[23] T. Goethals, F. DeTurck, and B. Volckaert, "Extending kubernetes clusters to low-resource edge devices using virtual kubelets," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020.
[24] R. A. Light, "Mosquitto: server and client implementation of the mqtt protocol," *Journal of Open Source Software*, vol. 2, no. 13, p. 265, 2017.
[25] M. Weber, R. Akella, and E. A. Lee, "Service discovery for the connected car with semantic accessors," in *Proc. of the 2019 IEEE Intelligent Vehicles Symposium (IV)*, 2019, pp. 2417–2422.
[26] C. Brooks, C. Jerad, H. Kim, E. A. Lee, M. Lohstroh, V. Nouvelletz, B. Osyk, and M. Weber, "A component architecture for the internet of things," *Proceedings of the IEEE*, vol. 106, no. 9, pp. 1527–1542, Sep. 2018.
[27] *The Accessor Project*, 2013, https://ptolemy.berkeley.edu/projects/terraswarm/accessors/.
[28] T. Mauldin, A. H. Ngu, V. Metsis, and M. E. Canby, "Ensemble deep learning on wearables using small datasets," *ACM Transactions on Computing for Healthcare*, vol. 2, no. 1, pp. 1–31, December 2020.
[29] A. H. Ngu, P.-T. Tseng, M. Paliwal, C. Carpenter, and W. Stipe, "Smartwatch-based iot fall detection application," *Open Journal of Internet Of Things (OJIOT)*, vol. 4, no. 1, pp. 87–98, 2018, special Issue: Proceedings of the International Workshop on Very Large Internet of Things (VLIoT 2018) in conjunction with the VLDB 2018 Conference in Rio de Janeiro, Brazil. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:101:1-201808051930495128214

Anne H.H. Ngu, Ph.D. University of Western Australia, is full professor and program director of Computer Science Ph.D. program at Texas State University. From 1992-2000, she worked as a Senior Lecturer in the School of Computer Science and Engineering, University of New South Wales, Australia. She had also held the research scientist/scholar position with Telcordia Technologies; Lawrence Livermore National Laboratory, University of California, Berkeley; Commonwealth Scientific and Industrial Research Organization (CSIRO), Australia and the Tilburg University, The Netherlands. Dr. Ngu has published over 130 technical papers in journals and refereed conferences in computer science. Her main research interests are in service discovery and integration, Internet of Things, scientific workflows, smart health, and software engineering. Her professional service features key leadership roles in ICDE and WISE conferences. She was a winner of the 2013 NCWIT Undergraduate Research Mentoring Award.

Jesuloluwa S. Eyitayo graduated with a Masters degree in Computer Science from Texas State University, USA in 2020. He received his Bachelors Degree in Computer Engineering from Covenant University, Nigeria. Jesuloluwa is currently working as a Software Engineer at Intel Corporation. His research interests include in software engineering, machine learning, and smart health.



Guowei Yang is an Associate Professor in the Department of Computer Science at Texas State University, USA. He received the B.E. in Software Engineering from Harbin Institute of Technology, China, in 2004, the M.E. in Computer Software and Theory from Institute of Software Chinese Academy of Sciences, China, in 2007, the M.S. in Computer Science from University of Nebraska-Lincoln, USA, in 2009, and the Ph.D. in Electrical and Computer Engineering from the University of Texas at Austin, USA, in 2013. His research interests include software engineering, program analysis, and formal methods, with a focus on improving software reliability and dependability.

Colin Campbell is currently an Undergraduate Research Assistant at Texas State University, USA. He will be graduating with his B.S. in Computer Science from Texas State University in 2021. Colin's main research interests are software engineering, Internet of Things (IoT), and machine learning.

Quan Z. Sheng is a full Professor and Head of Department of Computing at Macquarie University, Sydney, Australia. His research interests include service oriented computing, distributed computing, Internet computing, and Internet of Things. Michael holds a PhD degree in computer science from the University of New South Wales (UNSW) and did his postdoc as a research scientist at CSIRO ICT Centre. He has more than 400 publications. Prof Michael Sheng is the recipient of AMiner Most Influential Scholar Award in IoT in 2019, ARC Future Fellowship in 2014, Chris Wallace Award for Outstanding Research Contribution in 2012, and Microsoft Fellowship in 2003. He is ranked by Microsoft Academic as one of the Most Impactful Authors in Services Computing (ranked the 4th all time).

Jianyuan Ni is currently a Ph.D. student in the Department of Computer Science at Texas State University, USA. He received his M.S. in Computer Science from Lamar University, USA, in 2020. His main research areas include computer vision and deep learning on activity recognition.