POLITECNICO DI MILANO

Dipartimento di Elettronica, Informazione e Bioingegneria

Laurea Magistrale-Ingegneria delle Telecomunicazioni

CLOUD PLATFORMS FOR THE INTERNET OF THINGS: HOW DO THEY STACK UP IN A REAL-WORLD APPLICATION?

Relatore: Prof. LUCA MOTTOLA

Tesi di laurea di:
KOUSTABH DOLUI
Matr. 836422

Anno Accademico 2016 - 2017

Abstract

The rapid growth and ubiquitous presence of the Internet of Things has laid the foundation for applications in various domains ranging from smart cities, transport systems and smart homes to healthcare, agriculture and smart grids. The data produced from these applications are diverse in nature with a diverse set of storage and processing requirements based on storage space, scope of the processed data and processing capacity required. These requirements have entailed a move away from storage and processing of data locally on end-devices to that on the cloud platforms for the Internet of Things. This paradigm shift of moving data storage and processing to the cloud has resulted in a steep increase in the number of cloud platforms. These cloud platforms are diverse in terms of the protocols they use, the application frameworks they offer, the security measures they implement and beyond. However, there is a lack of standard or framework to compare these cloud platforms, when selecting a platform, for building an Internet of Things application. Studies in existing literature compare platforms based on the generic requirements of Internet of Things applications and not based on any real world use-case scenario. The work in this thesis aims at bridging this gap in the existing literature.

The contribution of this thesis is divided into two parts. In the first part, we have studied more than 20 cloud services offering different kinds of services for the Internet of Things including Infrastructure-as-a-Service, Platform-as-a-Service and Software-as-a Service. We have defined a novel taxonomy to classify the features of cloud services offered based on the PaaS. The taxonomy is applicable not only to the cloud platforms that we have studied but to any cloud platform for the Internet of Things. We have discussed the taxonomy in detail to elaborate the diversity of the features of the cloud platforms and the tradeoffs involved while selecting a platform for a use case. Furthermore, we have selected 12 cloud services offering PaaS from the above cloud services and have studied each of them in detail based on the taxonomy and presented the trends that we have observed for these cloud platforms.

In the second part of the thesis, we have selected four of the above platforms to study in detail for a real deployment based on monitoring the environment and structural health of an ancient underground temple in Circo Massimo, Rome, as a part of a project in collaboration with archaeologists from the University of Trieste. We have discussed the challenges involved in building an end-to-end solution in a constrained environment such as this based on cloud platforms and the Internet of Things. For each of the cloud platforms, we have deployed a solution to store data from the site, process the data and visualize the data in accordance to the requirements of the archaeologists. We have studied the deployment for each cloud platform in detail and compared them based on their ease-of-use, documentation and the extent to which the cloud platforms fulfilled the requirements for the use-case.

Sommario

La rapida crescita e l'onnipresenza dell'Internet delle cose ha gettato le basi per applicazioni in diversi settori, dalle città intelligenti, ai sistemi di trasporto e case intelligenti, all'assistenza sanitaria, all'agricoltura. I dati prodotti da queste applicazioni sono di natura diversa con diverse esigenze di storage e di elaborazione. Tali requisiti hanno comportato un allontanamento dalla memorizzazione ed elaborazione locale, e uno spostamento sulle piattaforme cloud. Questo cambiamento di paradigma ha portato ad un forte aumento del numero di piattaforme cloud disponibili. Queste sono diverse in termini di protocolli che usano, i framework applicativi che offrono, le misure di sicurezza che implementano. Tuttavia, vi è una mancanza di standard o framework per confrontare tali piattaforme, quando si seleziona una piattaforma, per costruire un applicazione. Gli studi esistenti effettuano un confronto di piattaforme in base alle esigenze generiche di applicazioni e non sulla base di scenario di uso reale. Il lavoro in questa tesi si propone di colmare questa lacuna.

Il contributo di questa tesi è divisa in due parti. Nella prima parte, abbiamo studiato più di 20 servizi cloud che offrono diversi tipi di servizi per l'Internet delle cose tra cui Infrastructure-as-a-Service, Platform-as-a-Service e Software-as-a Service. Abbiamo definito un nuova tassonomia per classificare le caratteristiche dei servizi cloud offerti sulla base del PaaS. La tassonomia è applicabile non solo alle piattaforme cloud che abbiamo studiato, ma a qualsiasi piattaforma cloud per l'Internet delle cose. Abbiamo discusso la tassonomia in dettaglio per elaborare la diversità delle caratteristiche delle piattaforme cloud ei compromessi coinvolti mentre la selezione di una piattaforma per un caso d'uso. Inoltre, abbiamo selezionato 12 servizi cloud offrono PaaS dai servizi cloud di cui sopra e abbiamo studiato ciascuno di essi in dettaglio sulla base della tassonomia e ha presentato le tendenze che abbiamo osservato per queste piattaforme cloud.

Nella seconda parte della tesi, abbiamo selezionato quattro delle piattaforme sopra per studiare in dettaglio per una vera distribuzione basato sul monitoraggio dell'ambiente e della salute strutturale di un antico tempio sotterraneo a Circo Massimo, Roma, come parte di un progetto in collaborazione con archeologi dell'Università di Trieste. Abbiamo discusso le sfide coinvolti nella costruzione di una soluzione end-to-end in un ambiente vincolato come questo sulla base di piattaforme cloud e l'Internet delle cose. Per ciascuna delle piattaforme cloud, abbiamo implementato una soluzione per memorizzare i dati dal sito, elaborare i dati e visualizzare i dati in base alle esigenze degli archeologi. Abbiamo studiato la distribuzione di ogni piattaforma cloud in dettaglio e loro confronto in base alla loro facilità d'uso, la documentazione e la misura in cui le piattaforme cloud soddisfano i requisiti per il caso d'uso.

Acknowledgements

My first token of gratitude would go towards my parents, to whom I owe all my achievements. Without their support, this day would not have been possible. I would like to thank my thesis advisor, Luca, for believing in me and my abilities and for walking me through this learning process of conducting research work and coming up with this document. I have been fortunate to learn from the professors here at Politecnico di Milano and from the professors at Polytechnic University of Catalonia (UPC), Barcelona, during my stay for the Erasmus exchange program. I would like to thank Mike and Naveed for the collaborations on the deployment of the project at Circo Massimo, Rome and for their insights whenever I needed them. I extend my gratitude to the archaeologists from the University of Trieste, to let us be a part of this project and let us test the platforms at Circo Massimo. Finally, I would like to thank my colleagues here in Politecnico di Milano, Armand, Leila, Agnirudra, Amitesh, Veronica, Oscar, Emanuele and Gabriele and my colleagues at UPC, Lucia and Guillem for being there throughout this journey.

Contents

List of Figures

Listings

Chapter 1

Introduction

The advent of the Internet of Things has connected regular physical objects, we interact with every day, to the internet and has resulted in a global network of interconnected objects [14] [20]. With recent advances in technology, embedded devices have become smaller with an increase in the amount of storage and processing capacities, leading to their usage in a wide range of pervasive applications. Technologies for wireless communications have evolved as well to enable use of low power radio, energy efficient protocols and smaller antennas [28] to facilitate communication with regular objects. These have acted as contributing factors to the ubiquitous nature of the Internet of Things, expediting a rise of applications in multiple domains including healthcare, industries, building automation, smart home, smart city and beyond [16] [25] [18] [26].

However, the unprecedented growth of the Internet of Things has resulted in a steep increase in the amount of data they generate and the processing required to build meaningful applications from the data [15]. Different applications produce different kinds of data ranging from textual data to images and videos. Thus, the applications have diverse requirements in terms of the storage and processing required on the data [24]. For example, an application based on smart cities may require data collected from a large geographical area while an industrial application would require storage and processing of a massive scale of data produced from a single site. Hence, the storage and processing capacity of the 'things' are insufficient with regard to that required for the applications based on the Internet of Things in terms of storage capacity, processing capacity and the scope and context of the processed data [15] [24]. This has entailed a move away from local storage and processing of data on the devices to that on the cloud, leading to a growth of cloud platforms offering their services for building Internet of Things applications.

The architecture for Internet of Things applications depicted in Figure 1.1 involves data generated from the devices or 'things' to be gathered optionally on a gateway depending on the specific application and then being pushed to the cloud [17]. The gateway offers services like protocol translation, local data storage and processing and local validation when the 'things' are resource-constrained and not equipped to handle the complexity of these services [27]. The cloud platforms offer numerous services to build end-to-end applications for the Internet of Things. Firstly, cloud platforms offer services to gather the data from the end devices and the gateways leveraging various underlying protocols. Secondly, cloud platforms offer various processing modules to process the data with simple modules offering basic statistical processing to more complex ones

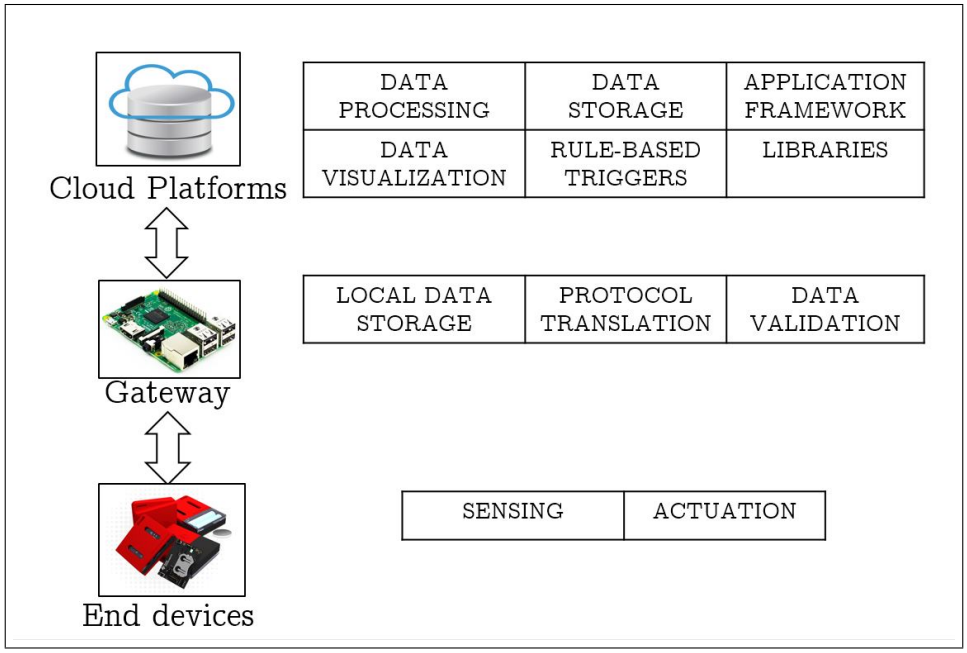

Figure 1.1: Architecture for the Internet of Things applications

like machine learning and pattern recognition. Thirdly, cloud platforms offer remote access to the stored data on the platform through various interfaces. Furthermore, some cloud platforms offer application frameworks to build applications on top of the platform leveraging the data stored on the platforms. Additionally, the cloud platforms offer services like rule-based triggers to perform actuations based on certain events captured from the data being processed, visualization of data stored on the platforms and libraries to access the services offered by the platform.

1.1 Contribution

There are plenty of cloud platforms available for designing and implementing solutions for the Internet of Things. These platforms are diverse on multiple accounts ranging from the type of services they offer, the protocols they use, the underlying interaction model and beyond. Thus, given a use case, the number of options for a user to choose from is quite large, while there is a dearth of a well-defined standard set of parameters based on which a user can select a platform given a use-case scenario. Previous studies conducted in existing literature offer an overview and a comparison among some of these cloud platforms based on their features and perform a gap analysis for the platforms [23]. On the other hand, authors of [19], presents a table defining features of each platform for multiple platforms and leaves it to the reader to choose the right platform based on the table. These studies do not offer a detailed description of these features, neither do they offer any insight into how the features affect the choice of a platform. The study in [22] provides an evaluation framework only considering the features from an application provider point of view. These features are only a subset of the features that the cloud platforms offer if we look at the complete architecture of an Internet of Things application. The authors of [21] present a study on current middle-ware solutions for the Internet of Things. They compare 4 cloud platforms based on a basic quick-start application for each

platform from a non-technical user's perspective. Thus, the comparisons in existing literature are based on a generic overview of the Internet of Things applications and their requirements and not on the requirements of real world use case.

In this thesis we have addressed these gaps in the existing literature. The contribution of this thesis is divided into two major parts.

- **Detailed study of cloud platforms:** In the first part, we have studied the features of various cloud platforms to present a taxonomy to classify the features of each platform. The taxonomy that we have defined is not limited to only the platforms we have studied but is generic and applicable to any cloud platform available. This taxonomy can be used to evaluate the features of any cloud service offered as Platform-as-a-Service(PaaS). Each of the terms defined in the taxonomy is elaborated with examples from the cloud platforms and the tradeoffs involving the features of the platforms based on these terms are discussed in detail. We have conducted a detailed study on 12 cloud platforms for the Internet of Things classifying their features based on the taxonomy.
- **Real-world deployment for use case:** In the second part of the thesis, we have discussed the challenges involved in a real-world application to monitor a heritage site in Rome, leveraging the Internet of Things architecture and the cloud platforms. Based on our study of the cloud platforms, we have selected 4 cloud platforms and implemented a solution based on each of these platforms. Unlike the generic comparisons of cloud platforms in the literature, we have stacked these cloud platforms against each other and compared their utility and ease-of-use based on this real use case.

1.2 Structure of thesis

Detailed study of cloud platforms: In the first part of the thesis, we conduct a qualitative study of the cloud platforms for the Internet of Things. In chapter 2, we provide an overview of the Internet of Things and discuss the state of the art for Internet of Things applications and cloud platforms. We present the architectural elements that comprise an Internet of Things application and describe each of the elements and their roles in detail. Furthermore, we discuss about the challenges involved in data storage and processing on the end devices and gateways and elucidate on how cloud platforms addresses these issues. We describe in further detail on the architecture of the Internet of Things applications and define two basic types of architectures for building cloud based solutions for use cases. Moreover, we study the importance of protocols in building an end-to-end Internet of Things application and classify these protocols based on their messaging model. We present some of the most widely used protocols of each type in detail.

In chapter 3, we discuss about the diversity of the cloud platforms and how the requirements from a cloud platform vary with different applications and architectures defined in chapter 2. In this chapter, we define a novel taxonomy which is used to classify the features of the cloud platforms. Each of these terms defined, are described in detail and examples of the different types of features corresponding to each term are presented. The features of the cloud platforms present a set of tradeoffs based on their diversity in different cloud platforms and the requirements of use cases. These tradeoffs are explained as well to provide the reader with a

clear understanding of the role of each of the terms defined in each cloud platform.

In chapter 4, we conduct an elaborative study on the cloud services available for Internet of Things applications. We select a subset of the cloud services offering Platform-as-a-Service (PaaS) and present their features in a tabular form, followed by a comprehensive analysis of their features based on the taxonomy defined in chapter 3. We discuss about some of the other services offered including Infrastructure-as-a-Service (IaaS) and Software-as-a-Service (SaaS) in brief along with a study of examples of some of these services. We highlight the trends we have observed in the cloud platforms studied above and present them in the concluding section of this chapter.

Real-world deployment for use case: In the second part of the thesis, we use the knowledge we have gathered from the first part and apply it to building an end-to-end solution for a real-world use case. In chapter 5, we select 4 cloud platforms among the platforms we studied in detail in chapter 4. The choice of the platforms is based on the diversity of the features of the platforms, such that we could study a wide range of features from these platforms. We discuss the rationale behind selecting these 4 platforms for our study. Furthermore, we comprehensively discuss our use-case scenario to monitor an underground temple in the heritage site of Circo Massimo, Rome, to study its environment and structural condition using a network of embedded devices. We also discuss the requirements from the deployment specified by the group of archaeologists we collaborated with, in terms of storage, processing and visualization of data.

The solution we developed for the use-case is presented in two parts. In the first part, we discuss about the implementation of the part of the architecture deployed at the site. We present the challenges involved in handling a deployment in a constrained environment like the one we worked, in terms of connectivity, device lifetime, architecture design and other unforeseen challenges. We discuss about our sensor network and gateway for the use-case and how we tried to address these challenges through our choice of architecture. In the second part, we deploy a working solution on each of the cloud platforms, to fulfill the requirements specified by the archaeologists. Furthermore, we compare the solutions we developed using each cloud platform, based on the documentation, ease-of-use of deploying solutions and how the platforms satisfied our requirements of the use case.

In chapter 6, we conclude our work by presenting our contribution in detail and presenting a scope for future work leveraging the contribution of this thesis.

Chapter 2

State of the art

The Internet of Things (IoT) is a novel concept connecting physical objects to the internet, changing the way we look at our surroundings and the way we are interacting with objects around us. Internet is moving from connecting people with people to connecting people with things and things with things as well. The Internet of Things is playing a major role in this transformation of the internet. The Internet of Things can be defined as a network of objects which have the following properties; these objects should be able to connect to the internet and these objects should be uniquely addressable in the internet. For example, the thermostat in the house can be such an object or 'thing' connected to the internet. The thermostat can be used to automatically control the temperature of the house based on sensors deployed inside the house along with the weather forecast fetched over the internet. With the simplicity, utility and pervasiveness that IoT has brought with it, IoT has found applications in various domains ranging from building automation, smart transport systems and industrial automation to healthcare, disaster management and emergency response services.

Interaction with a plethora of objects through the Internet of Things has brought about a rapid increase in the amount of data being generated and exchanged over the internet. The data generated is diverse in nature, ranging from textual data to images and videos which have different requirements for storage and processing. This diversity of requirements has been fulfilled by cloud platforms which allow storage and processing of the data and offers a medium to build IoT applications leveraging the data. In section 2.1, we discuss about the elements comprising a cloud platform based IoT application and how each of these elements contribute to the development of an IoT application. The types of architecture based on these elements are discussed in section 2.2. Furthermore, we discuss about the importance of protocols in the interaction of the devices with the cloud platforms and study some of the frequently used protocols in section 2.3.

2.1 Architectural elements in IoT

The architecture for the Internet of Things comprises of three distinct elements, namely end devices, gateway devices and the cloud platform. These elements play their individual roles in order to build the services offered by Internet of Things. The end devices lie in the lowest level of the architecture in terms of abstraction, with direct interaction with the environment. They are equipped with sensors or actuators or both to interact

with the real world. These devices are connected to one or multiple gateway devices which act as a bridge between the end devices and the cloud platform. The gateway offers local storage and processing of the data gathered from the end devices and also acts as a management interface for the end devices. The gateways communicate with both the devices and the cloud platform at the same time and often in different communication mediums using different communication protocols. The cloud platforms offer storage and processing of data, accumulated from multiple devices and gateways, on a large scale. The cloud platforms offer services like remote access to the data, design of triggers based on the data stored and visualization of data. In the following subsections these architectural elements are described in detail.

2.1.1 End devices

The end devices in the above architecture represent the 'things' which comprise the Internet of Things. These devices lie at the edge of the network and are usually equipped with limited storage and processing capabilities. The end devices perform two basic functionalities, namely sensing and actuation. The devices that perform sensing have sensors which gather data from the environment they are placed in. Depending on the storage and processing capabilities, the device stores the data or passes the data on to the higher layer of abstraction in the architecture. Devices performing actuation, are equipped with actuators and perform some action which is usually controlled based on a feedback gathered from the sensed data. The sensed data can be from the same end device or from a different group of end devices, depending on the application. For example, if we consider an application to count the number of free parking spaces in a parking area and display it, we can use end devices containing infrared sensors which act as sensing devices while we have another end device which displays the number of free spaces which can be considered the actuation end device. In this application, the actuation devices perform the actuation based on data from other sensing devices. The complexity of processing the sensed data and performing an actuation is passed on to higher layers of abstraction in the architecture. The end devices are also equipped with communication modules which allow them to communicate with each other or with entities in a higher layer of the architecture. These communication modules are usually diverse in nature and follow a variety of communication standards and protocols. These standards include 802.15.4, Bluetooth, Zigbee, LoRA, WiFi and beyond, which follow different methods for encryption and operate at different bandwidth and frequencies. Thus, one of the primary challenges of the Internet of Things lies in leveraging these standards and connecting the devices to the internet. The end devices are usually equipped with microcontrollers with 8-bit or 16-bit architecture. Thus, the operating systems running on these devices are usually low-level and have limited capabilities of executing complex arithmetic and logical operations. These devices are also programmed using low level or assembly languages. Hence the complexity of storage and processing of the data gathered from the end devices are passed on to higher layers of abstraction. The end devices may also follow different topologies, namely single-hop and multi-hop, depending on the type of application in which they are deployed. These topologies are illustrated in figure 2.1. In the one hop network topology depicted in figure 2.1(a), the devices directly connect to an entity at the higher layer of the architecture. On the other hand, for multi-hop topologies depicted in figure 2.1(b) , the end devices form a Personal Area Network and communicate among each other to pass data to an end device acting as the base station. The base station stores the data locally and passes the data on to a higher layer in the architecture.

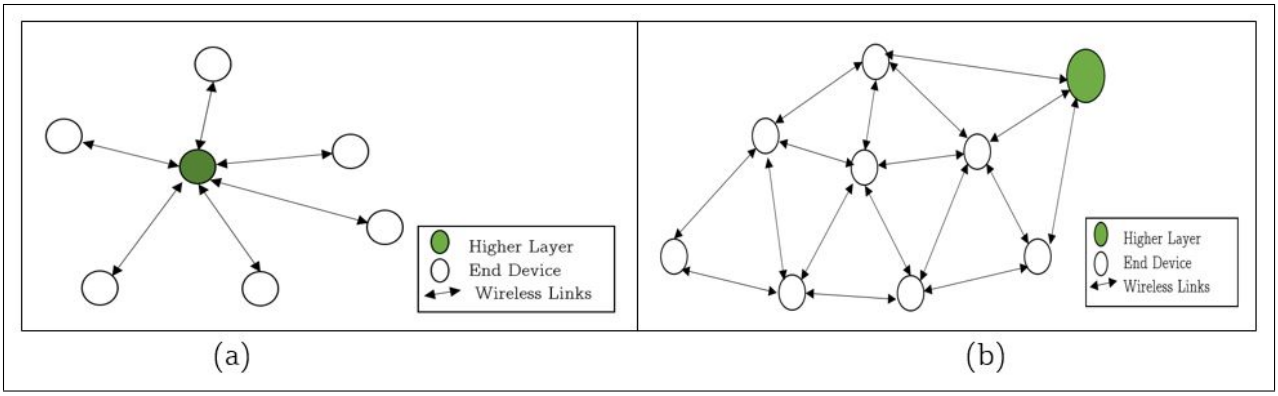

Figure 2.1: Topologies for end devices

2.1.2 Gateway

The idea of the Internet of Things is not only to connect new devices to the internet but to connect existing devices as well. However, two issues are encountered to connect devices to the internet. Firstly, some of these devices are not equipped with Ethernet or Wi-Fi interface to connect them to the internet. Installation of an interface to connect to the internet with a full protocol stack is also a cumbersome task on existing devices and devices with limited storage and processing capabilities. If we consider an example of a smart home, where we want to connect the electricity meter directly to the internet, the task of installing an interface to connect it to the internet would be an overkill just to get the data from the meter onto the internet. Similarly, some of the edge devices which are equipped to gather data from sensors lack enough memory to run a full IP stack on the devices. Secondly, there is an issue of interoperability, where existing and new devices follow different standards of communication. Some of these devices are equipped with Bluetooth Low Energy (BLE) communication modules, some of these devices communicate over 802.15.4 while some of them may also communicate over a wired interface. Instead of equipping these devices to connect to the internet, there was a need to have a device which can communicate in these mediums and also connect to the internet simultaneously.

Gateways can be primarily classified into two kinds, namely simple gateways and embedded control gateways, based on the functionality of the gateway in the architecture. A simple control gateway acts as a device which receives the data from the end devices in one form and packetizes the data to send it over the internet and vice versa. The gateway is not involved in processing of the data received from the end devices. On the other hand, embedded control gateways provide the functionalities of a simple gateway and provide processing and intelligence build on top of it. Thus, the embedded control gateway can be used to process data to offer functionalities like sensor data filtering and validation. For example, if the end devices are equipped with humidity sensors, the gateway can be used to reject negative values generated from the end devices. Thus, with embedded control gateways, complexity of the end devices are further offloaded on to the gateway device.

Gateways can support multiple devices at a time, communicating over different underlying technologies and protocols. This solves the issue of interoperability among devices. Thus, when a consumer buys a new end device, the complexity is reduced from how to connect the end device to the internet, to how to connect the

end device to the gateway leveraging the interfaces offered on the end device. We can consider an example where a consumer buys a smart thermostat which adjusts the temperature of the house based on ambient temperatures inside and outside the house. If we assume that the thermostat communicates over Bluetooth, we can connect it to a gateway device which can get live data about the weather of the city over the internet and communicate the data to the thermostat over Bluetooth. Thus, the complexity and the cost involved to equip each end device with an interface to connect to the internet is reduced by sharing the resources of the gateway to connect to the internet among the end devices. The gateway can also act as a management system for devices connected to it. For example, we can create a web service on top of the gateway which the consumer can use to control devices connected to the gateway. Thus, the use of gateways to bridge the gap between end devices and the internet paved a smoother path for the growth of the Internet of Things. However, gateways can only store data from a limited number of devices over a limited period of time due to processing and storage constraints. Hence, for complex processing of data and remote access to the stored data, the gateways are used to push the data further, to the cloud platforms.

2.1.3 Cloud platforms

The wide range of applications for the Internet of Things have different requirements for storage and processing and data. End devices and gateways fall short in multiple aspects to satisfy these requirements. For example, some of the applications require a global overview of the data generated from multiple gateways to build an application. If we look at a smart city application to ease traffic on different lanes based on the amount of traffic on each lane, we require data from each of these lanes to provide a working solution. Thus, data from a single gateway device is insufficient to build such a solution. Similarly, we may require remote access to data over a long period of time, generated by a set of sensors. If we consider a remote healthcare application, where we want to view historical data from a patient's wearable sensors, the capacity of the gateway is limited in terms of storage to provide such a solution. These shortcomings of the gateway are handled by using cloud platforms on top of end devices and gateways.

Cloud platforms offer primarily three basic services namely, data storage, data processing and remote access to data. Cloud platforms offer storage of data from multiple end devices and gateways over long periods of time. They offer different modules for processing data based on the kind of data that is stored and the kind of processing required. These modules range from simple ones like computing basic statistical measures on data to applying machine learning to large data sets for pattern recognition. Remote access to data allows users to view data generated from the things from anywhere in the world over the internet.

In addition to the above services, cloud platforms also offer visualization of data stored on the platform. For example, users can view the changes in heart-rate of a patient being monitored through wearable sensors connected to the cloud platform. The visualization also contributes to the outreach of IoT applications users unfamiliar with the technicalities of IoT. Cloud platforms offer trigger based services to perform actions based on the data accumulated from the end devices. For example, in the above example, if the heart rate of the patient is consistently over a certain value, an email can be sent out to the cloud platform to the responsible caregiver. Moreover, cloud platforms also offers libraries to simplify the development of IoT applications and allow non-technical people not familiar with programming embedded devices a simple interface to connect

end devices to the cloud platforms.

2.2 Types of architecture

In the above subsections we discussed about the elements comprising the architecture for the Internet of Things. Given these elements, the architecture can be classified into two basic types namely, three-tier architecture and two tier architecture respectively.

2.2.1 Three-tier architecture

In the three-tier architecture, the end devices, gateways and the cloud platform form the three tiers of the architecture. The end devices gather data from the sensors equipped with the device and send the data to the gateway. The gateway can perform two functions once the data is received from the end devices. The gateway can process the data and send the data to the cloud platform or locally process the data and send a command back to the end devices to perform actuations based on the sensed data. The cloud platform is used for storage of larger amounts of data in comparison to the gateway and perform complex computations on the stored data. For example, we can consider an application of controlling humidity in a house based on a trace of humidity values sensed and gathered over time. In such an application, the end devices sense humidity data and are also equipped with a dehumidifier to act as an actuator. The devices, then send the data gathered from the humidity sensors to the gateway. The gateway can then act in two different ways. It can locally store and process the data to control the parameters of the actuator and use the cloud platform for data storage or bypass the processing to the cloud platform altogether. The control of the actuator is then also bypassed to the cloud platform, which controls the actuator by communicating with the end device through the gateway. The three-tier architecture is depicted in figure 2.2.

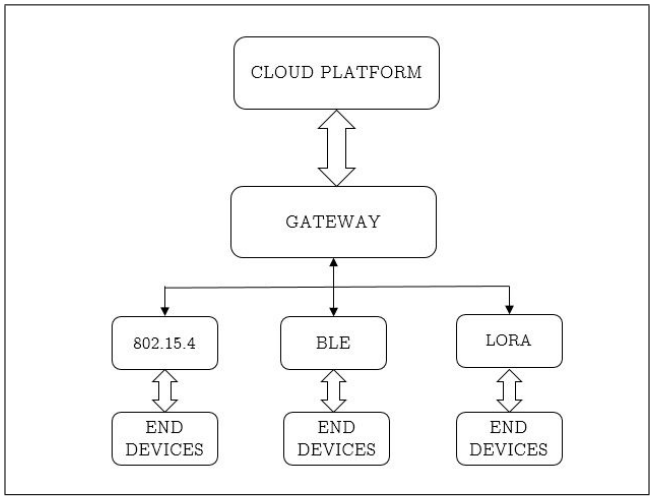

Figure 2.2: Three tier architecture for a cloud platform based IoT application.

2.2.2 Two-tier architecture

In the two-tier architecture, the gateway is bypassed and the end devices communicate directly with the cloud platform. Thus, in this kind of architecture, the end devices are equipped with interfaces to connect to the internet and use an IP based protocol stack. The absence of the gateway also entails the end devices to have higher storage and processing capacities in comparison to the three tier architecture. The communication between the end devices and the cloud platform needs to be reliable as well, since the gateway cannot be leveraged here for storing the data for longer periods if the connection between the devices and the cloud platform is lost. The two-tier architecture also allows finer control of the devices from the cloud platform directly. Since the cloud platform can be accessed remotely, the end devices can be controlled remotely as well, directly through the cloud platform for this kind of architecture. The data gathered from the devices are also mapped directly back to the end device since the devices communicate directly with the cloud platform. If we have multiple humidity sensors and dehumidifiers from the previous example, with the three-tier architecture, the gateway has to maintain a state or context while sending the data to the cloud platform to map the data to its source. For example, if the humidity from a device with ID 'n' is received by the gateway, it must mark the data with the ID of the node when passing the data on to the cloud platform. On the other hand, in the two-tier architecture, the data from the devices are directly sent to the cloud platform. Hence, the cloud platform can directly map the data to its source reducing the complexity of intermediate state storage.

2.3 Protocols

In this section, we discuss about protocols used for communication of end devices and gateways with the cloud platforms. The protocols define a messaging model to define how the above entities communicate with each other, the role each of these entities play and the structure of messaging between them. Based on the messaging model we can broadly classify the protocols into two types, namely request-response protocols and message-passing protocols. In the following subsections we describe these types of protocols and discuss about two widely used protocols from each type.

2.3.1 Request-response protocols

In request-response type protocols, the participants in the protocols communicate with each other through a set of requests and corresponding responses. The interaction model for this kind of protocols is defined based on two entities, server and client. A server is an entity providing a set of services or resources which are availed by a client. The client requests the server for a certain resource while the server listens for incoming requests. The server processes the request and responds to the client accordingly. Thus, the interaction model between the server and client is usually pull based, where the entity sends a request for a service when the service is required. Two commonly used protocols based on the request-response model are Hyper Text Transfer Protocol (HTTP) and Constrained Application Protocol (CoAP) which are discussed in the subsequent subsections.

2.3.1.1 Hyper Text Transfer Protocol (HTTP)

Hyper Text Transfer Protocol (HTTP) is an application layer protocol based on the request-response messaging model. The architecture for the HTTP protocol involves a server and a client in its simplest form. The

server offers services in the form of shared resources, which are identified uniquely by a Uniform Resource Identifier (URI). To avail the services offered by a server, the client sends a request message to the server by including the URI of the resource, protocol version and a request method along with other parameters and a message body. The request methods are compliant with the RESTful architecture, where the methods follow a standard set of predefined operations. The methods are based on the Create-Read-Update-Delete (CRUD) paradigm and are defined by a set of HTTP verbs. The GET request method is used to read a resource, the PUT request method is used to create a resource, the POST request method is used to update a resource and the DELETE request method is used to delete a resource. The server responds to the request with a status line and a code which denotes success or error. For example, if a client sends a GET request for reading a resource the server responds with a success code 200 and the requested resource in the body or responds with errors of different types including unavailability of the resource or the server.

In a more complex architecture, there are entities present between the client and the server which handle requests on behalf of the client or the server. A proxy server receives requests from the client and forwards the request to the server by reformatting the request. On the other hand, a gateway acts as a receiving agent between the client and the server and performs functions like protocol translation and running firewall system. This architecture is depicted in figure 2.3. The proxy converts a request made locally to another request addressed to the server while the gateway acts as a protocol translator between a client running a different protocol and the server. HTTP also offers secure communication between the client and the server over the Transport Layer Security (TLS) protocol for authentication while using the services offered by the server.

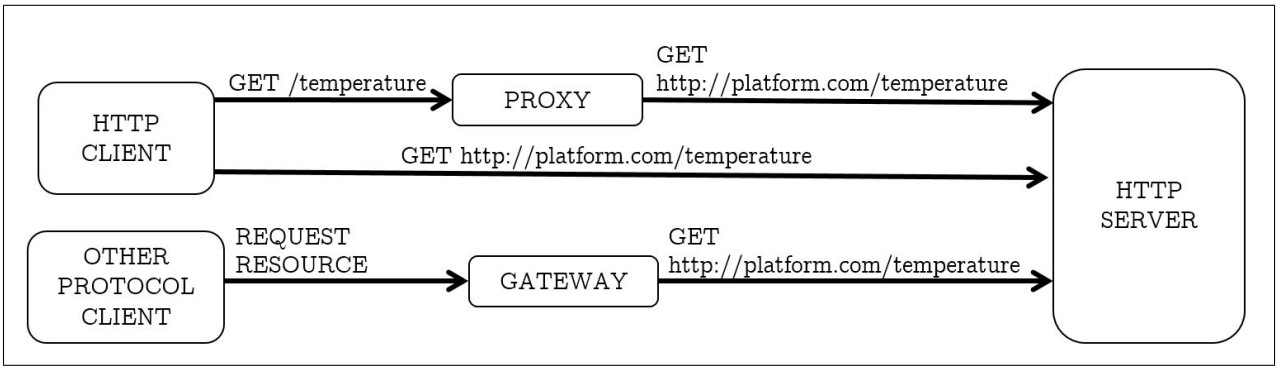

Figure 2.3: Architecture for HTTP request-response messaging.

The communication model and architecture of HTTP makes it a suitable communication protocol for the Internet of Things. HTTP servers offering services for the Internet of Things, expose resources and methods to store and process the data. End devices and gateways running the HTTP protocol, leverage the PUT and POST methods to send, modify and store the data to the cloud platform. The devices use GET requests to fetch the data and DELETE to delete the respective data from the server. Thus, the compliance with the RESTful architecture, offers a standard for designing and deploying IoT applications leveraging the HTTP protocol. Moreover, improvements added to the HTTP protocol including transmission of parallel responses from the server, header compression and server push facilitates the use of HTTP for the Internet of Things

even further.

2.3.1.2 Constrained Application Protocol (CoAP)

Constrained Application Protocol (CoAP) is a web based application layer protocol developed for devices with constrained memory and processing power while also considering constrained and lossy networks with high packet drop rates. Since the devices have limited resources, implementation of the full stack of HTTP based on the REST architecture, entails a large overhead for the devices. CoAP is aimed at building a RESTful architecture for these devices and environments taking the constraints into consideration.

Similar to HTTP, CoAP follows the request-response model and leverages the use of URIs for exchange of messages. In addition, CoAP allows built-in resource and service discovery from devices along with support for multicast messaging and low overhead. The low overhead is achieved by using simple mechanisms to handle packet duplication and to associate packets with requests. Packets are marked using a 16 bit ID to avoid duplication while a variable length token is used to associate a packet with a particular request-response exchange. However, unlike in HTTP, where there is clear distinction between a client and the server, in case of CoAP, devices running the CoAP protocol can act as both a server-device and the client-device. CoAP relies on the use of proxies to translate queries made to a CoAP endpoint to HTTP requests to another server. The proxy may be a CoAP-to-CoAP proxy where the request is translated from one CoAP request to another or a cross proxy where the request is translated to another request using a different protocol like HTTP. The example of the use of proxies is illustrated in figure 2.4.

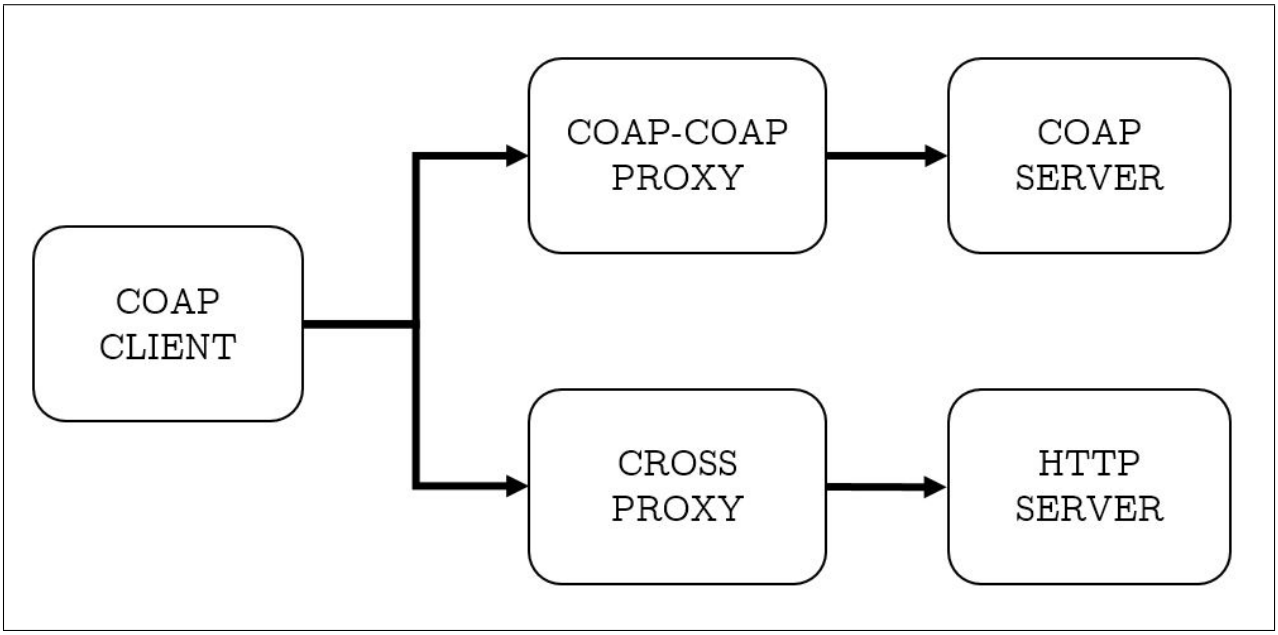

Figure 2.4: Use of proxies in CoAP.

The CoAP protocol leverages the UDP protocol to send asynchronous messages with optional reliability. The messages can be confirmable or non-confirmable depending on the option chosen by the user. The acknowl-

edgement of the messages are either piggybacked to the response or sent separately as an acknowledgment message. CoAP also supports an option for observation, where a client device can register for observations on a resource while the resource sends the client devices, the changes made to the resource.

Thus, use of CoAP is particularly useful in end devices with limited processing and storage where the end devices connect directly to the internet using protocols like 6LoWPAN. By blurring the line between the client and the server, CoAP brings the capabilities of a simple server on the end devices. Leveraging the CoAP server on the end devices along with the feature of observe, users can extract values from end devices without having to request for the data each time. Along with these advantages, CoAP also has a low header overhead which allows simpler parsing and higher reliability in constrained networks where packet drops are common.

2.3.2 Message-passing protocols

In message-passing protocols, clients following the protocol exchange messages among each other through a central broker which acts an arbiter for these clients. The clients for message passing protocols are either data producers or data consumers or both. Data producers send their messages to the broker marked with an identifier. The broker parses the message and according to a set of rules based on the identifier forwards the messages to the corresponding message consumers. Thus, the interaction model for this kind of protocols is push, where the consumer does not request for the message, rather the message is pushed to the consumer from the broker. Two commonly used message passing protocols used for Internet of Things, MQTT and Advanced Message Queuing Protocol (AMQP) are discussed in the subsequent subsections

2.3.2.1 MQTT

MQTT is an open message passing type protocol based on the publish and subscribe model. The architecture for the MQTT protocol consists of an MQTT broker which handles the exchange of messages among the clients. Based on the publish-subscribe paradigm the clients are primarily either message publishers or message subscribers or both. Message publishers generate messages while message subscribers receive them. The publisher and the subscriber are decoupled in space, hence they do not need to know each other uniquely to communicate with each other. They communicate with each other indirectly through the MQTT broker. Message publishers publishes messages addressed to the MQTT broker. The MQTT broker filters these messages based on two types of filtering, content-based and subject-based. In subject-based filtering, the messages are filtered based on a string called topic defined by the publisher of the message while publishing it. In content-based filtering, messages are filtered by the broker based on the content of the published message. Message subscribers subscribe to a set of topics or specific content filters, depending on the type of filtering, for receiving messages from the MQTT broker. The MQTT broker receives the messages from the publishers, filter them according to the filtering method and publishes the message back to the corresponding subscribers. This architecture is depicted in figure 2.5. However, the subscriber and the publisher are decoupled in time as well, hence do not need to be connected at the same time for the subscriber to receive the messages. When the messages are published, the broker stores the messages for a period of time and then sends them to the corresponding subscribers when they connect to the broker. The messages can be exchanged among the clients and the broker defining a Quality of Service in three levels, namely at most once, at least once

and exactly once. The QoS level can be defined differently for a publishing a message to the broker and for the broker publishing the message to a subscriber.

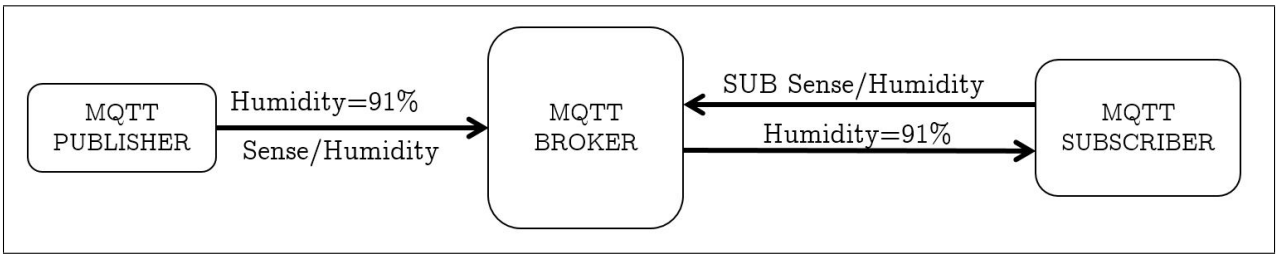

Figure 2.5: Subject based publish-subscribe using MQTT.

The MQTT protocol is lightweight with a small overhead and is thus suitable for installing on the client side for constrained devices. It also provides an advantage over request-response protocols in terms of scalability since the operations of the broker can be done in parallel since the clients are decoupled both in time and space. MQTT also does not use queues to store the messages when they are received from the publisher, thus bypassing the complexity of naming and handling queues uniquely. These advantages have contributed in the growth of MQTT as one of the most widely used message passing protocols for the Internet of Things.

2.3.2.2 Advanced Message Queuing Protocol (AMQP)

AMQP is a message passing type protocol based on the Advanced Message Queuing (AMQ) Model which defines an architecture among clients implementing the protocol and the messaging middleware based on which the clients communicate. The AMQ model defines a modular set of components which comprise the server, running the messaging middleware for connecting clients. The server consists of three modules, namely exchange, message queue and binding. The exchange receives the messages from a client which publishes the message and routes these messages to the message queues. The message queues store the messages until clients are available to consume the messages and forward the messages accordingly. The relationship between the exchange and the message queue is called a binding, based on which the message is sent to a particular queue. The exchange follows the rules set in binding tables to forward messages to the appropriate message queues. The flow of messages for the AMQ model is illustrated in figure 2.6.

The AMQ model offers two distinct features to configure the server through the protocol. Firstly, a user can add exchanges and message queues at runtime using the protocol and secondly, the binding between the exchange and the message queues are programmable at runtime as well. When a message is received at the exchange, the exchange examines the 'routing key' property of the message. The routing key usually contains the name of a message queue when its directed towards a set of clients subscribed to a queue or it contains a topic based on which the messages are routed to specific queues. Based on the type of routing key used, the exchanges are of two types, direct message exchange which handles messages sent to specific messages queues and topic exchange, which routes the messages by parsing the topic used as the routing key. On the other hand, messages can be durable or temporary depending on whether they delete the messages once they are delivered to the consumer or stored for a longer period.

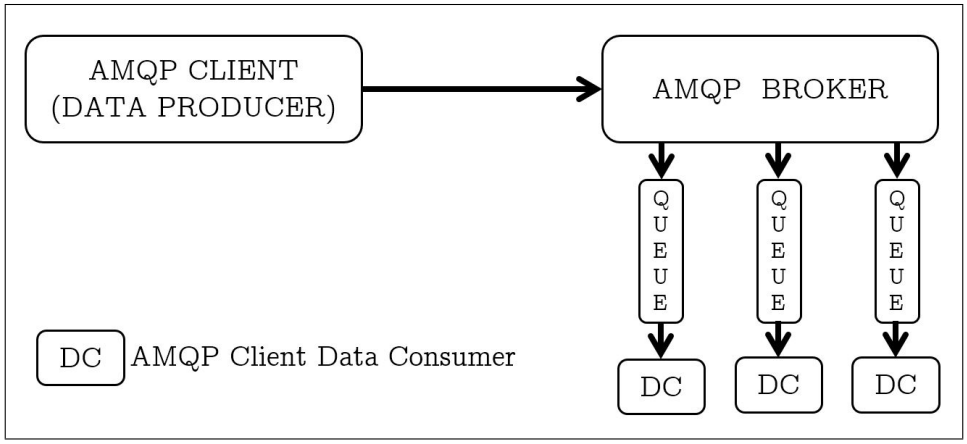

Figure 2.6: Architecture for AMQP clients and broker.

The AMQP protocol brings a modular approach to the server handling the messaging middleware, allowing the user to create different types of queues and engines for different applications. The protocol also offers more flexibility by allowing the user to change the logic based on which the messages are sent to different consumers based on the bindings defined.

Chapter 3

Taxonomy of IoT cloud platforms

Internet of Things applications are based on different domains ranging from healthcare, smart home and building automation to industrial applications, autonomous transport and smart cities. These applications have different requirements in terms of the type and amount of data they produce, the kind of processing the data requires along with the type of users who consume the data. For example, if we consider an industrial IoT application where a machine is being monitored for consistent performance and wear and tear, the monitoring has to be at a high frequency ensuring any issue occurring with the device to be foreseen. The data is also monitored by experts who have an understanding of how the machine works and how the data is to be interpreted if an issue arises with the functioning of the machine. On the other hand, if we look at a smart home application, where we monitor the ambience of the house, the frequency at which the data is gathered is much lower. The consumers of the data are the residents of the house who may not be well accustomed with the semantics of the data. Thus, the data must be well-defined, articulate and visual for ease-of-use of the consumer. Hence, if we look at different applications, the choice of the cloud platform to store and process the data from the IoT devices becomes an integral part of the planning required for designing and implementing the application.

When we consider cloud platforms for an Internet of Things application, we have a plethora of cloud platforms to choose from. Given the large number of platforms that are available, we require concrete parameters which we can use, to compare these cloud platforms and choose the one which suits an application. In this chapter, these parameters are illustrated, which can be used to compare cloud platforms and choose a suitable one based on the application.

3.1 Protocols

The communication between devices and the cloud platform is governed by the different types of protocols as mentioned in section 2.3. These protocols are primarily of two types, namely request-response protocols and message-passing protocols, based on the type of messaging model the protocols follow. The choice of protocol becomes an important factor when choosing a cloud platform based on the kind of IoT application to be deployed. These tradeoffs involved with the choice of protocols are described as follows.

Request-response protocols: In request-response oriented protocols, each resource on the cloud platforms has a well-defined end point in the form of a URI. The devices access the resources through this endpoint by using the API offered by the cloud platform based on the Create-Read-Update-Delete (CRUD) paradigm. The CRUD paradigm defines that each resource can be handled in four different ways. Users can create a new resource, read an existing resource, update a resource and delete a resource using the methods offered by the API. In these kind of protocols the device usually pushes the data to a particular endpoint by performing requests to update the resource at endpoint while applications or other devices fetching the data perform requests to pull the data from the endpoint. This kind of protocols are particularly useful when the request is to be made accompanied by certain conditions. For example, if we want to fetch data from a platform in a particular time interval for a specific node, the request can be made by adding these conditions to the body of the request.

This kind of protocols support two different architectures. In the first kind, the server is situated on the cloud platform and the devices interact with the server through a request-response protocol. An example of a protocol that can be used in this scenario is HTTP. HTTP supports the CRUD paradigm with 4 standard verbs, which define the kind of request being made. To create a resource, the verb PUT is used with the end-point URI where the resource is to be created. Similarly, a resource is updated using the POST verb, by defining the endpoint URI of the location where the resource being updated is stored. The GET verb is used to fetch data from the server and is usually accompanied by conditions. The data that satisfies these conditions are only returned in response to a GET request. The DELETE verb is used to remove a resource altogether. For example, if we consider a scenario where a device equipped with a humidity sensor is sending data to a cloud platform using the HTTP protocol. The device would first create a resource by performing a PUT request to the endpoint and then update the resource by performing a POST request. If the user wants to fetch the data where humidity is equal to a particular value, then the device can be used to perform a GET request with the added conditions to the URI. If we consider the URI as 'sample.platform.com' and the value to be 80%, the request would contain GET http://sample.platform.com?value=80. To delete values the device performs a DELETE request to the same endpoint. Most of the platforms support this interaction model and the HTTP protocol. These platforms include Xively, SenseIoT, Amazon AWS IoT, Parse, Phant and many others.

In the second kind of architecture, the server may be also hosted on the device itself. Thus, the interaction model between the devices and the cloud platform changes, where the cloud platform can perform a request to the server hosted on the device. The devices have constrained memory and processing capabilities, thus they are too resource constrained to run HTTP. For these kind of applications Constrained Application Protocol (CoAP) is used. The verbs used by CoAP are same as that of HTTP, however it has a reduced stack in comparison to the HTTP protocol and thus consumes fewer resources. An example of a cloud platform using CoAP is the mbed platform.

Message-passing protocols: In case of message-passing protocols, there are two entities involved, data producers which produce the data in the form of messages and data consumers which receive the data in the form of messages. A device can be both a data consumer and a producer at the same time. The messages are produced and consumed with conformation to a string known as 'topic' defined by a message broker, which

acts as an arbiter between data producers and consumers. For example, if the messages being produced are data from sensors, the topic can be set by the broker as 'Sensor Data'. The data producers publish the messages to a specific topic on the cloud platform while the data consumers subscribe to topics they are interested in. The cloud platform acts as a broker between the producers and the consumers, forwarding the messages to the consumers subscribed to the same topic.

In this kind of protocols the producer pushes the data to the cloud platform which furthermore pushes the data to other message consumers. Thus, if the use-case for which a platform needs to be chosen, entails reaching out to multiple data consumers, message-passing protocols prove to be more effective. This kind of protocols require fewer messages to be sent from message consumer to the cloud platform to actually fetch the data. As an example, we can consider a use-case where we are monitoring a water body for water temperature and other parameters measuring water quality. If we have a research team monitoring the values, with certain members responsible for monitoring certain parameters, a message-passing protocol will be more suitable. Each member can subscribe to the topic gathering data for the respective parameter he/she is interested in while the broker forwards the respective messages in accordance to the subscriptions. A researcher monitoring only the water temperature can specifically subscribe to the topic 'Sensors/WaterTemperature' to get the respective values.

Message-passing protocols can also prove to be effective when we want to associate data generated to the source of the data, i.e. we are interested in the data generated based on the node which is generating the data. For example, we can consider a use case where we have 10 nodes having different IDs and we want to get the data only from a particular node. This can be achieved by adding the node ID to the topic to which the device is publishing the data and subscribe only to topics in which the particular node ID is present. For example, if node 4 and node 6 generate humidity data, the topic to which they publish can be set as 'Sense/Humidity/4' and 'Sense/Humidity/6' respectively. Thus a user can subscribe to 'Sense/Humidity/*' for all humidity data or 'Sense/Humidity/6' for the data from node 6. Examples of platforms offering communication over message-passing protocols include Microsoft Azure IoT supporting communication over AMQP and MQTT and Amazon AWS IoT supporting communication over MQTT.

3.2 Interaction model

The communication between IoT devices and the cloud platform can occur in a bidirectional manner, i.e. devices can send messages to the cloud platform and cloud platforms can send messages to the devices as well. The interaction model of a cloud platform defines the nature of messages exchanged between the devices and the cloud platform. The cloud platform can be used to build applications to manage devices and process the data on the platform. Messages are also exchanged between the cloud platform and the applications built on top of the cloud platform. The messages can be sent from the application to the cloud platform and from the cloud platform to the application as well.

Device to cloud messaging: The majority of the messages exchanged between devices and the cloud platform are messages sent from the device to the cloud platform. The devices are required to communicate with the cloud platform to serve various purposes. Initially, a device may need to send a message to the

cloud platform with the required credentials to register itself. Following the registration, the device can start performing requests to the respective resources on the cloud platform for sending data to the cloud platform or publish data to a topic in accordance to the protocol being used to communicate between the device and the platform. The device may also perform a request to fetch the data from the cloud platform.

Cloud to device messaging: The cloud platform can be used to send messages to the devices to perform certain actions like updating parameters stored on the device or perform actuations from the device, based on messages sent from the cloud backend to the device. For example, we consider a scenario where a gateway device is connected to both a sensor network monitoring light intensity and an actuator controlling the light intensity of the light sources. The data collected from the sensors can be sent to the cloud platform. The cloud platform then processes the data and sends a message back to the device to modify the light output from the lamps. This can be achieved with the Microsoft Azure IoT platform which allows sending a message from its IoT Hub to the device. The device can then respond to the message accordingly and perform some action. Amazon AWS IoT also offers communication from the cloud platform to the device using their 'Thing Shadow service'. The Thing Shadow is a JSON document accessible to both the device and the cloud platform. This is an example of the distributed data sharing model, where the data is shared between two entities, the device and the cloud platform. To update a parameter on the device, the cloud platform can modify the parameter on the Thing Shadow to the desired value. The device will then read the Thing Shadow and update its own parameter accordingly. For example, if the device is connected to a lamp with a specific intensity, the cloud platform can modify the intensity by updating the light intensity parameter on the Thing Shadow.

Application to cloud messaging: Applications play a major role in processing of the data stored on the cloud platforms. Applications are used to fetch the data from the cloud platform and process the data. Thus, the messages sent from the application to the cloud platform usually involves performing requests to the respective resources to fetch the data for request-response protocols. The data can be then furthermore sent to other third party applications for processing or visualization. Applications may also perform requests to update a resource on the cloud platform after processing the data from another resource. For example, on Parse, where HTTP protocol is used to communicate with the cloud platform, the applications can query the class responsible for storing objects of a particular type by performing GET requests on the class. A sample class called TemperatureMilan may contain objects containing values of temperatures gathered from various sensors. An application can be written on top of the Parse platform to query and get the objects where the temperature is above 25 by performing GET requests to the endpoint of the respective class specifying the necessary condition. The application can also update another class used for storing average values of temperature in a 5-minute window, by performing a POST request to the respective endpoint of the class with the processed value.

Cloud to application messaging: The cloud platform can communicate with applications based on a push based interaction. For example, if an application subscribes to a topic for message-passing protocols, the cloud platform publishes the data to the application by sending messages to the application. Similarly, the cloud platform can send push notifications to an application based on certain changes made to a resource on the cloud platform. If we look at the Parse platform, where the HTTP protocol is used to communicate

between the application and the cloud, users can write applications on top of the platform to register for changes made to a specific class on the cloud platform. When the class is modified by another application or device, the cloud platform sends a push notification to the registered application.

3.3 Type of services

Vendors offering their services for the Internet of Things can be differentiated by the type of service that they offer. These services can be broadly classified into three kinds of services namely, Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS) and Infrastructure-as-a-Service (IaaS). These services differ from each other based on the domains on which the services are offered.

Platform-as-a-Service: Vendors offering their services in the form of PaaS, offer storage of data from the IoT devices and a platform as well to build applications on top of the platform to process the data generated by the devices. This platform can be used to visualize the data generated by the devices, perform basic statistical processing like calculating mean, median and extrema in certain time periods and also complex processing using other modules offered by the platform for data processing as well. For example, Microsoft Azure offers a platform for storage and basic statistical processing of data using stream analytics and advanced processing of data through a business intelligence tool called PowerBI. Amazon AWS offers modules like Amazon Kinesis for analyzing the data generated from IoT devices on a large scale and visualize the data on their business intelligence tool called Amazon Quicksight. The applications developed on top of the cloud platform can be used to send notifications to devices or users. They can be used to perform actuations as well based on triggers written on top of the platform. For example, Xively offers an option to perform actions using third party modules like Zapier based on certain changes in the data generated by the IoT devices. If we look at a simple application where a set of sensors are used to monitor the water level in a house reservoir, a trigger can be used to send an email to a user, when water level drops below a certain level.

Software-as-a-Service: In case of SaaS, the vendor hosts a software application on their own server and offers subscription to the users. A user can then subscribe to the service and send data to the software on the cloud. The data gets processed on the software and the user is notified of the output of the processing. The service offered here appears as a black box to the user, where the user provides some input and gets some output in return. The services offered by the vendors in the SaaS domain differ based on different application scenarios unlike PaaS. For example, Element Blue is a vendor offering SaaS solutions for IoT. Their solutions are tailor-made for different domains like water monitoring, energy management and healthcare management unlike PaaS, where the same platform is used to build different applications in case of different domains. Devicify offers proprietary and patented solutions for connected devices in the IoT domain applying business context to the IoT data. Thus, unlike PaaS, where the user can build applications on top of the platform, SaaS offers proprietary software which can't be modified. The SaaS service providers offer solutions specific to the use-case where the user feeds the data into the SaaS service and gets a desired output.

Infrastructure-as-a-Service: Services offered as IaaS include all the requirements necessary to build a complete IoT application, i.e. from the hardware end to the software end. IaaS services usually include the IoT end devices along with sensors required for sensing the environment, communication modules to

communicate with the cloud platform. The service provider also offers the cloud platform itself for storage and processing of data similar to PaaS. An example of a vendor offering IaaS is IoTSens which deals with applications involving smart cities ranging from smart lighting, smart parking to industrial IoT applications. They offer end device modules for different applications ranging from smart parking sensors, soil moisture sensors and waste height measurement modules for smart cities to air quality sensing modules, pluviometer and meteorological station modules for smart environment applications. The communications among the devices, the gateway and the cloud platform are offered using various technologies including LoRa and Zigbee. The cloud platform hosted by IoTSens offers services like mapping of sensors, historical data storage along with APIs and dashboards to monitor stored data.

3.4 Openness

Cloud platforms for IoT can be primarily classified into two types, namely open source platforms and closed or proprietary platforms, based on the availability of the source code for a cloud platform and the ability to modify a cloud platform. These two types of platforms are discussed in detail below.

Proprietary cloud platforms: Most of the cloud platforms available are proprietary in nature, i.e. the cloud platform is offered by a vendor as a service and the source code for the cloud platform is not available to the users and developers of applications. The platform is hosted by the vendor on their own servers and users subscribe to the services offered by the cloud platform. The services offered are usually limited by the kind of subscription purchased by the user. For example, a user may only have limited amount of space to store their data, register a limited number of devices and make a limited number of API calls based on a free tier subscription to the cloud platform. The features of the platform are rigid as well for closed platforms. For example, if a cloud platform specifies certain number of security mechanisms which a device or user needs to follow, the user cannot modify these specifications to make them more stringent or less stringent based on the applications. Examples of closed cloud platforms include Amazon AWS IoT, IBM Watson IoT and Microsoft Azure IoT, which offer basic services in IoT like storage, processing and visualization along with other advanced modules like data management for big data and data processing modules like machine learning. On the other hand, we also have proprietary cloud platforms which only offer basic services for IoT devices like storage, processing and visualization including platforms like Xively, Ubidots and Sense-IoT.

Open-source cloud platforms: Open-source cloud platforms are available on various repositories across the internet for cloning and self-hosting. To host an open-source platform a user needs to clone a repository of the platform and install it on a server to avail the services of the cloud platform. However, this requires a basic level of expertise of installing the dependencies involved with the cloud platform and also knowledge of solving the roadblocks encountered while installing the same. For example, if a user is using a closed platform, the user can simply opt for a service offered by the platform, use the API offered by the platform and start pushing data. However, for an open-source platform the user has to first clone and install the platform to get it running consistently before pushing the data onto the cloud platform.
This is a tradeoff involved for open-source cloud platforms. On the other hand, since the source code is available for the cloud platform, the user deploying the cloud platform has the freedom to modify the platform according to their needs. For example, if a user wants to deploy the cloud platform and store

data which requires basic amount of security, the user can modify the source code of the cloud platform to reduce the security mechanisms involved. Similarly, a user can also add new features to the platform building on top of the features already available from the cloud platform. Open source platforms can be hosted by the vendor as well where the user can simply register and access the platform. For example, Phant is an open-source platform hosted by Sparkfun Inc. at 'data.sparkfun.com' while it can also be self-hosted by cloning the respective repository. Some other examples of open-source platforms include SicsthSense, an open source platform developed and maintained Swedish Institute of Computer Science (SICS) and Parse, a closed platform which was made open source when its proprietary counterpart opted for terminating its services in 2017.

3.5 Cost

The cost involved for using a cloud platform can be classified into two basic types, the cost to use the services offered by the platform and the cost incurred to host the platform on a server. These costs are different for open-source platforms and closed platforms. For open-source platforms, the cost incurred is of the second type where the user has to deploy the platform on his/her own hardware and there is a cost involved in hosting the platform along with maintenance costs. Since the platform is usually cloned from a repository, the entire platform is available to the user. Thus, there is no cost involved in using the services offered by the platform like processing the data and performing API requests to the cloud platform. On the other hand, for the closed platforms, the platform is usually hosted by the entity offering the platform and thus there is no cost involved for hosting the platform on a server. However, the user has to pay for the services offered by the platform. The costs incurred for using closed platforms are based on these following models.

Device-cloud communication: The use of cloud platforms entails exchange of messages between the device and the cloud in the order of thousands per day depending on the application being deployed. For example, if a device is sending data to the cloud once every hour, the number of messages are sparse. However, if a device with multiple sensors sends data every second, the number of messages in a day from a single device accumulates up to the order of thousands. These messages can be bidirectional, i.e. from the device to the cloud or from the cloud to the device based on the interaction model between the platform and the device. Various cloud platforms offer subscription to their services in terms of number of messages that a device and the cloud platform can exchange. For example, Amazon charges the user per million messages being exchanged between the device and the cloud platform. The pricing is different in different regions of the world. Similarly, Sense IoT charges the user based on every 1000 API calls made from the device to the cloud platform. Microsoft Azure IoT charges the user in terms of 3 tiers which allows exchange of messages from 400,000 per day to 300,000,000 per day between the device and the cloud platform.

Storage: A major aspect of Internet of Things applications is the amount of data that the devices generate and their storage on cloud platforms. The amount of storage may differ based on applications. For example, textual data from sensors would consume lesser amount of space in comparison to multimedia applications like visual sensor networks, which require storage and processing of images and videos. The amount of storage required is also dependent on how frequently the data is gathered from the end devices to be stored on the cloud platform. Thus, cloud platforms may charge differently based on the amount of storage a user or

device consumes for the use-case scenario. For example, Sense IoT charges the user based on multiples of 1MB storage on the cloud platform.

Number of devices: Most IoT platforms require a device to be registered to the cloud platform in order to exchange messages with the cloud platform. This registration is validated usually, with assignment of an API key or a certificate to the respective device. Thus, the cloud platform is aware of the number of devices a user has registered and can charge the user based on the number of devices being used. For example, Microsoft Azure IoT allows connection of 500 devices for free beyond which the user has to opt for a paid tier in which unlimited devices can be connected to the platform. IBM Watson IoT charges the user in various tiers in multiples of thousand devices registered to the platform.

In this pricing model, use of a gateway becomes handy, where many end devices can be placed which connect to the gateway while the only device actually registered to the platform is the gateway itself. Thus, the cloud platform is oblivious of the actual number of devices gathering the data and is aware only of the gateway device. However, using a gateway would increase the number of messages being exchanged between a device and the gateway, since it aggregates data from multiple devices and sends them in unison. Therefore, the use of gateway can be seen as a method to reduce cost when only the number of devices is a parameter for measuring the cost, however when number of messages are also considered in the pricing model, this advantage is nullified.

Data processing and visualizations: Cloud platforms offering their services based on the Platform-as-a-Service (PaaS) paradigm, allows the user to build applications on top of the platform and use these applications to process the data or visualize the processed data from the devices. Cloud platforms can thus, charge users based on the amount of data being processed. For example, IBM Watson IoT platform charges the user in multiples of 1MB of data processed on the platform. It also offers other pricing options for different kinds of sensor networks. For example, for a visual sensor network dealing with images, Watson IoT charges the user in terms of face detection events and image tagging events based on the number of events handled. Cloud platforms may also charge the user for visualization of the data. This kind of pricing model is based on the number of variables being visualized. We can consider each physical quantity measured by the sensors as a variable. For example, temperature of a building can be considered a variable, while humidity of the building can be considered as another variable even though the sensor data is produced from the same device. Ubidots offers visualization of upto 5 free variables per device upto 5 devices, beyond which the user has to opt for a paid version.

3.6 Authentication and authorization

The data generated from the devices are stored on the cloud platform in the form of resources. These resources have their own endpoints, which can be accessed using the APIs offered by the platform. For example, if a cloud platform is used to store the intensity of light inside a building, the data will be stored in the form of a resource. There will be a specific endpoint for the resource like 'light.userID.platform.com' through which the light intensity values stored on the platform can be accessed to read, write and update the values. However, these resources require to have restricted access such that the resource is accessible only to users

owning the resource or having authority to access the resource. For example, the user owning the device can have both read and write access to the resource while another user who wants to process the data from the device may only have read access to the resource.

The authorization to access a resource is a two-step process. When a device attempts to access a resource, the platform first tries to authenticate the device, i.e. it tries to verify if the device is indeed the device it claims to be or it is a rogue device trying to gain access to the platform. However, even after a device is authenticated, it can only have access to resources which it is authorized to access. This access control for resources are implemented based on various authorization mechanisms. The methods used to authenticate the user for accessing the cloud platform are described below followed by the methods used for authorization.

3.6.1 Authentication methods

Authentication methods are based on a user either having knowledge of an entity or actually possessing an entity which the user can then use to authenticate itself. For example, a user can have knowledge of the username and password used to register to a cloud platform or a user might be in possession of a certificate provided by the cloud platform during registration. The following authentication mechanisms are used in various cloud platforms for IoT.

Username and password: The simplest way of authorizing a device to access the data is to use the username and password of the user owning the device. When the device communicates with the cloud platform, the device also needs to send the username and the hash of the password along with the message in order to gain access to a particular resource. For example, with Sense IoT, the device is required to send the username and hash of the password in the body of an HTTP POST request to authenticate itself to access the services offered by the cloud platform.

API keys: When a device registers to a cloud platform, the cloud platform assigns the device API Keys, which are character strings of a certain length. These API Keys can be used for accessing the services offered by a cloud platform. When a device communicates with the cloud platform, it must attach the respective API key it was assigned, to authenticate itself to the cloud platform. Devices which require write access to resources need to authenticate themselves with keys that would offer write access to resources. On the other hand, applications processing the data stored on the platform may use only the API Keys assigned for reading, to read the values stored in a resource. For example, Phant from Sparkfun uses two kinds of API keys, PUBLIC_KEY and PRIVATE_KEY for authorizing access to resources. The PUBLIC_KEY can be used only for reading resources while the PRIVATE_KEY can be used for both reading and writing.

Certificates: Some platforms may require the devices exchanging messages with the cloud platform to actually possess certificates assigned by a certification authority. The certificates are available to a device when the device registers to the cloud platform. These certificates must be downloaded and stored locally on the device. When the device exchanges messages with the cloud platform, the device should point to the certificate on its storage to gain access to the cloud platform. For example, Amazon AWS IoT uses different mechanisms for different protocols. For MQTT, Amazon uses X.509 certificates to authenticate a device and

authorize access to resources.

3.6.2 Authorization methods

Once a device is authenticated to access a cloud platform, authorization methods are used to determine if a particular device has authority to access to a particular resource it is requesting. The following methods are used by various platforms to authorize a device to access a resource.

Access control list: In some platforms the authorization to access resources is handled based on an Access Control List (ACL) maintained by the cloud platform. In an ACL, we have a matrix consisting of users on one side and resources on the other. If there is an association between a user and a resource, only then can a device or application owned by the user have access to that particular resource. The association may be of different kinds, for example, a user can have a read association with a resource, a write association with a resource or both. Thus when a device requests access to a particular resource, the ACL is checked with the user of the particular device to determine if the device actually has access to the data. For example, Parse is a cloud platform which uses ACL to control access to resources stored on the platform.

Roles and policies: The use of ACL on particular users and resources can be taken one step further by defining roles which have particular types of accesses to specific resources. The roles are associated with policies which define the resources a particular role has access to and the type of access it has. For example, we can define a role called 'admin' which can have read and write access to temperature data, humidity data and light data from a building. Users can then be assigned a particular role based on which the user can access the resources. For example, if a particular user is assigned the role admin, that user would automatically have access to the resources mentioned in the policy for the role. If we look at a cloud platform, Amazon AWS Identity and Access Management (IAM) assigns policies to users to access different modules of Amazon AWS like AWS IoT, AWS Lambda for writing scripts on the cloud platform, AWS S3 to store files and many other services. The users can be further combined to form groups and the group can have its own policy which is shared among the users in the group.

3.7 Libraries for devices

The services offered by a cloud platform are accessible through an Application Programming Interface (API) based on the protocol used to communicate between the devices and the cloud platform. The API offers a set of functions and procedures that describes the behavior of the cloud platform in handling the data when requests are made to the cloud platform. For example, if humidity data is stored on an endpoint 'humidity.data.platform.com', the API offers functionality to read, update and modify the data and describe how the data is handled when a particular request is made. To leverage and actually implement the specifications of the API offered by a cloud platform, the user is required to write a piece of code on the device in the language of his/her choice. Thus, the API can have multiple implementations in different languages. For users who are accustomed with programming, this is an achievable task if the endpoint is given and the user has the documentation of the API for the given platform. However, with IoT growing rapidly, the services are also accessed by people who are not well accustomed with programming. Thus, libraries play a pivotal

role in making the API more accessible to users.

The cloud platforms offer libraries for different devices like Raspberry Pi and Arduino along with quickstart guides. The libraries offered in various programming languages offer client classes which simply need to be initialized. The client is equipped with methods to access the cloud platform in different ways by leveraging the APIs. If an API for a cloud platform offers a procedure to read a single data point from the data stored on the cloud platform, the library can leverage the same to offer a function to read multiple data points on a period defined by the user. The user can obtain the necessary output he/she requires by performing a function call from the library, being oblivious to the underlying complexity. Thus, libraries for devices makes development on devices easier for the user. For example, IBM Watson IoT offers libraries for MQTT in C, Python, Java and Node.js. The user needs to choose a language to program the device in and download the Software Development Kit (SDK) corresponding to the programming language. Then, the user is required to define the configuration of the device in terms of the organization ID and security tokens and initialize the client by passing the configuration. Once the client is initialized, the user can call methods associated with the client and perform operations leveraging the underlying API. For example, the user can pass data gathered from a sensor to the client and publish the data to a topic on the cloud platform.

Libraries for the devices are usually open-source, available for contribution from other developers and the open-source community. For example, Parse offers its open source libraries on Github for JavaScript, Embedded C, PHP and other languages as well. The notion of a library being open-source is not tied to a platform being open-source or proprietary. Closed platforms like Amazon AWS IoT and Microsoft Azure IoT offer open-source libraries for various devices on Github.

3.8 Libraries for applications

Cloud platforms offer libraries to build applications on top of the platform to process the data accumulated from the devices. Similar to the purpose of libraries for devices, libraries for the cloud platforms makes it easier for the user to manage the devices and data stored on the cloud platform. These applications can be used for various purposes like storage of data in a database on the platform, performing queries on the data, passing data onto other entities and visualization of the data as well. Some platforms also offer gateway libraries which can be used to receive data from edge devices on a gateway using a medium or protocol different from the one used to communicate with the platform. The gateway API is then used to send the data to the cloud platform. For example, Microsoft Azure IoT offers libraries to access the Azure IoT Hub to interact with the connected devices and manage security for the devices. The libraries are offered in .NET, Java and Node.JS. Microsoft Azure IoT also offers gateway libraries to handle data arriving through Bluetooth and gather the data to send it to the Azure IoT platform. IBM Watson IoT offers libraries to build applications on the platform in Python, Node.JS, C# and Java. These libraries can be used to communicate with the devices using the MQTT protocol by subscribing to device events and device statuses. The library can be used to publish events and commands to the devices connected to the Watson IoT platform as well.

Libraries for the platforms can be also used to write triggers and rules based on the data being received by the cloud platform. For example, Parse offers a library in Node.js to write functions on the cloud platform

which can be called from devices or from other applications on the cloud platform. Amazon AWS IoT offers simple code snippets to write triggers on their own console to handle the data based on the topic to which the data was published using MQTT. The data received can be then inserted into a noSQL database, or uploaded as a raw file to the S3, or a script can be called when the data is received. For example, leveraging the AWS IoT platform a user can write a rule that if the data is published on topic 'temperature/node1', it should be inserted to a database allocated to node1.

Chapter 4

Overview of IoT cloud platforms

In this chapter, we discuss about some of the cloud platforms currently available for the Internet of Things. Among all the cloud platforms available, we have chosen a few such that we can study the diversity of these platforms in terms of the features offered by these platforms. We discuss about these diverse set of platforms in detail, demonstrating their features in accordance to the taxonomy defined in the previous chapter. We describe these cloud platforms in detail in section 4.1. In section 4.2, we summarize the features of these platforms and present a trend with the most prominent features among these platforms.

4.1 IoT cloud platforms

There are a plethora of cloud platforms available when choosing a cloud platform for a use-case scenario in IoT. With our taxonomy defined in the previous chapter we aim at simplifying the process of choosing a cloud platform and provide a detailed overview of some of these cloud platforms such that the reader can have a clear understanding of what services these cloud platforms offer. We have studied 12 cloud platforms offering Platform-as-a-Service (PaaS) in detail. The features of these platforms are classified in accordance to the taxonomy and are depicted in table 4.1 . In the following subsections, we describe each cloud platform with brief introduction and a working principle followed by the services offered by the platform. We also discuss about the interaction model between the user devices and the cloud platform, authentication and authorization mechanisms and costs of using the cloud platform. The libraries offered by each platform for the end devices and for building applications on the platform are highlighted as well. In subsection 4.1.13 we discuss some of the other cloud services offered for the Internet of Things and the rationale behind not including them in the detailed study.

Platform	Protocol	Cost Metric	Security	Interaction Model	Openness	Libraries (Devices)	Devices Supported	Libraries (Apps)
Xively	HTTP, MQTT	Not specified	API keys, Oauth for 3rd party	Push data from devices, pull data using applications	Proprietary	C, Python	Arduino	JavaScript, ObjectiveC, PHP, Ruby, Python, Application development for Android
IBM IoT Foundation	HTTP, MQTT	Amount of data exchanged and processed	API keys, certificates	Devices push data to cloud, applications pull data from devices and cloud	Proprietary	Python, Java, Node.js, C#	Any device running the SDK	Node.js, Python, C#
Sense IoT	HTTP	Per 1000 API Calls	Username-Password hash with session ID, Oauth for 3rd party	Push data from devices, pull data using applications	Proprietary	C, Python	Arduino, Electric Imp, RPi	PHP, Python, Application development for Android and iOS, Javascript
ThingSpeak	HTTP	Free on offered endpoint	API keys	Push data from devices	Open-source	MATLAB, Python	Arduino, Electric Imp, RPi	MATLAB, Ruby 2
Ubidots	HTTP	Number of variables visualized	Tokens/API keys	Push data from devices	Proprietary	C, Python, Java	Arduino, RPi, Spark Core, Electric Imp	Python, Java, C, PHP, NodeJS, Ruby, LabVIEW
Sparkfun (phant.io)	HTTP	Free on offered endpoint	API keys	Push data from devices	Open-source	C, Python,Node.js, Python, C#, Ruby, Java	RPi, BBB, Electric Imp, WiFly	Not Specified

Platform	Protocol	Cost Metric	Security	Interaction Model	Openness	Libraries (Devices)	Devices Supported	Libraries (Apps)
AWS IoT	HTTP, MQTT, Websockets	Per 1 million queries, module based pricing	Certificates, user groups and policies for authorization	Push data from devices, pull data using applications	Proprietary	SDK in C and NodeJS	Arduino	C, NodeJS
mBed	CoAP	Free on offered endpoint	TLS libraries for web authentication	Push data from devices, pull data from device	Open-source	C++	Any device running the mBed OS	Not Specified
SicsthSense	HTTP, MQTT	Free on offered endpoint	API keys	Push data from devices, pull data using applications	Open-source	JSON	Any device running the SDK	Python, Application development for Android
Azure IoT Hub	HTTP, MQTT, AMQP	Number of applications, disk space used	Tokens, connection strings	Push data from devices, pull messages from the device	Proprietary	.NET, C, NodeJS, Java	Any device running the SDK	NodeJS, Java
Parse	HTTP	Free on offered endpoint (till 2017)	API Key and application ID	Push data from devices, push notifications for applications	Open-source	C, EmbeddedC	Arduino	Javascript, PHP, .Net
ThingPlus	HTTP	End devices and triggers	Oauth, tokens	Push data from devices, pull data using applications	Proprietary	C (Arduino)	Arduino, Beaglebone, Raspberry Pi, Edison	Not Specified

Table 4.1: Comparison of cloud platforms based on our taxonomy

4.1.1 XIVELY

Xively [13] is a platform offering connectivity between end-devices and the cloud with the use of RESTful APIs over HTTP or with the use of MQTT. Based on the Platform-as-a-Service (PaaS) model, Xively can be used to receive, store and visualize data gathered from the devices, on the cloud. To use the service, a device should be first registered to Xively following which it can be used as a source generating data. This data received from the devices, can then be stored, analyzed and viewed by the user or a monitoring application.

Protocols used: Devices can communicate with the Xively API primarily using two methods, RESTful calls over HTTP or through MQTT. RESTful calls are addressed over HTTP to a set of URIs, which handle device information and data, as specified in the API documentation. Xively can also be accessed through any MQTT enabled device. Xively supports the publish-subscribe paradigm of MQTT. Topics for the publish-subscribe paradigm are determined based on the resource, the device is subscribing or publishing to. For example, if the device wants to subscribe to a feed numbered 556, the topic should be '/v2/feeds/556.json'. The stream of data generated from each device is distinguished through these uniquely named topics.

Working principle: The devices connected to Xively communicate with the various API resources offered by Xively. These API resources can be classified into four types, namely products, data, triggers and keys. These resources are used to create higher abstractions which offer ease of access to the applications and also reduce the number of calls to be performed to access multiple resources.

The resource for 'Product' defines the generic class of the actual devices being used, including a product name, product description and a template for data generation. The product is individually instantiated to form a 'Device'. Each Device when instantiated from the product is assigned a unique serial number.

Devices generate data in the form of Data-points. A 'Data-point' is a single value generated from a device with a timestamp of when it was generated. Various Data-points are accumulated to form a higher abstraction in the form of an archive called a 'Data-stream'. Data-streams are bidirectional streams used by the Xively API and a registered device to communicate between each other. Each Data-stream is responsible for a specific type of attribute or information. For example, we can have a Data-stream for temperature sensor data. An accumulation of Data-streams assigned to a device is defined as a 'Feed'. A device can have multiple Data-streams but can have only one Feed. The metadata for Feed may also specify the location or a tag for the device to support mobility.

The Xively API also provides support for 'Triggers'. They can be used to create a notification when data generated by a device meets certain requirements, for example, when data from a sensor exceeds a value. This event can be expressed as a Trigger with a request to an API for third party platforms like Zapier or IFTTT. These third party platforms can be used to take certain actions by leveraging the trigger from Xively, for example, send an email using Zapier when the ambient light in a room is high after midnight.

The resources can be accessed and modified using HTTP calls to the respective APIs for the resources. The Feeds and Data-Streams are offered in JSON, CSV and XML Formats. They can be 'read' using GET calls,

'updated' using 'PUT', 'created' using 'POST' and 'deleted' using 'DELETE' calls to the respective resources, following the CRUD paradigm for REST calls. Data for Product and Device resources can be requested in JSON format. On the other hand, Triggers can be created in XML and JSON formats. Xively also offers visualization for data streams which can be availed as an image in the Portable Network Graphics (PNG) extension.

Interaction model: Registered devices push data to the Xively server through REST API calls POST and PUT to send data to the cloud. The data can be pulled by other applications by performing GET requests. The server responds to these GET requests with the data in CSV/JSON/XML formats. The data can be requested as a single Feed which is a snapshot of the current values of each Data-stream in the feed along with the metadata, or as a single Data-stream for specific attributes or for all Feeds up to a maximum of 50 Feeds. The data can also be requested for a particular range of time in both Data-stream level and Feed level.

Security measures: Communications for Xively are based on the standard HTTP(S) connection with SSL/TLS protocol. The API for Xively uses the API Keys assigned to a device on registration for the authentication. Multiple keys can be used for authentication with a different key for the device and the application monitoring it. Access to a device and its data feed requires permission. The API Key defines a set of permissions for the resources it can access/modify and thus, is used for permission to access a feed or a resource. The API key can be fine grained to offer access just to a Feed or a master key to access all types of resources.

The API key is handled differently for accessing the resources using MQTT and HTTP REST calls. When a RESTful call is performed to a resource, the API Key is placed in the HTTP header. On the other hand, for MQTT, the API Key can be used in place of the username eliminating the need for a password. The API Key can also be appended as a prefix to the topic as a mean of authentication. For example, using the following as the topic 'API_KEY/v2/feeds/556.json'.

Xively also offers support for Open Authentication for third party applications to access the URIs updated by the user's devices. The API Keys used for Open Authentication are account-wide keys, i.e. a key is not specific to a feed, it is a master key to all resources in the particular account. Open Authentication is particularly useful when an application is developed to access the resources in an account, however without sharing the username and password for the account. For example, if a temperature feed is generated for sensors deployed in a remote area and the owner wants to share this data with other applications/users, open authentication can be used.

Cost: Xively is a free platform when used for personal purposes, i.e. gathering and analyzing data on a small scale for prototyping, research and experiments. Furthermore, Xively also offers professional services which are aimed at conceiving business oriented potential from an IoT based solution connected to Xively. These services are paid and are aimed at increasing revenue and sales from connected IoT solutions.

Libraries: Xively offers various libraries to facilitate integration of the user code with the Xively API. These libraries for devices and applications are open source and available on GitHub. For devices accessing

the Xively API, libraries are available in C and Python. These libraries offer functions in three layers, the communication layer offering wrappers for networking functions, the transport layer for protocol oriented functions and data layer for decoding the various formats in which the data is sent and received. The HTTP and MQTT communication with the API is built on top of these functions provided by the libraries. For applications accessing and analyzing the data, the libraries are available in JavaScript, ObjectiveC, PHP, Ruby, Python, Android.

Devices supported: Using the libraries in C and Python, Arduino and Android devices can communicate with the Xively API by simply creating instances of classes provided by the library. These classes are defined keeping in mind the resources supported by the Xively API, like Data-Streams, Data-Points and Feeds. Xively also provides support for ARM architectures and mbed devices from ARM through the C library offering functions to interact with the Xively API. The Xively C library can be also used for embedded devices with microcontrollers or system on chip (SoC) to interface the device with the Xively API.

4.1.2 IBM IoT Foundation

IBM Internet of Things Foundation (IBM-IOT) [3] offers a platform for management of data from devices and applications by using MQTT or HTTP(S). IBM IoT Foundation uses the novel concept of an 'organization' to which a set of devices and applications are affiliated; uniquely isolated from other organizations. IBM IoT also offers a QuickStart paradigm to connect a physical device or simulate a device without registering for the service, demonstrating the ease of connecting to IBM IoT.

Protocols used: The protocol used by devices and applications to connect to the IBM IoT Foundation is MQTT which can be handled using the various client libraries, available to communicate with the IBM IoT API. Each organization has its own MQTT endpoint in the form of 'org_id.messaging.internetofthings.ibmcloud.com' accessible by the devices and applications which are part of the organization. MQTT offers both unencrypted connection and encrypted connection to the endpoints. Plaintext communication is used for unencrypted connections whereas encrypted web-sockets optionally use certificates for some libraries.

IBM IoT Foundation also provides options for choosing one of the three Quality of Service parameters to communicate a message between the device and the server, at most once, at least once and exactly once while communicating over MQTT. These options are particularly useful given the nature of the application considering quality of connectivity to the server and energy constraints attached to the devices. Devices subscribed to a topic with MQTT are also provided with the option to store messages with a buffer size of 5000 messages such that the broker can store messages while subscribers are offline or not connected to the broker.

IBM IoT Foundation also offers the choice of using HTTP(S) to communicate with the resources. This is achieved by sending a POST request to a resource, with an Authorization header with the username and password for an Organization. However, for HTTP(S), the Quality of Service is not explicitly available and is left to the user for implementation.

Working principle: IBM IoT primarily functions on registration of devices and applications, followed by communications with the API through HTTP(S) or MQTT. When a device or application is registered, it becomes a part of an organization. Thus, organizations are at the heart of the IBM IoT Foundation to ensure security and data isolation among devices and applications. When a user registers to the IBM IoT Foundation, a 6 character identity is assigned to the user representing an organization. Data within an organization is exclusive and is not exchanged with other organizations.

'Devices' are defined as any 'object' which generates data and is connected to the internet. In MQTT, Devices are identified uniquely by their client IDs in the form of 'd:org_id:device_type:device_id', where 'd' stands for device, followed by the ID of the organization, the device type and the device ID. Applications on the other hand, analyze data generated from multiple devices. They are identified uniquely by their client IDs in the form of 'a:org_id:app_id', where 'a' stands for application, followed by the ID of the organization and the application ID . Devices accumulate data and disseminate them in the form of events published to topics. Devices have two degrees of freedom while generating events, to choose the data to send and the name of the event it is disseminating. Applications acquire these events from the devices by subscribing to topics to which the events were published. From an event, an application can extract the data which was disseminated by the device and recognize the source device with its unique ID which published the event. Applications, on the other hand, communicate with devices by publishing commands to topics, which are in turn subscribed to, by the devices.

Devices and applications can communicate with the IBM IoT foundation by using HTTP as well. In HTTP(S), the publish-subscribe paradigm for devices is emulated by performing a POST request on a predefined URI containing the device_type, device_id and the event_id. The messages sent through HTTP(S) follow QoS level 0, i.e. messages are sent at most once with best effort delivery. The IoT Platform API offers a series of functionalities with standard HTTP verbs for applications. These functionalities offer bulk operations for devices, viewing organizational details, management of devices, connection problem determination among others. The IBM IoT platform also offers a historical storage option in which data from a custom range of time can be stored. This is offered by default for HTTP(S), while QoS level has to be set to zero for MQTT.

When a device registers to the IBM IoT platform, it can choose the 'Manage Device' operation, following which a 'Device Management Agent' becomes a part of the device. IBM IoT Foundation offers a Device Management Service which can be availed by devices which are managed. The Device Management Agent uses the 'IoT Platform Device Management Protocol,' built on top of MQTT, to offer services which include firmware download, factory reset and rebooting. Devices which are void of a Management Agent can also participate in data collection and dissemination, but are not able to avail the above features.

Interaction model: Devices registered to the IBM IoT Foundation can push data gathered, to multiple applications through events. However, events cannot be pushed back into devices, even if the event is from the same device. On the other hand, applications communicate with the devices through commands. However, devices cannot send back commands to applications. Commands and events are differentiated from one another by the resource identifier used to publish them or subscribe to them. For events the identifier is set to 'evt' while for commands it is set to 'cmd'. Applications publish commands to a topic according

the format specified. Devices can subscribe to a subset of commands and can choose how to respond when receiving a command. Subscriptions for devices can be made durable with a parameter 'cleansession' set to 'false' and QoS level set to 1. This allows devices to queue messages sent when the device is not connected.

Security measures: In IBM IoT Foundation, the security is handled at the organization level. Each organization is assigned an API key which is used by the devices and applications registered to the organization. While using encrypted connections through MQTT and HTTPs, IBM IoT uses TLS v1.2 for encryption.

When using MQTT, a device is authenticated by using a username and password. The username is the same for all the devices belonging to the same organization. However, for passwords the devices pass their own authentication tokens as passwords. The use of tokens for authentication is indicated by the string 'use-token-auth'. For applications, the authentication is achieved by passing the client ID, API Key and the authentication token for the application. On the other hand, when using HTTP(S), the username is set to the API key for applications and the password is set to the authentication token. For devices, only the Authentication token is used while the username is set to 'use-token-auth'.

Cost: IBM IoT Foundation offers Quick Connectivity as a trial for free. Services offering a wider range of services including use of multiple devices and applications are priced based on the number of devices required for the application being developed. The services offered start from a bronze product for 100 devices to a gold product for 15000 devices. However, the data storage and data traffic is limited to 1 GB and 100 MB respectively, for all the products.

Libraries: IBM IoT Foundation offers various client libraries to access the API for generating data and management of devices. They offer guides for client libraries in Python, Java, Node.js and C# to develop applications. Libraries for client devices are available in Embedded C and mBed C++ in addition to the above languages.

Devices supported: IBM IoT Foundation offers out of the box connectivity to devices for simple set of applications specified as recipes through QuickStart. Virtual devices are also offered by IBM IoT Foundation to test the recipes and understand how the API works. If devices used do not support the languages in which the libraries are offered, raw MQTT may be used to communicate with the API.

4.1.3 Sense-IoT

Sense-IoT [7]is a platform offering storage of data gathered from sensors and devices. The data gathered can be accessed remotely through REST API and HTTP calls. The stored data can be visualized as well to gather meaningful information from the data. Sense-IoT also offers support for setting triggers allowing active monitoring of the data generated through rules based on thresholds.

Protocols used: The Sense-IoT platform follows the REST API paradigm along with HTTP calls performed to its API for data storage and retrieval. The HTTP requests available for Sense-IoT are GET, PUT, POST, DELETE and OPTIONS. The API for REST calls supports upload of data in two formats namely JSON

and HTML which has to be specified in the HTTP header. However, the data returned by the API is only available in JSON format. Sense-IoT offers an option for compression of data as well, using the standard HTTP compression mechanism in gzip format for sending and retrieving data from the server resources. The compression format for the communication has to be stated in the header as well.

Working principle: The Sense-IoT API is based on performing HTTP requests to interact with the API. The primary requirement to allow interaction between a sensor node or an application with the API is to create an account with Sense-IoT and to sign in with the account. The account sign in phase is executed by sending the user ID and hashed password to the login resource of the API. When signed in, a session ID is assigned to the user. After logging in to the system, the user needs to register a sensor. This can be accomplished by sending the name of the device, the device type describing the metadata and model for the sensor along with a name for display in JSON format. This data is sent to the resource responsible for registering sensors. Once a sensor is registered, the API responds with 201 Created Status, along with a sensor ID included in the header.

The data upload process involves sending the values of data in the form of a string or as a JSON object. When multiple values are sent at a time, the data must be accompanied by a timestamp. If unaccompanied by a timestamp, the API adds the timestamp as the time it received the data. Applications can be used to retrieve data gathered from the sensors specifying the id of the sensor from which the data is gathered. The data is retrieved in the form of 'pages' with each page containing a default of 100 to a maximum of 1000 Data-points. The data retrieval process is based on a few parameters offering the applications flexibility on the number of pages on which the data is to be spread, the start and end dates for the data retrieved along with an option to sort the data according to time. The user may also specify the interval in which the samples of sensor data are spaced in time, e.g. Data-points are available in the interval of 60 seconds incrementally up to 604,800 seconds. This feature is useful to reduce the number of Data-points retrieved, when gathering data over a large interval of time. For example, when data is gathered for 7 days, we can choose a high value of interval so we only get a few Data-points and get an overview of the pattern of data. However, if we require more fine-grained information, the interval can be reduced and we get a large number of Data-points.

Sensors can be used to share data among other users by offering access to its own data. This can be achieved by performing a POST request to the resource 'https://api.sense-os.nl/sensors/id/users' specifying the username or user id of the user; the data is to be shared with. Although other users can access the data from the sensor, only the user who is the owner of the sensor can upload data from a sensor. Sensors can also be made virtual by creating a sensor which physically doesn't exist, however replicates data gathered from other sensors. This can be useful in creating an abstract sensor by combining data sensed from other physical sensors. The Sense-IoT API offers various services as well for the sensors which include 'methods' or 'functions' which can be called to operate upon the data generated. For example, averaging the values received from a sensor over time. When a sensor is associated with a service, a virtual sensor is created as well to store the data received from the associated service. The Sense-IoT API can be also used for data processing using Octave scripts. Octave, a high level interpreted language, can be used to do basic numerical computations and processing on the data acquired from sensors.

Sense-IoT offers the functionality of setting triggers to monitor the values of the sensors. The trigger resource on the API can be called with a PUT or POST to add new triggers or update triggers. A trigger is characterized by an id and an expression which results in the trigger to be activated. For example, 'expression': room_temperature >10. The trigger can be activated as well by overriding the expression by making a POST call to the resource handling the trigger. A trigger can be updated as well using a PUT request with the trigger id specifying the attributes, 'active' stating if the trigger is active, 'last_triggered' and 'times_triggered' specifying the activity of the trigger along with 'last_sensor_data_value' and ' last_sensor_data_date' for the last data measured. These triggers can be used in applications to actuate actions when a sensor trigger is activated. For example, when the temperature of a room goes above a certain value, the actuation can be to start a cooler. A trigger can be activated for maximum once in 60 seconds.

The location of the device can be recorded in an 'environment' resource containing a 'gps-outline' which forms a polygon of latitude-longitude pairs in which the device lies along with an optional additional parameter of altitude. The Sense-IoT API also emulates a closed network of sensors by introducing domains and groups. Domains are defined by the type of data a sensor is gathering and is managed by a 'Domain Manager'. A user needs to get accepted into a domain by the domain manager for the user to be considered part of a domain. On the contrary, groups are formed with an access password which has to be used by a new user to become a part of the group, if the group is private. A user may also become a part of a private group if the user is added by another user who is a part of the group and holds 'add-user' rights. Groups and domains play a major role in classifying the sensors into classes and gather data in a more meaningful manner. The presence of groups and domains also allows updating the properties of multiple sensors at a time, by updating the properties of the group. For example, updating the permissions allowed for devices in a group.

Interaction model: For Sense-IoT, the interaction model is primarily pushing of data from the sensor to the server where the data is stored. This is done in two formats, either using JSON or by using the HTML form along with Content-type headers. Data generated by sensors can be shared among other users as well, who can only view or leverage the data, but cannot upload to the sensor. Applications developed to leverage the data generated from the sensors can pull data using GET calls to the respective resources. Devices which are used as data-sources are not equipped with the ability to pull data from other data-sources. However, with the Raspberry Pi, it can be used as both a data source and also to implement an application, to pull data from the server generated by other sources.

Security measures: The initial authentication of the user trying to upload data or accessing a resource requires sending the user name and hashed MD5 password to the resource for login. Once logged in, the user is assigned a session_id, which can be then further used for communication whenever authentication is necessary by specifying it in the 'X-SESSION_ID' header. Sense-IoT also supports use of Open Authentication to gain access to data generated from sensors using an application. However, to access the data, the application trying to access the data must be registered to the API.

Cost: The services offered by Sense-IoT are classified into three categories based on the pricing, namely Junior, Senior and Pro. These categories are separated by data usage per month, API calls per month and

the length for which data can be stored from the devices. The basic service offered in the form of 'Junior' offers 5MB of data usage for a month, 50,000 API calls per month and data storage for 1 month. The other plans are priced as per data usage and the number of API calls performed while allowing storage of data for a longer time. The API Request rates for GET and POST calls per minute in each type of account are limited as well along with the amount of data that can be uploaded in a month. The Pro account offers 180 POST calls per minute and 480 GET calls per minute along with 2000 MB of data upload per month.

Libraries: Sense-IoT offers libraries for Android and iOS, which can be used to configure the sensors of the device to become data sources. They can be also used to access data stored on the server through the Sense-IoT API. Sense-IoT offers client application libraries in Ruby, Python, JavaScript and PHP to access the Sense-IoT API. The libraries offer function calls for the API methods specified. Arduino devices use the Sense-IoT library offered in C while Raspberry Pi uses the Python client application library offered.

Devices supported: Sense-IoT supports data from devices including Raspberry Pi, Electric Imp and Arduino. The General Purpose Input Output (GPIO) in Raspberry Pi is used to gather data from the sensor and store the output to the API through Python libraries. Electric Imp can be used with an April development board to send data to the Sense-IoT API. Android and IoS devices can be used for generating data from sensors as well. They can be also used with context awareness with the support of Cortex, a mobile intelligence platform and Brain, a Cloud Intelligence platform.

4.1.4 Thingspeak

Thingspeak [11]offers an open data platform and API for Internet of Things to collect, visualize, analyze and act upon data generated by sensors. The platform is primarily based on 'channels', which are used for storing data generated from the sensors. The analysis and visualization of the data generated from the devices can be performed using MATLAB applications which are provided by the Thingspeak platform. Thingspeak also offers 'Thingspeak Apps', which are applications used to leverage the data on the channels to create triggers and interact with social networks and other web services.

Protocols used: Communications in Thingspeak are based on the REST API, used to create Thingspeak channels and retrieve data from channels for analysis and visualization. Using the HTTP verbs POST, PUT and GET, the REST API can be used to create a channel, read data from a channel and update a channel as well. The formats accepted for performing these calls include Text, JSON and XML.

Working principle: The basic building block of Thingspeak is the use of channels, which can be used to store data generated from the devices, location of the devices and the status of devices. The data stored from devices can have up to 8 different fields. For example, a channel can be used to monitor a room, storing temperature and humidity values, light intensity values and air quality measures. The location data for the devices is stored in the form of latitude and longitude in decimal format. The data stored in a channel can be made public or private. The type of channel is stored in the parameter 'public_flag', the form of a boolean where false translates to a private channel and true translates to a public channel. The channel can be also used for storing description of the channel, name of the channel as well as tags and the URL of the channel.

A channel can be created from the interface offered on the Thingspeak website or by performing a 'POST' request to the respective URI. To create a channel, the user creating the channel must pass the API key assigned to the user while creating a Thingspeak account. A user can access the data from a channel by performing a GET request on the URI responsible. A user can also simply view a channel by performing a GET request on the respective URI, to which the API responds with the various parameters of the channel. The properties of a channel can be modified or deleted by performing a PUT and DELETE request on the above URI respectively.

The values taken by the various parameters defined for a channel are defined in a channel feed. For example, if temperature is a parameter for storing data on a channel, the channel feed contains the values of all the parameters defined for the channel, including temperature in this case. A channel feed can be updated by performing a POST or PUT request to the corresponding URI. The channel to which the data is to be written is recognized by the Write API key for the channel. Requests can be performed to acquire a channel feed by performing a GET request with the CHANNEL_ID or to acquire data from a specific field by also specifying the FIELD_ID along with the Read API Key. The time range for the data acquired, the maximum number of values desired and a threshold for the maximum and minimum value for the value of the field can be specified in the body of the request. Basic operations on the data requested can also be obtained including mean, median and sum of all the values requested. For example, we can perform a GET request for channel ID 23, on a field 'light', which is the first field for the channel and request 10 values by making a request to, 'https://api.thingspeak.com/channels/9/fields/1.xml?results=10'.

The server for Thingspeak runs MATLAB applications which can be used to analyze and visualize the data acquired from a channel. MATLAB functions 'thingSpeakRead' and 'thingSpeakWrite' are used to read and write data to a channel respectively. MATLAB visualizations are offered in the form of various kinds of plots including area, scatter and line plots. Thingspeak also offers another set of applications called Thingspeak Apps, which are used to analyze data from channels and visualize the data or trigger an action. For example, if a user sends location data to a Thingspeak channel, this can be leveraged to develop a trigger to turn of lights of the house when the distance of the user from the house is above a certain threshold. Thingspeak also offers the option of viewing the data as charts which can be created by performing a POST request on the URI responsible specifying the data from the channel feed and the parameters for the chart to be used.

Interaction model: The interaction between the client devices and the server occurs through the channel. The device collects the data from its sensors and writes the data to the channel. The server gathers the data generated by the device through the channel and can analyze the data. The data produced as a result of the analysis, is written back on a channel by the server which can then be read by another application to visualize the data or set triggers on the data.

Security measures: When a user registers for Thingspeak, the user is assigned an API key by which the user is recognized while creating a channel. Each channel has its own Read API Key and Write API Key. Based on whether a channel is public or private, the access to the data for a channel is different. To read a private channel, a user who doesn't own the channel needs to pass the Read API Key for the channel.

However, if the channel is public, no key is required to access the channel data. Similarly, to update a channel or its channel feed a user needs to pass the Write API Key of the channel.

Cost: Thingspeak is a free platform for storage, analysis and visualization of data. Thingspeak also offers an open source implementation of their application and API on GitHub, which can be run locally on a server.

Libraries: Thingspeak offers a 'Thingspeak Communication Library' for Arduino and Particle devices. This library allows Arduino and Particle Photon Devices to interact with the Thingspeak API to read and write data from and to the Thingspeak channels.

Devices: Arduino and Particle Photon Devices can leverage the 'Thingspeak Communication Library' and read and write data to the channels. Any other hardware which is also compatible with the library can use it to interact with Thingspeak. Raspberry Pi devices can be used to create channels and send data to channels using Python as a language to write the code to send and receive data. Android devices can be used as well to develop applications which can report the data from the phone's sensors to the Thingspeak channel for storage and analysis.

4.1.5 Ubidots

Ubidots [12] is a platform for storing and analyzing data from devices over the REST API for both small scale applications and enterprises. Ubidots can be used to gather data generated by devices and use this data to create triggers or create visualizations. Ubidots handles the quantity of data by the number of variables or types of data being stored rather than the number of devices connected. Ubidots also offers support for a wide array of devices to send data from using the Ubidots API including Arduino and embedded Linux devices.

Protocols used: Ubidots offers its services based on the REST API using the HTTP methods; PUT to update a resource, POST to create a new resource, GET to retrieve information from a resource and DELETE to delete a resource. The resources for Ubidots are primarily classified into variables, values and data-sources accessible through 3 different URIs. Ubidots also offers the option to perform the API calls using 'curl' which is a command line toll to perform HTTP requests to URIs. Curl can be used to update values of multiple variables at a time as well.

Working principle: The services offered by Ubidots work by performing HTTP method requests using the REST APIs on the various resources defined. These resources are classified into 3 types based on their level of abstraction. They are data-sources, variables and values. Data-sources offer an abstraction of a device which can generate data. For example, an Android device can be a data-source generating data from its sensors. Performing a POST request on the endpoint for data-sources creates a new data-source, while performing a GET request lists all the data-sources. Data-sources generate data which is assigned to variables. For example, a data source generating temperature data can be assigned a variable 'temp' which will accept the temperature values from the data source. The variables have their own IDs, which are used to address them uniquely. Variables are accessible through the corresponding URI specifying VARIABLE_ID, which stands for the unique ID assigned to the variable. A GET request performed on this endpoint without specifyng

the variable ID returns all the variables while when a variable ID is specified details of only that variable is returned. Variables are assigned 'values' and are stored as a part of the variable. Values can be accessed by performing a GET request to the respective API endpoint. To write a value to a variable, a POST request to the particular variable must be made, appending the token to access the variable and the value of the variable to the request. The server responds to the POST request with the value entered and the timestamp at which it was written in JSON format. Multiple values can be written to a variable as well by passing multiple values to a POST request.

The Ubidots REST API offers a few other features including pagination and error handling. When requesting a set of values by performing a GET request, the number of items returned for a request can be changed by changing the value of the parameter 'page_size' which is set by default to 30. When an error occurs when a request is performed, the API responds with an error tag specifying the kind of error that occured. For example, if we try to write a value with a variable ID which is not correct, the API responds with 'Not found'. Ubidots can be used to trigger an action by analyzing the value of multiple variables and set thresholds to trigger an action. An action maybe sending out an email, an SMS or to invoke a WebService when the threshold is breached. The Ubidots REST API also has an endpoint for analyzing the data from variables and generate basic outputs like maximum value, minimum value, mean and variance of the values. This can be achieved by performing a GET request on the endpoint responsible for statistics followed by the type of output required. For example, to get the mean value from a variable, the GET request is to be made to URI specifying 'mean' as the type of value with a start time and an end time.

Interaction model: The interaction model for Ubidots is quite simple. The resources on the server for data sources, variables and values can be modified by posting values of the variables from the data sources themselves. Whereas, applications developed to access the data and analyze them can fetch the values from the resources by performing a GET request on the same resources provided.

Security measures: Authentication in Ubidots is handled using tokens along with an optional HTTPS/SSL layer for security. The tokens are generated based on the API Key of the user. A user can request a token by performing a POST request to the API endpoint URL 'http://things.ubidots.com/api/v1.6/auth/token' appending the user's own API Key in the header for the request. If the user's API key is valid, the endpoint responds with a token which can be used for 6 hours to authenticate REST API calls made in the time-frame.

Cost: Ubidots offers its services in 2 tiers, one for small scale users and startups and another for enterprises. The small scale services tier is offered in 3 differently priced forms. The basic one is offered for free, allowing handling of only 5 sensor variables with real time analytics and alerts. With the 'Individual' services, up to 15 sensor variables can be handled while for 'Startup' based services, 65 sensor variables are offered. In addition to the previous features in the basic service, a Math Engine is offered for Individual and Startup services which allows analysis of the data generated from the devices using mathematical functions. These services also include 500,000 sensor updates and 1 month of historical data storage. For the Enterprise oriented services tier, the services are offered in 3 differently priced forms as well. These services have additional features like different levels of access for different users and creation of sub-accounts for customers with modifiable branding for each customer. In the enterprise services, 1 million sensor updates are allowed

with sensor variables ranging from 180 to a customized value which can be chosen by the client. Hence, the cost for Ubidots is not determined by the number of devices connected but by the number of variables for data the sensors generate. Thus a user generating temperature data, accelerometer data and light sensor data from 3 different devices will be priced same as a user generating the three variables from the same device.

Libraries: Libraries for client applications to interact with the API endpoints are open source and are available in multiple languages including Python, Java, C, PHP, Node, Ruby and LabVIEW. While for devices to be used as data-sources, the libraries are available in C, Python and Java. The libraries offer methods to send and receive values leveraging the Ubidots API and create and handle data sources as well.

Devices: Ubidots offers support for Arduino models, Electric Imp and Particle Photon devices to perform requests on the Ubidots API to post data to variables. Ubidots also offers support for embedded Linux devices including Raspberry Pi and NodeRed as well to simulate applications. Android devices can be used to develop applications to send data using the Ubidots API.

4.1.6 Sparkfun

Sparkfun [9]is a free platform built on top of an open source software called Phant based on node.js for data collection and storage. Sparkfun can be used by using the server side implementation provided by Sparkfun at 'http://data.sparkfun.com' or by implementing an instance of Phant on a server of the user's choice. The platform is also free to use and is limited only by the amount of data that can be stored and by the amount of data that can be pushed into the server from the devices.

Protocols used: Data for the Sparkfun platform is stored on data streams. These data streams can be accessed by performing HTTP requests on the data streams provided by Sparkfun. The API does not entirely follow the REST paradigm, since here GET requests can be used to post and modify data on the data streams, thus providing predominant functionalities for GET and POST. POST requests can be used as well to modify the resources while GET requests are used to fetch data from a resource. Data can be posted and read from the Sparkfun server using the 'curl' module for HTTP methods.

Working principle: To log and store data generated from devices, data streams are used. Data streams are channels on which the data is stored with the following arguments: title, tags, description and fields. The title describes the stream along with a brief overview in the description. The tags are used to categorize streams over Sparkfun. Each stream can contain multiple fields, which are the variables to which the data is actually posted or stored. For example, we can create a data stream for carbon monoxide quantity in air and create a field 'CO_amount'. We can tag the data stream as 'environment' to denote that the data stream is logging environmental data. The data streams can be made available to be viewed by all by setting the stream as public or can be set to private. Data for each field is stored as a string and is limited to 50 Kilobytes. Hence any kind of data that can be converted to a string, can be stored in a field as a string.

Devices can post data to a field in a data stream by performing a GET request or a POST request. When a GET request is performed to post data to a field, the field values are included inline with the URI to which

the request is performed. For example, if we want to set 'temperature' to 32 and the 'speed' of a fan as 4, we can perform a GET request to the URI by appending '&temp=32&speed=4' to the URI responsible for handling the request. When POST method is used to modify data on a data stream, the corresponding URI is used with the data in the body added separately. The PUBLIC_KEY for the data stream is used to uniquely identify the data stream to which the request is being made. The server responds to a request in plaintext, JSON or JSONP formats. The reply is usually in 0 or 1 for failure and success in the plaintext format while in JSON and JSONP, a key-value pair is used to describe a boolean 'Success' as true or false and a 'message' denoting the reason of failure, if any.

Data can be extracted from the data streams by performing a GET query to the following URI, 'http://data.sparkfun.com/output/PUBLIC_KEY.format' where format specifies the format in which the response is requested. The format can be set to JSON, JSONP or CSV. When data is retrieved, all the data is returned by default from the channel. This can be changed by requesting the data in 250 Kilobyte chunks by setting the parameter 'page' to true. If the data is requested in pages, different pages can be requested in different formats by specifying the format for each page number. Data retrieved from the stream can be filtered by adding the following to the query string, 'FILTER[FIELD]=VALUE'. The FILTER here can be set to eq, ne, gt, lt, lte and gte for various equalities and inequalities. For example, we can use 'gte[temp]=40' to filter out values where the temperature is greater than or equal to 40 degrees. Multiple filters can be combined as well to further narrow down the retrieved data.

Sparkfun can be used to retrieve statistics of the current usage for a data stream. By performing a GET request on the following URI, 'http://data.sparkfun.com/output/PUBLIC_KEY/stats.FORMAT', the number of pages of data currently stored, the number of bytes that can be stored furthermore, the number of bytes the stream has already used up and the current maximum amount of data that can be stored on the stream can be retrieved.

Interaction model: The requests made to the resources in Sparkfun can be simply classified into 'input' and 'output'. If the request is an input, the data is being written to the stream by a device while if the request is an output, an application is retrieving data from the data stream.

Security measures: This platform is still in development stages and is awaiting an official release. Thus, the data stored is not entirely secure due to attempts on hacking the data for development purposes. Sparkfun uses the notion of API Keys for authentication. Each data stream is recognized by a PUBLIC_KEY which is required to both read and write data to a data stream. However, for writing to a data stream or modifying a data stream the PRIVATE_KEY is needed as well, which is not required while reading the data. Data on the Sparkfun server is considered public for development purposes, however, data can be made private by hosting one's own phant server.

Cost: The Sparkfun platform is free for the users and is only limited by a set of constraints when using the Sparkfun server implementation of Phant. Each data stream can only store a maximum of 50 MB of data. After this value is surpassed, the API starts to delete values in ascending chronologiocal order.However, if using a local instance of Phant, this constraint can be removed. The logging of data is also limited to a

maximum of 100 data pushes in a 15 minute window. If this limit is surpassed, the server responds with 'HTTP 429: Too many requests'.

Libraries: Most of the libraries offered by Sparkfun are user contributed libraries. Sparkfun offers a phant-arduino client for Arduino devices to make Phant compatible HTTP requests. PietteTech_Phant is a library offered for Spark Core devices which supports writing data to data streams and clearing data streams as well. Other libraries to modify data on the Phant server are available in Python, Rust, Ruby, PHP, node.js, Java and C#.

Devices: Devices supported by Sparkfun include Ethernet shield, Raspberry Pi, BBB, Electric Imp and WiFly. Other devices capable of making HTTP requests using 'curl' on Linux based systems can be used as well to modify data streams on the Phant server.

4.1.7 AWS IoT

AWS IoT [1] is a cloud platform offered by Amazon Web Services (AWS) to facilitate communication among end devices through an AWS message broker on the server. AWS IoT is based on multiple protocols like HTTP and MQTT while also offering support to other customized protocols. AWS IoT offers communication among devices using different protocols as well based on the publish subscribe paradigm. For example, a device communicating with the AWS message broker with MQTT can publish or subscribe to updates from a device using HTTP. AWS IoT also offers the option to use other services from Amazon including Amazon S3 Bucket and Amazon DynamoDB along with the standard AWS IoT services.

Protocols used: AWS IoT supports communication among devices through the AWS endpoints based on HTTP, MQTT or connections using WebSockets. AWS IoT also offers support for other customized and industry standard protocols.One of the primary advantages that AWS IoT offers is the support for communication among devices which are operating based on different protocols. The communication among the devices and the AWS IoT endpoints are handled using the publish-subscribe paradigm, i.e. client devices can publish messages labelled with certain topics while clients interested subscribe to specific topics to receive messages about. This is achieved through a message broker in between the device and the AWS IoT endpoints. The message broker stores a list of clients with their session timings who subscribe for a topic along with a list of topics. The message broker implements MQTT v3.1.1 and supports publishing and subscription to topics.
The message broker implementation is based on MQTT 3.1.1, however with certain differences from the MQTT specifications. Unlike in MQTT where Quality of Service Level 0 (QoS0) stands for delivery of a message at most once, for this implementation, QoS0 stands for delivery for zero or more times. However, messages delivered multiple times may differ in the values of their packet IDs. This implementation of the message broker does not support QoS level 2, which specifies delivery of a message to occur exactly once. The AWS IoT implementation also does not support retained messages, i.e. retainment of the last message published to a topic to be sent to subscribers to the topic in future. This implementation also considers all sessions to be clean sessions, i.e. there is no persistence in the sessions. The message broker for AWS IoT also supports clients connecting to the endpoint using HTTP based on the REST API. The publish-subscribe

paradigm is emulated using the standard HTTP verbs and requests made accordingly. AWS IoT offers support for connecting to clients and applications through a websocket on port 443 using TCP following the MQTT paradigm for communication.

Working principle: The services offered by the AWS IoT is centered around the message broker which is responsible for handling the communication among devices and the AWS-IoT. When a client connects to the message broker, it is assigned a client ID. This client ID is used to identify each client and is passed on to the message broker by the client as a part of the MQTT payload for subsequent communications. However, the message broker does not support connection with multiple clients with the same client ID. Thus, if a client sends a client ID as a part of the payload which is the same as another client ID currently connected, the client that is currently connected is disconnected. Each connected client device is a part of a user account that owns the client device and is a part of an AWS region. The endpoints for each region is unique. Thus topics from different user accounts or regions may share the same name but are treated differently by the message broker.

AWS IoT defines the concept of 'Thing Shadow', which is a JSON document that stores and retrieves the state information for a 'thing'. The thing can be a client device connected to the AWS IoT or can be an application as well. The Thing Shadow can be used to get the state of a thing or to modify the state of a thing. Each thing can have multiple attributes, the states of which are maintained in this document. For example, the state of an LED lighting up a room can be an attribute which has the states ON and OFF. This document contains the following properties, state, metadata, timestamp, clientToken and the version. The state of the thing can be classified into two parts, 'desired' and 'reported'. The desired state of the thing can be set by an application independent of whether the device is connected or not to the AWS IoT at that moment. If the device is not connected, it can set itself to the desired state when it connects back to the AWS IoT. On the other hand, the reported state of the device is the current state of the thing. The thing updates the reported state which is then read by an application to analyze the thing. The metadata offers a fine-grained illustration of the state of the device with the timestamp of the attributes in the state section, i.e. when did the device become active or when the device went offline. Timestamps are used to determine the age of the messages received or the time at which a particular attribute was set in a thing even in the absence of an internal clock for the thing. The 'clientToken' is a string used to identify the response to a corresponding request, since a request and its corresponding response must have the same clientToken. The 'version' states the version of the document which is incremented every time the Thing Shadow is updated. The version is useful to check for the latest version when reading the Thing Shadow.

The Thing Shadow document can be accessed using the HTTP verbs UPDATE, GET and DELETE. The UPDATE request is performed on the respective URI by specifying the thingName which defines the name of the Thing and the endpoint is specified by the region for which the thing is registered. The UPDATE request is used to update an existing document and to create a new one, if the document does not exist. The modifications made to the document are stored here with the timestamp. This is used to set the state of a device to the desired state or to modify the attributes used by an application. Performing a GET request on the URI for the Thing Shadow returns the entire document in JSON format along with the metadata whereas performing a DELETE request removes the document altogether. In case of MQTT, the Thing Shadow is handled using a fixed set of predefined topics where topic_spec defines the specific topic being handled. If

the AWS IoT accepts the change, it publishes a message with the topic appended with the term accepted or with the term rejected if the change is rejected. While requesting the Thing Shadow or updating it, the version number can be specified as well. For AWS IoT the messages are not delivered in the order of their generation. Thus if the message broker receives two messages with different version numbers, the message with an older version number is discarded.

AWS IoT also offers the option of creating triggers based on rules. The addition of a rule is not as intuitive as the other platforms, however the paradigm for adding a rule and choosing parameters for a rule is quite robust. To add a rule, a user must first create a role allowing access to the thing and the data in the form of a trust policy document in JSON format. Following the creation of a role, the user needs to assign AWS IoT the role, in order to allow access to the data for the thing. Once AWS IoT has access to the data, the rules can be created with the following attributes, 'rulename' which adds a unique name to the rule, 'description' to optionally describe what the rule does, an SQL statement to provide the clause for the rule and one or more actions to be taken if the condition for the rule is true. AWS IoT connects the things to a wide range of Amazon services including Amazon S3 for storage, Amazon DynamoDB for databases and others. The actions taken when a rule is triggered leverage these services offered by Amazon. For example, if a rule is satisfied, data is written to an Amazon S3 bucket.

Interaction model: The AWS IoT interaction model is based on reading and writing on the Thing Shadow document for the things, both devices and applications. The applications can write the desired state for the attributes in the Thing Shadow document which allows applications to change the state of an attribute even when the device is offline. Devices on the other hand can set the reported state of an attribute. This reported state can be then read by an application. Since the document is shared and can be written by both an application and the device itself, it allows flexibility for handling the 'things'. For example, an application can write the desired state when the device is offline. When the device comes back online, it can then set the reported state to the desired state. In this way an application can set the state of a device even though the device is offline, when the desired state is set by the application.

Security measures: AWS IoT works with three types of identitiy principals namely, X.509 certificates, Integrated Access Management (IAM) and Amazon Cognito Identities. The type of identity principal chosen depends on the protocol being used. For MQTT, X.509 certificates are used. X.509 certificates offer a significant advantage over standard use of tokens or username-password authentication, since the private key can be stored on the device and need not be sent everytime for authentication. The client authentication is based on TLS 1.2 where the AWS IoT requests the certificate from the client and validates the signature and status on the certificate. The Amazon IoT Command Line Interface (CLI) can be used to create a certificate, revoke an older certificate and transfer certificate among accounts as well. For HTTP requests based on the REST API, the IAM identity or Amazon Cognito identity is used. The IAM identity principal is based on assigning roles to the devices. For an AWS IoT account, a user may create a role equipped with the necessary permissions and assign the role to a device. According to the role assigned to a device, the device can access a service or can publish or subscribe to messages. For example, if assigned a role which allows storage of data on the Amazon DynamoDB table, a device can publish its data to the DynamoDB. Users may be combined to form groups and roles can be assigned to groups of users as a whole. On the other hand, Amazon Cognito

identity allows open authentication by using third party accounts like Facebook and Google to access AWS IoT services.

Following the authentication of the user using one of the above identities, the identities are assigned policies which are used to provide fine-grained permissions to the devices accessing the AWS IoT services. These policies are attached to the X.509 certificates or to the Amazon Cognito identities, whereas for IAM, the policies are assigned in accordance to the roles specified for each client. The AWS IoT policies are stored as JSON documents as name-value pairs of 'Effect', 'Action' and 'Resource'. The Effect is used to specify whether to allow or restrict the mentioned Action on the Resource specified.

Cost: The AWS IoT services are priced by usage, i.e. the user pays according to the number of messages published to the AWS IoT server and the number of messages published by the AWS IoT to the client. Using the free account provided initially by Amazon, 250,000 messages can be exchanged between the AWS IoT and the client devices per month for 12 months. The amount of usage beyond this period is different for each region and is billed per million messages. The cost for publishing a message from each device to AWS IoT and for subscribing to each message per device are calculated separately and then added to calculate the final cost incurred. 512 Byte messages are considered as the standard size of the message and is considered as a unit while paying for the messages. For example, if a device publishes 10 512 Byte messages per hour, addressed to 4 other devices, the user has to pay for 10 messages the device publishes and the 40 messages in total the devices receive, i.e. for 50 messages per hour.

Libraries: Amazon offers Software Development Kits (SDKs) for devices in C and Node.js. These libraries were developed for implementing the security requirements for AWS IoT with the use of certificates and basic publish-subscribe functionalities for the devices. The SDK offers an interface to communicate with the AWS IoT message broker and also update, retrieve and delete Thing Shadows as well. The SDK inherently supports TLS 1.2 authentication but can be also used to support third party crypographic library implementations.

Devices: The libraries offered by AWS IoT are designed to work with constrained devices or with system on a chip having memory of 256 KB or more. They work on embedded Linux devices and other industry standards as well by modifying the wrapper provided by the SDK for the devices. The SDKs are designed and demonstrated for Raspberry Pi and Arduino Yun devices and thus can be used for rapid prototyping. However the SDKs can be used for operating systems other than Embedded Linux as well with a porting guide provided by the AWS IoT, which allows porting the application code or the device code to other devices not based on Linux with minimal modification to the code.

4.1.8 mBed

mBed [4] is an open source platform to connect emBedded devices to the cloud. The primary components of this platform include the mBed operating system, mBed device server, mBed device connector service and the mBed client. mBed offers its own operating system based on which the devices can be programmed. mBed also supports other operating systems, offering services that can be used on top of other operating

systems. mBed offers a library to connect devices to the cloud using the device connector service and a library to implement cryptographic components as well. It also offers an option to port code to other platforms.

Protocols used: The mBed devices implement the Constrained Application (CoAP) protocol to advertise their information which includes, parameters, sensors of the device as well as data generated by the devices. This information is advertised in the form of web resources. The mBed server on the other hand implements a resource directory to which the devices register their resources. The communication between devices and the mBed server is handled by using the REST API verbs over CoAP. For example, The devices register their resources by performing a POST request to the devices register on the web server. The use of REST API over CoAP is particularly useful since the communication can be handled by using a 4 byte binary header over UDP, thus minimizing the overhead for communication.

Working principle: The mBed device server plays a major role in the registration of the devices and the discovery of the resources presented by the devices. The devices initially register their own resources with the device server by performing a POST request to the resource directory of the web server. The device server acts as a middleware between the devices and the web applications. To discover resources registered with the mBed server, a web application can be developed which can perform device and resource discovery based on the lookup web interface provided by the mBed device server. The use of the REST API allows caching of the responses to GET requests with a Max-Age option such that requests can be handled directly from the cache if their age is less than the Max-Age defined.

The web application communicates with the devices using the observe service offered by CoAP. The web applications express interest in a resource offered by a device through a subscription. Based on the subscription, the device publishes the data periodically by aggregating and sending the resources requested. For example, a web application may be interested in receiving the temperature values gathered by a device. The web application registers for observing the temperature value with a token. The device sends back the value of the resource attaching the same token to denote that the value sent is in response to the subscription registered. The device may send the values observed at periodic intervals or may only send the value only when a change is observed. For example, the observed value of temperature can be 24 with a token value 0x8C. The observe service may continue to update the value over an interval like 15 minutes or when the value of temperature changes from 24. However the token value of 0x8C is maintained for all responses.

The device registration and de-registration is done by initializing a Device Server object which specifies the address and security mode for the device server. The data for each device is presented to the server as a resource and are created as resource objects. The resource objects can be static if the value can be initialized only once or dynamic if the value for the resource object changes multiple times. For example, the manufacturer and the serial number for the device can be initialized to static resource objects. The sensor data that requires monitoring from the device server has to be registered as well as a set of dynamic objects. Registration is handled by passing these objects to the device server using the API provided to interact with the device server. When values for the resources registered change, the resources are updated by performing a POST request on the device server. The device server can read the values of a resource using a 'Read' operation on the resource while the client can change the value of a resource by performing a 'Write' oper-

ation overwriting the previous value with a PUT request to the resource. mBed also offers an mBed device connector service to have plug and play connectivity with the cloud for mBed devices. This service is based on the mBed OS offering end to end trust security, efficient data communication and device management.

Interaction model: The interaction model of the mBed platform is centered around the observe service offered over CoAP. The web applications subscribe to notifications for resources that they desire, while the devices push data to the server through the observe service over CoAP. The web applications are also notified by the mBed server when there is a modification in the resource directory of the mBed server. The device server on the other hand can read the values of resources from the devices, in the form of a pull operation on the resources registered by the device.

Security measures: mBed offers a library containing cryptographic components which include public key cryptography, hashing and use of symmetric encryption algorithms. The library supports hashing algorithms like SHA-256, SHA-512 and older MD-5 and MD-4 hashing algorithms. The library also supports standard key exchange methods like Diffie-Hellman-Merkel Key Exchange and algorithms based on elliptic curves. Based on these implementations, mBed offers another library for complete implementation of the TLS and SSL standards in all versions. mBed offers an X.509 certificate handling library, which can be used for parsing of certificates, parsing of the Certificate Revocation List (CRL) and check for the signature chain up to the root for trusted certification authority. The communication of the devices with the server is handled by using an X.509 certificate, private API Key along with the public API key of the server.

Cost: The mBed platform offers free libraries to use over the mBed OS to connect devices to the mBed device server. However, the mBed device connector service is offered in two tiers. The basic tier is offered for free and allows 100 devices and 10,000 events per hour to be observed along with 2 API keys. For using more devices or to publish more events, customized pricing plans are offered by mBed.

Libraries: mBed offers a client device library to connect devices to the mBed device connector service. This library is based on a C++ API to turn any device into an endpoint for the mBed device server. This library uses the CoAP protocol for energy constrained devices and networks including services like Notify and Observe. This library also offers management of devices through registration and de-registration on the device server along with performing write operation on a resource on the device server. The library offered also supports porting of the mBed client to other platforms. mBed also offers standard libraries for implementation of encryption and cryptographic components based on SSL and TLS.

Devices: The mBed library supports applications written on top of the mBed OS and thus a wide range of devices can be supported based on Cortex MCUs on which mBed OS can be installed. Thus the library is more OS specfic than providing support for a specific set of devices. The devices for which the mBed OS has already been tested include devices from Nordic and NXP semiconductors and Silicon Labs.

4.1.9 SicsthSense

SicsthSense [8] is an open source platform for the Internet of Things to accumulate and store data from data sources based on the REST API. The SicsthSense system offers an option to have a Java implementation of the server locally and store data locally as well or use the publicly offered server on the Swedish Institute of Computer Science (SICS) website. It offers a parser to parse data from JSON format and basic mathematical functions to apply on data-sets. Open source libraries are offered as well to implement the parser and functions to access the SicsthSense server.

Protocols used: The SicsthSense system uses the REST API for creation and modification of resources and to post and retrieve data-points. Standard HTTP verbs are used to perform these requests. For example, new resources are created by performing a POST request to the URI responsible for creation of resources while resources are retrieved by performing a GET request on the respective URI. Web-sockets can be used as well to create a persistent connection with the SicsthSense server to retrieve data-points continuously.

Working principle: The SicsthSense system is comprised of entities which are defined in various levels of abstraction. These entities are represented in the JSON format in the SicsthSense system and can be accessed through HTTP requests to the URI responsible for handling the entity. The user is in the highest level of abstraction uniquely identified by the 'username', which is the only mandatory field to be specified. Other parameters for the user including creationDate and lastLogin times are handled internally by the SicsthSense system. A user is in control of a Resource, which has a mandatory field 'label', by which the resource is recognized. The resources can be organized hierarchically through definitions of 'owner_id' and 'parent_id' which are handled by the SicsthSense system. Resources contain a 'Stream' of values which are posted to the resource by a client device. For example, a resource may contain a Stream for temperature values. The payload of the data values being actually posted to a Stream is defined in the JSON format as name-value pairs including the value of the data and the timestamp. The timestamp is optional and is automatically defined by the SicsthSense engine.

The entities on the SicsthSense system can be defined and handled on the publicly available server offered by SicsthSense, in the form of 'http://sense.sics.se' or 'http://presense.sics.se/' or can be defined on an implementation of the server locally as well. A user is created on the SicsthSense by performing a POST request on the URI 'http://HOST:/users' by specifying the username in the body of the POST request. Here HOST corresponds to the public instance or the local implementation in the form of 'localhost'. The server responds to the POST request with the USERID of the new user as a confirmation that the user has been created. The user can then register a resource by performing a POST request with a label for the resource. The SicsthSense server responds with a RESOURCEID as confirmation for the creation of a resource. The USERID and RESOURCE ID are used for posting data to streams and reading data from streams. A stream can be created automatically, by posting data points in JSON format to the respective URI. However, once a user interacts with the resource, the auto-creation of streams cannot be done. Data points can be posted directly to a stream as well with a STREAMID by posting the data points to the respective URI. For example, if a stream is already created on a resource, posting a data point on a new stream would not result in automatic creation of the stream. Data is retrieved by performing a GET request on the same URI, which returns the

last 100 data-points in a JSON representation. However, data can be requested for specific periods by using keywords 'from' or 'until' to limit the period of time or use 'limit' to reduce the number of values to be returned. The data can be requested in CSV format as well. These constraints are added to the end of the URI.

SicsthSense offers another entity called a 'Parser' which is represented in a JSON structure. The Parser is used to define the format in which the data-points are defined for the output Streams on a resource. A new Parser for an output Stream can be defined by performing a POST request on the respective URI. When a Stream is automatically created by posting data-points, the Parser is created automatically as well. When data is posted to a resource, the Parser parses the data from the JSON format and adds them to the Stream. The Parser offers two formatting methods, the first being JSON parsing and the other being pattern matching for regular expressions, which is used to extract data from simple text payloads. Thus, using SicsthSense one can also prefer to send the data in simple text formatting, while the complexity of handling the data is offloaded to the Parser. SicsthSense also offers functions like 'mean', 'median', 'min' and 'max' which can be applied to a Stream and the output of the function is used to populate another Stream. For example, a Stream can be generated by taking the mean of the last 20 data-points of a Stream periodically.

Interaction model: The interaction model for SicsthSense involves creation of a Resource and a Stream to which data-points are posted. The Stream can then be read by performing a POST request to the Stream. SicsthSense offers an option to poll a data source for data. This is achieved by specifying 'polling_period', which represents the interval between two polling requests. The data accumulated from polling is then added to a Stream. SicsthSense also offers another method of getting the updates from a data source, by using Web-sockets. This approach is more granular and immediate than polling, i.e. the connection with the server is persistent and the data values updated on the resource are reported as soon as they are posted to the resource URI.

Security measures: The data that is posted to a Stream is not public by default. Thus, to access data on a Stream, credentials are required in the form of a user-key. The user-key is a randomly generated text String which is available in the user profile. The user-key can be provided simply by appending it to the URI on which the data is being posted or retrieved from and not separately in the body of the request, thus offering a simple mechanism to authenticate a user.

Cost: The SicsthSense system is open source and is offered for local deployment. However, both the publicly available server and the locally deployed server are free for use to send, store and retrieve data from.

Libraries: The SicsthSense engine is offered as a Java implementation based on DropWizard which leverages standard Java frameworks like Jetty and JDBI. SicsthSense also offers a Python library for an implementation of the engine along with scripts to implement the Parser and the mathematical functions offered by the engine. For devices to post data to a Resource, a C library is offered which is based on the Contiki OS along with a Python library. An Android library and a Python library for retrieving and viewing data from a Stream on the SicsthSense server is available as well.

Devices: The SicsthSense system does not provide explicit support for any specific devices. However, it is

compatible to be used with any embedded device which can run the client device library based on Contiki or the Python library.

4.1.10 Microsoft Azure IoT

Azure IoT [2] is a platform for IoT devices offered by Microsoft with a rich protocol set including MQTT, HTTP and AMQP. With this variety of protocols, the platform offers great flexibility in terms of choosing a protocol which is suitable for devices of different specifications. Azure also offers a standard messaging format for accessing the server to allow the ease of use among the various protocols. The libraries offered by Microsoft are also available for multiple languages and operating systems and can be implemented on a wide range of devices.

Protocols used: Azure IoT provides multiple options when it comes to choice of protocols for the application being deployed. The user can choose to interact with the Azure IoT Server using MQTT, HTTP/1, AMQP or AMQP over Web Sockets. However, there are certain trade-offs while choosing the protocols for different applications. While using HTTP/1, the server push function is not available, i.e. the cloud cannot push data to devices. The devices are required to poll the server for new messages. This is an issue with resource constrained devices, since the server has to be polled periodically to receive messages. On the other hand, MQTT and AMQP can be used to push data directly to the device from the cloud. While using MQTT, Azure IoT only offers quality of service upto QoS1, i.e. at least once. When QoS2, i.e. delivery exactly once is requested either only QoS1 is offered or the request is rejected. The use of 'Retain' message is not handled by the broker directly, the messages with the Retain flag set to true is passed on to the backend application. For example, if a client device publishes a message with the Retain flag set to True, the broker is supposed to retain the message and pass it on to newly subscribed clients. However, for Azure IoT, this message is not handled by the broker, but is passed on to a backend application which retains the message. AMQP and MQTT are binary protocols, thus are more compact than HTTP/1. However, AMQP is resource intensive and is thus preferred only for devices with sufficient resources. For resource constrained devices, MQTT is chosen, while if network traversal and configuration prevents use of AMQP and MQTT, HTTP/1 should be used.

Working principle: Azure IoT offers a set of endpoints to expose various functionalities in the system. The 'Resource Provider' endpoint provides the highest abstraction for the user, used to create, update and delete IoT hubs and their properties. Each hub comprises of a list of devices, the identities of which can be handled using the 'Device Identity Management' endpoint. The HTTP REST verbs are used to create, read and update the list of device identities called 'Device Identity Registry'. The identity for each device includes a DeviceID, which identifies a device, the status of the device, if enabled or disabled and the connectionState of the device, if connected or disconnected based on pings in only MQTT and AMQP. If DeviceID for two devices match, they are differentiated by GenerationID, which is generated based on when the device has been created or deleted. Using the operations exposed by the endpoint, the identity registry can be exported to a text file in JSON format with the metadata of each device and can be imported as well from a text file. Using the deviceID as the key, devices can be created, updated and deleted. The entire list of devices in the registry can be retrieved as well.

Furthermore, the Azure IoT offers two other kinds of endpoints for handling messaging from the device to the cloud and from the cloud to the device. The first kind of endpoint called 'Device Endpoints' handles sending of messages from the device to the cloud and receiving messages from the cloud to the device. On the other hand, the second kind of endpoint called 'Service Endpoints' allows backend applications to read device to cloud messages and also send cloud to device messages with acknowledgements. The 'Device Endpoints' for messaging are exposed through HTTP, MQTT, AMQP and also over Websockets while the 'Service Endpoints' are available only over AMQP.

The Azure IoT supports multiple protocols for the services they offer including HTTP, MQTT, AMQP and AMQP over Websockets. Thus to offer interoperability among these protocols, Azure IoT offers a standard format for messages which includes a set of properties. These properties include the MessageID uniquely identifying the message and a userID to define the origin of the message. For messages sent from the devices to the cloud, the DeviceID and the GenerationID are specified in the message as well to uniquely identify the device sending the message. For cloud to device messages, a sequence number is assigned to each message with a 'To' field which is used to specify the device to which the message is being currently sent. For cloud to device messages, Azure IoT offers an option as well to receive acknowledgements for the messages consumed by the device. The response to a message maybe none, by default, only positive, only negative or both, i.e. the device responds in both cases that it receives the message and when it fails to receive the message.

Interaction model: The interaction model for the Azure IoT is presented differently for applications in the backend and devices. The devices can publish data to the cloud, i.e. can report sensor values and its state to the cloud and can receive messages from the cloud to the device. While using HTTP/1 the client device polls the server for messages to be sent from cloud to the device. On the other hand, the backend applications can read the messages sent from the device to the cloud and can also send new messages from the cloud to the device. When the backend application sends cloud to device messages, an option is offered to receive the delivery acknowledgements or expiration acknowledgements regarding the message sent.

Security measures: The authorization to access resources and endpoints are handled in Azure IoT using 'Shared Access Signature' (SAS) security tokens. These tokens offer time bound access and prevents sending the keys for communication. The security token can be signed using a shared access policy key or a symmetric key stored with a device identity. To understand how each of the keys work, we must consider the fact that the Azure IoT controls access to the endpoints offered through a set of permissions. These permissions include RegistryRead and RegistryReadWrite to read and also write to the Device Identity Registry respectively. ServiceConnect offers the permission to backend applications to utilize the Service Endpoint while DeviceConnect offers the permission to devices to utilize the Device Endpoint. These access policies can be defined specific to a device using the Device Identity specified in each hub or can be shared by the entire hub of devices. For example, only a particular device maybe allowed access to read from the registry by specifying the DeviceID in the access policy or all devices of the hub can be allowed to access the device registry. The permissions can be offered together in the form of a bundle or individually. For example, the IoT Hub Owner has a policy which includes all the permissions while a device may only have the DeviceConnect permission and RegistryReadWrite permission defined separately.

When a token is signed with a shared access policy key, the permissions included in the policy are assigned to all the devices in the hub. However, when a user wants to assign only the DeviceConnect permission for a specific device, the token is signed with a symmetric key associated with the device identity. The security tokens used have the parameters of expiry specifying how long the token grants access, a policyName to identify the policy. These tokens are utilized differently when using different protocols. When using HTTP, authentication is handled by including a valid token in the Authorization request header. On the other hand while using MQTT, the CONNECT packet is used to send the ClientID set as the DeviceID for the username and the SAS token is assigned to the password field.

Cost: The Azure IoT services are priced based on the number of messages transmitted from the client devices per day. The free service offered initially for a month allows the user to send 8,000 messages per day. The next tier of service S1, offers 400,000 messages to be transmitted per day while the highest tier of service offers transmission of 6 million messages per day. The services are priced differently based on the region the services are accessed from. For example, the pricing for the tiers S1 and S2 are $50 and $500 per month for the East United States(US) region.

Libraries: The Azure IoT libraries for client devices are offered in C, Node.js, Java, .Net and Python. The primary functions offered by the client device libraries include sending data to the Azure IoT server, data serialization into JSON format, mapping server commands to device functions and buffering data while connection to the server is not available. They also offer communication with the server in the aforementioned protocols. The libraries to develop applications in the backend for offering services are available in .Net, Node.js and Java. These libraries primarily provide the interfaces to handle the identity of the client devices and to send cloud to device messages to the client devices with acknowledgement.

Devices: The libraries offered for devices have been tested on multiple platforms for devices including various forms of Linux, mbed OS, Windows and Texas Instruments Real Time OS (TI-RTOS). The libraries have also been tested on a wide range of devices including Arduino, Beaglebone, Raspberry Pi, Libelium and Intel Edison. A complete list of tested devices are available in the documentation provided by the Azure IoT.

4.1.11 Parse

Parse [5] is an open source platform for gathering, processing and storing data to the Cloud. Parse initially offered their services on their own server, however, after discontinuation of their services, the server is available for local deployment. Parse uses the REST API to send and receive information from the cloud. Data is handled in the form of objects which can be sent, read and modified on the server. Security in Parse is offered in multiple layers for classes and objects along with open authentication on the Parse server.

Protocols used: Parse offers its services leveraging the REST API and the standard HTTP verbs. Since the service of Parse is getting discontinued, the HTTP requests are to be made to the local URI of the hosted Parse server instead of the one hosted by Parse on 'https://api.parse.com'. The body for requests made with PUT and POST require the body to be in JSON format while the response from the server is a JSON object.

Working principle: The Parse server can be hosted locally based on Express, a framework for node.js. Any infrastructure capable of running node.js can be used to host the Parse server locally. When hosting the Parse server, the following parameters need to be specified. The Application ID and Master Key are two arbitrary strings passed as parameters while creating the server which are used by client devices to authenticate with the Parse server. Parse server also uses MongoDB as the database to store data. The database can be specified by passing the URI for the database. There are parameters that can be passed while instantiating the server, the port at which the server is to be hosted, the serverURL and configurations for cloud code and push. The server implementation also offers a few other advanced parameters to specify if anonymous users can be allowed, the maximum session length, the maximum upload size for the server and configuration for open authentication among others.

Data is stored on the Parse server by creating a JSON object. The object does not have a specific schema and can be defined with any key-value pairs. A new object can be created by performing a POST request with the JSON body to the URL responsible for creating objects. Each object has a class name which can be used to differentiate among the data stored on different objects. For example, we can create an object storing temperature data and assign it to a class named 'TempData'. The class defined has its own URI. On successful creation of an object the server replies with a 201 Created status with keys 'createdAt' and 'objectID'. The objectID returned by the server on creation of an object, can be used later to retrieve the object by performing a GET request to the URI of the class. The GET request can be made specifically for a particular objectID or for the entire class of objects. For example, if the TempData class is requested, all the objects belonging to the class are returned, while if only a particular objectID is requested, only that object is returned from the TempData class. The retrieved objects are returned enclosed in a field in the JSON object. The data stored in an object can be modified by performing a PUT request on the class URI, specifying the keys for whom the values we want to change. The values for the keys not specified in the request, remain unchanged. The server responds to an update with 'updatedAt' key and the time at which the value was updated. Objects can be deleted by performing a DELETE request on the object. Parse also offers a batch operation to create, update and delete up to 50 objects at a time.

Users are also similar to objects, however they require username and password as mandatory fields to the object. When a user is created, a sessionToken is also returned along with the createdAt and objectID keys. A user can then login by sending a GET request to the login endpoint of Parse with the username and password. When the user logs in, a ParseSession object is created automatically while the server responds to the request with the username, createdAt, updatedAt, objectID and sessionToken key-value pairs. The ParseSession contains a sessionToken to authenticate Parse API requests, the ParseUser object corresponding to the user logging in, a createdWith object specifying whether the user logged in or signed up and the authenticator for the session, whether using a password or open authentication. ParseSession also contains an installationID which maps the user to the device from which the user is logged in from and an expiresAt key which specifies the time, approximately at which the ParseSession object will be automatically deleted. ParseSession objects can be also created, queried and deleted manually by performing requests to the respective URI.

Parse offers a set of data types for the objects which include an array, where an array of objects are considered together. Objects can be added to or removed from the array. If data in the object is numerical, it can be incremented or decremented using the API from Parse when updating the object. When retrieving an object or a set of objects belonging to a class, the GET request can be made with parameters specified in a where statement. For example, retrieve an object where the value of temperature >10 in that object. Various mathematical operators including inequalities and relational operators like 'select' and 'exists' are offered to specify the constraints. The number of objects returned can be limited by setting the key 'limit' to a numerical value. For example, if limit is set to 10, only 10 objects are returned. Users can also place functions in the cloud in the form of cloud code and call the function to analyze the data received on the cloud using the function. For example, a function can be written on the cloud to find the maximum value among the set of received values and return the maximum value.

Parse offers another data type in the form of relations which allows objects to be associated with each other. The relations are of three types, one-to-one, one-to-many and many-to-many. One-to-one relationships are most common when one object is associated with exactly one other object. This relation is implemented when splitting one object to two to limit access on each part of the object or if an object is larger than 128KB in size. For example, if we want to make data beyond a certain timestamp public and data before the timestamp private to the user, we can split the object into two objects using one-to-one relations. When one object is associated with multiple objects, one-to-many relation is used. For example, temperature data from multiple sensors are stored in a single temperature object. This relation is handled in two ways. When the number of associated objects are high, pointers are used to point to the objects while when the number of associated objects is low arrays are used. Many-to-many relations are used when there are multiple objects being associated with multiple objects. For example, we want to associate temperature data and humidity data in a weather object. Each site may have multiple sensors which act as sources of temperature and sensor data. Thus, this association of multiple objects is handled through many-to-many relations.

Parse allows uploading of files to the Parse server by performing a POST request to the respective URI followed by the name of the file being created. The server responds with a Created status and the name and location of the file. An object can be associated with a location as well, using the GeoPoint datatype which takes the latitude and longitude as parameters. Queries to objects can be made with constraints according to the location. For example, selecting 10 objects nearest to the current object. Objects do not have any predefined schemas, however, users can create specific schemas for their applications which the objects must abide by. A schema can be added by performing a POST request to the respective URI followed by the name of the schema.

Interaction model: Applications based on Parse can be installed on devices of various kinds, like Android, iOS and embedded devices. When an application is installed on a device, an installation object is created which represents an instance of the application being installed on the device. The Universally Unique Identifier (UUID) for the device is used as the InstallationID, which uniquely identifies an application being installed on a device. Devices can communicate with the Parse server by updating the objects on the Parse server created by the user owning the device and also by retrieving objects from the Parse server.

The interaction model for Parse also offers a publication and subscription paradigm based on either a channel or advanced targeting. A channel is identified by a string and can be subscribed to by the devices. For example, to get updated humidity data from a building, a device can subscribe to a channel 'Humidity_Building'. The subscription is stored in the 'channels' field of the Installation object. When data is published or pushed to a channel, the subscribers receive an alert message regarding the data pushed. Advanced targeting allows pushing data to a refined set of devices, pushing the data only to some of the devices from the Installation objects of the application.

Security measures: Parse uses HTTPS and SSL for all communications and rejects non-HTTP connections to eliminate man in the middle attacks. The authentication for Parse applications are different for the client and the server. When an application first connects to Parse, it passes the Application ID and optionally the client key. Depending on the platform being used, another key is assigned to the clients called the client key or REST API key or .NET key or JavaScript Key. The client key is not a necessary requirement by default. However, when the Parse server is configured, the requirement of the client key can me made necessary, making it mandatory for the clients to pass one of the four keys.

The security for Parse begins at the class level for the objects. By default, a client application can create new classes, add new fields and modify or query objects on Parse. However, a user owning an application can specify the permission for each client to handle the classes. A client can have Get permission, which allows the client to fetch an object if objectID is known or the client can have Find permission which allows the client to query all the objects in the class without knowing the specific objectID. A client can have permission to create an object in a class, delete objects in a class or add fields to the schemas specified during object creation. The next level of security offered by Parse is object level access control. Data owned by users in the form of objects can be made readable or writeable by using Access Control Lists (ACLs) on the objects. Each object may have different read and write permissions specified on the ACL for the object. If an object does not have an ACL, it is readable and writeable by all users. The ACL may also specify a group of users by leveraging another object type called 'roles'. Roles can be assigned users in the form of a group and the role of the group is specified on the ACL. However, only a user with a role of 'Admin' can assign roles to other users. For example, we can assign a group of users a role of 'monitor' which can read objects containing data from multiple sensors monitoring a building. Access control can be applied to a class as well by using pointer permissions. Pointer permissions are a special kind of class level permission which access to all the objects in a class based on a user specified in the class with a role. For example, if a user is the 'owner' of a class, a pointer permission can be used to make all objects in the class readable to the owner.

Parse Server handles authentication through username and password from the registered user or through open authentication as well. The open authentication is offered for third party credentials including Google, Twitter and Facebook. The open authentication is supported by configuring the server with the 'oauth' option. Other custom authentication providers can be supported as well by specifying the path to the module for the authentication provider and implementing the interface to validate data for authentication.

Cost: Parse was offered with different pricing plans when it was initially introduced. However, after the support on 'https://api.parse.com' has been discontinued, the Parser server is now open source and can be

implemented locally.

Libraries: Parse offers libraries for Android, iOS and OS X to develop applications based on the backend solutions offered by the Parse server. Libraries are also available in PHP and JavaScript to develop applications for devices. Embedded C libraries are offered for Arduino and Raspberry Pi devices. The server for Parse is also available for open source implementation and various configurations are offered for the server.

Devices: The SDKs offered by Parse are available for multiple platforms and devices. The mobile SDKs are offered for Android, iOS and OS X for Android and Apple devices. Parse also supports Arduino devices including Arduino Yun, Raspberry Pi along with UNIX-based and Real Time Operating System (RTOS) based devices.

4.1.12 ThingPlus

ThingPlus [10] is a platform to gather data from sensors and devices and analyze them on the cloud. The platform offers creation of services to handle the gathered and perform actions based on the data received on the cloud. ThingPlus requires Gateways between devices and the server, which collect data from the devices and report them to the server. ThingPlus offers a closed package for implementing their services on the devices and gateways and support for various kinds of sensors.

Protocols used: ThingPlus uses REST API for accessing the services offered and the data stored on the cloud. The requests are made using the standard HTTP verbs over HTTPS. The data for the requests and responses are handled in the JSON format.

Working principle: ThingPlus offers creation of services to gather and monitor data generated from devices. Services can be named by the user creating it, which results in creation of a URL for the service itself. The data being monitored using a service is categorized into sites, i.e. each service comprises of one or more sites. For example, if a building is being monitored for temperature and humidity, the service can be called 'buildingmonitor' while the sites for the service may be 3, one site for each floor of the building. Each service has a manager which has authorization to handle the entire service while the manager for sites only has authorization to handle its own site. A user, on the other hand, can be registered to the ThingPlus service, can be granted access by the site or service manager.

The data generators for the ThingPlus platforms are defined in three levels of abstraction. The gateway is the highest level of abstraction which connects to the ThingPlus cloud. The gateway communicates with the connected devices, which is the next level of abstraction. The resource for the gateways consist of parameters including name, ctime and mtime specifying creation time and last modification time respectively, along with the list of IDs of devices and sensors which are part of the gateway. The gateway can be virtual as well, which can be identified by a Boolean 'virtual' defined in the resource responsible for the gateway. The devices have sensors, which is the lowest level of abstraction. The sensors collect the data and report to the devices, which are furthermore sent to the ThingPlus cloud through the gateway. For example, a Raspberry Pi can be used as a gateway, which gets its data from an Arduino device using a humidity sensor to collect data. The

resource for sensors consists of a name, deviceID specifying the ID of the device the sensor belongs to, a type specifying what kind of data the sensor gathers, an address along with other parameters like model, sequence and category. The components in the abstraction level of sensor may also be an actuator which can be used to take an action. For example, a set of LEDs can act as an actuator, which can be activated when the data from the light sensor in the environment being monitored gets too low. Sensors can be grouped together to form a tag. For example, three light sensors from three different sites can be grouped together to form a tag. Data from a tag can be viewed together on the ThingPlus interface for visualizing and monitoring data.

ThingPlus can be used to analyze the data and create a trigger to perform an action when a certain condition or multiple conditions are met in the form of 'rules'. The rules are defined in the form of Trigger-Condition-Action. For the trigger, the type of trigger is chosen, i.e. whether to trigger based on data from the gateway, device or the sensor. The condition specifies an expression or multiple expressions to be met for the action to be triggered. The light reading<100 and the timestamp suggesting that the time of the day is night can be considered a condition for the rule. If the condition is satisfied the action is triggered, which is handled by an actuator at the sensor level of abstraction. There can be cases where the condition is met too frequently or the condition is for an emergency where an accidental trigger can result in an actuation which is costly. For example, a trigger for sprinkling water if the temperature in a room is too high and the carbon dioxide level in the room is high. Thus to prevent these occurrences, the rules can be set to trigger only if the condition is met for a certain period of time, for example if the temperature and carbon dioxide level is too high persistently then the action is performed. On the other hand, to prevent frequent occurrences, a rule trigger can be ignored if the same device or gateway triggers in the rule within a short span of time.

Data is requested from the ThingPlus server by performing a GET request to the URI responsible. The requests are performed to the base URI followed by the component we are querying. For example, to get a list of gateways, a GET request is performed on 'https://api.thingplus.net/v1/gateways'. The response is in JSON format with the data in key value pairs. The request can be further refined by adding parameters like count, specifying number of items to retrieve, the pagination number for the response along with filters and ordering options for the data. When retrieving data from a sensor or device, data can be requested serially specifying the start and end date between which the data is requested or by specifying the interval for the data. Gateways cannot be created yet using the REST API offered by ThingPlus, however, sensors can be created by performing a POST request to the URI responsible by specifying the parameters required for the sensor which include, a name, the device type, the driver name, the model and the network to which the sensor belongs. The data for a sensor or a gateway, can be updated by performing a PUT request on the respective URI. Resources can be deleted by performing a DELETE request on the URI for the resource. ThingPlus also allows creation, modification and deletion of tags through the REST API.

Interaction model: The data gathered on the ThingPlus platform is handled by gateways, devices and sensors. The data is stored on the ThingPlus cloud storage and can be accessed by the gateways and devices. The data can be accessed by applications as well, which have the permission to access the data. The data gathered can be analyzed and viewed from the web interface provided by ThingPlus. The web interface can be also used to manage the gateways and devices directly.

Security measures: ThingPlus uses OAuth2 protocol leveraging Open Authentication for registration of applications and authentication of the users. When an application is registered using Open Authentication, it is assigned a unique ClientID and a Client Secret string. When a user tries to access an application, the user has to pass the ClientID while performing a GET request to the URI 'https://api.thingplus.net/v1/oauth2/authorize'. Along with the ClientID, the user also has to specify the URI the of the application for which the authentication is required, for redirecting the user to the application once the user is authenticated. Authorization is handled in two ways, by code or by token. When the user is authorized by code, the authorization code is passed to the URI for the application while a token is passed to the application for authorization when the user is authorized by token. When a user is authorized by code, the session is valid for 10 minutes, while authorization by token leads to a validity for 15 days. The authorization code can be used to exchange the code for a token by performing a POST request to the responsible URI with the authorization code, ClientID and Client Secret string in the body of the request. The type of access, whether read or write for the authorization tokens is limited by scopes. Scopes are defined to specify the resource which the user can access and the type of access the user has. For example, a scope can specify that a user can only read data from the gateway but cannot write to or update the gateway.

Cost: ThingPlus offers their services with two options in terms of pricing. Their basic services for personal use are offered for free where the user can add up to 20 sensors and create up to 5 rules to analyze data. On the other hand, pricing for business oriented solutions on a larger scale is not specified. The user needs to contact ThingPlus for customized pricing of the solutions.

Libraries: ThingPlus offers an embedded package for devices and gateways which is downloaded to the respective device or gateway by running a script. The script names are different for different devices or gateways to which the package is installed.

Devices: The embedded package can be installed on Android devices, Arduino and Edison modules, Raspberry Pi and Beaglebone devices. All devices can be used both as devices to report data or as gateways. The package for the devices are designed for Grove systems which are ready to use. However, other devices and gateways can be registered as well by specifying the MAC Address and model of the device on the gateway registration page of ThingPlus.

4.1.13 Other Platforms for IoT

We have chosen the above platforms based on the criteria that the platforms offered are open-source, i.e. we can look into how the platforms work and how the devices and applications interact with the server based on the API provided. We have also considered platforms where the platforms are either offered as Platform-as-a-Service (PaaS) or Software-as-a-Service (SaaS) for the IoT devices and offer libraries for the devices to work on, for various operating systems and languages. Other than these platforms, we have also come across other platforms which are not suitable for our analysis, based on the above mentioned criteria.

There are several platforms offering services for business oriented IoT solutions and Industrial IoT solutions. 'Devicify' offers their services as SaaS, building and designing IoT applications for handling product manage-

ment. Devicify treats physical products as devices going through the various processes including production, invoicing and logistics. Devicify offers their own software which can be used to register products for the services offered by Devicify. 'Solvver' offers a framework with cloud services to accumulate and analyze data in bulk. The Solvver engine offers mathematical functions and signal processing functions to act on data generated from devices and to transform the raw data into meaningful data. Solvver also offers a platform for both static and dynamic Structural Health Monitoring and helps design solutions for the civil engineering sector. 'Thingworx' offers services to gather and mashup data from different devices. They offer their own applications to mashup data and a composer to build end to end applications for devices. For example, data from a humidity sensor and a temperature sensor can be mashed to predict rainfall in a weather station application. However, the software is proprietary in nature, hence we did not consider it for our study in the previous section. The solutions from Thingworx, Devicify and Solvver are offered as SaaS and cannot be delved into, to determine how the services are actually functioning.

'Element Blue' is a platform offering enterprise solutions in Energy, Healthcare and Manufacturing as well as other interdisciplinary fields to optimize infrastructure and services. For example, it offers a service to optimize the toll prices on US freeways based on the amount of traffic as a smart transportation solution. 'IoTSens' is another platform offering Smart City as a Service where data is gathered from a city and analyzed to take certain actions accordingly. For example, an application developed by IoTSens monitors the water flow and supply to a city and helps improve accuracy and precision of the water supply. Both Element Blue and IoTSens offer their services as Infrastructure as a Service (IaaS), i.e. they handle the design of a solution and offer the software as well as hardware for the solutions offered. For example for Smart Metering, IoTSens offers software services which generate heat maps of the sensors, alarms and notifications as well as historical analysis of the data. They also offer the devices and handle the communication among them based on sensing hubs like WmBus leveraging multiple communication protocols. 'Beyond Scada' is also another platform offering IaaS, for data accumulation and visualization based on data gathered from IoT devices. They also offer their own systems to connect to various kinds of hardware devices for Smart City and IoT applications. However, similar to the above platforms, their systems are closed as well.

4.2 Discussion

In this section we summarize the features of the cloud platforms discussed in the section 4.1. For each attribute we discuss about the options available on different cloud platforms and present a trend among the cloud platforms for the Internet of Things.

Protocols used: In accordance to the types of protocols defined in section 2.3, the most widely used request-response protocol is HTTP while the most widely used message passing protocol is MQTT. Protcols CoAP and AMQP are used only in platforms mBed and Azure IoT respectively. The proprietary platforms offer support for multiple protocols and thus, they support multiple end-points for these different protocols. On the other hand, open source platforms are primarily based on HTTP with the exception of SicsthSense which also supports MQTT.

Working principle: The working principle of most of the cloud platforms are based on three functions,

receiving data from the devices, storing data on the platforms and processing the stored data. Feature-rich proprietary platforms like AWS IoT, IBM IoT and Azure IoT which support multiple protocols for receiving data from the devices, have different modules for handling each of these functions. Thus, the data is received on the module responsible for communicating with the devices where the data is stored temporarily and the user has a choice of how to handle the data. These platforms also offer processing modules which support complex processing operations like machine learning and pattern recognition on large sets of data. On the other hand, on proprietary platforms like Xively, Ubidots, SenseIoT, data is stored as soon as the data is received on the platform and the underlying complexity of having to store the data is hidden from the users. For open-source platforms like Phant, ThingSpeak and Parse, the data is also stored when it is received. However, the data is stored to a file or database defined by the user.

Interaction model: The interaction model of most of the platforms are skewed towards unidirectional device-to-cloud communication where the device initiates communication with the server to request for a service or data or to upload data. The number of platforms supporting cloud-to-device communication to send messages from the server to the device to change state on the device is few in number. mBed with the CoAP server deployed on the devices supports cloud-to-device messaging. Azure IoT supports messaging from the cloud platform to the devices with libraries on the client devices offering methods to handle cloud-to-device messaging while AWS IoT handles client device state from the cloud platform using the distributed sharing model.

Security: We discuss the security measures of the platform in two parts, the authentication measures and the authorization measures. In terms of authentication measures, the use of API keys to authenticate devices is quite prominent among the platforms like Phant, Xively, ThingSpeak and Ubidots, since they are easier to handle on constrained devices. The use of certificates is also prominent, mostly for proprietary platforms like Azure IoT, AWS IoT and IBM IoT. Most of the platforms also support multiple authentication methods to support a wider array of client devices. Authorization methods used on the platforms are primarily based on policies and user groups to offer permission to access a subset of resources stored on the platform to specific user groups. These kind of authorization methods are supported on Parse, IBM IoT, AWS IoT and Azure IoT. On some of the platforms, resources are access-restricted and require authentication to access. In these cases authorization is not required. Phant, Xively and Ubidots are examples of platforms which follow this security model.

Cost: The cost involved for the cloud platforms are diverse in nature and are based on different pricing models for different platforms. However, the common feature among the proprietary platforms is that they offer a free tier of service which allows the user to build prototypes and test them before opting for higher priced solutions. The pricing models are based on different modules like data storage, data processing and data visualization. Pricing models may be based on multiple parameters at the same time or on a single parameter. For example, platforms like AWS IoT and IBM IoT charge users based on the number of messages exchanged, the amount of data stored as well as the amount of data processed on the platform beyond the free tiers. On the other hand, SenseIoT charges the user based only on the number of messages exchanged with the server, while Ubidots charges the user based on the number of variables visualized per device beyond the free tier. Different cloud platforms may have pricing models based on the same parameter but with different

metrics. For example, AWS IoT and IBM IoT charge users based on the exchange of messages between the device and the platform. However, AWS IoT charges the user based on the number of messages while IBM IoT charges the user based on the amount of data exchanged on the messages.

Libraries: The platforms offer libraries primarily of two types, libraries for devices and libraries for building applications on platforms. Most of the platforms offer their libraries for devices in C and EmbeddedC to reach out to a wide range of devices, since most devices support programming in C. The other languages popular for libraries offered for devices are Python and Java Script. On the other hand, libraries for building applications are offered for multiple operating systems and in different languages. JavaScript is one of the most popular languages based on which these libraries are offered owing to the ease-of-use for JavaScript to build web services. Other prominent languages for application libraries include PHP and Python. Popoular operating systems supported by cloud platforms for building the applications include Android and iOS.

Devices supported: Cloud platforms usually offer tutorials and quickstart guides for certain devices and present specifications for other devices to satisfy in order to communicate with the platform. Most platforms offer support for Raspberry Pi and Arduino devices. The support for other devices are specified differently in case of different platforms. Some of the platforms like mBed support devices based on their operating systems while some of the platforms like Phant support devices based on their ability to run a particular programming language framework like Node.js.

Chapter 5

Experiences and lessons learned

In the previous chapters, we have studied various cloud platforms for the Internet of Things along with features based on which these platforms can be compared. The choice to be made for a cloud platform varies based on the kind of use case being considered and the tradeoffs involved for the same. In this chapter, we discuss such a use case in detail. Based on our use case, we select 4 cloud platforms from the cloud platforms discussed in chapter 4 for a comprehensive study. We discuss the challenges encountered in implementing an end to end solution; from getting data from the edge devices to sending and processing the data on the cloud platform. We perform an in-depth comparison among the four cloud platforms we have chosen for the use case by sending and processing data on each of these platforms.

5.1 Selection of cloud platforms

In the previous chapter we have discussed about some of the major platforms offering Platform-as-a-Service (PaaS) for the Internet of Things. Keeping our use-case scenario in mind, which we describe in detail in section 5.2, we have chosen a subset of these platforms to perform an in-depth comparison among them. While making this choice among all the platforms, we have tried to consider diversity of the features among the platforms, such that we can study and cover most of the protocols, authorization and authentication measures, pricing models and libraries discussed in chapter 3. The rationale behind our choices for the platforms are discussed below.

Amazon AWS IoT: We chose Amazon AWS IoT as one of the proprietary platforms to conduct our comparison among the platforms. Amazon AWS IoT offers connectivity to their platform from devices and gateways through a diverse set of protocols like MQTT, HTTP and web sockets. One of the major reasons for choosing Amazon AWS is the rule-based trigger approach for the data that is accumulated on the platform, i.e. the data that is received on the platform can be sent to multiple modules at a time based on the trigger defined by the user. For example, if we have a particular topic on an MQTT message, we can choose to parse the message and insert it into a database, execute a script to process the data and also publish the data to another topic. This rule-based trigger approach offers a diverse data-handling approach with more flexibility in connecting various modules.

In terms of the other features of the cloud platform, the authorization model for Amazon AWS is based on certificates and user roles, which can be leveraged to moderate access to user data. The device-cloud interaction can be handled based on the standard device to cloud interaction model and on distributed data sharing model as well based on 'thing shadows' described in section 3.2. Amazon also offers a diverse set of open source libraries for devices and building applications in various languages and a business intelligence tool called Amazon QuickSight which can be used for visualization of data.

Microsoft Azure IoT: Azure IoT from Microsoft is our second proprietary cloud platform chosen for the use case study. With Azure IoT, the data is handled on the Azure IoT Hub and the communication between the devices and the IoT Hub is handled in various protocols like HTTP, AMQP and MQTT. With Azure IoT we had the opportunity to study the AMQP protocol for communicating between the devices and the cloud platform. Azure IoT also offers a bidirectional interaction model where the data can be sent from devices to the cloud platform while the cloud platform can also send the data to the devices.

Authorization on Azure IoT is handled using Shared Access Security (SAS) tokens assigned to devices and gateways. Azure IoT also offers various other modules for data processing and visualization which can be leveraged by passing the data from the event hub to streams where the data can be temporarily stored. Azure IoT offers a business intelligence tool, Power BI which can be used for visualizing the data stored on the Azure platform.

Phant: Phant in an open-source cloud platform which is offered by SparkFun. The unique feature for this platform is that the platform is both hosted at data.sparkfun.com and can be also hosted on a server by the user. This flexibility is the primary reason for choosing Phant as an open-source platform. Devices can connect to the platform based on the HTTP protocol while the data exchanged is in the form of JSON objects. The data is pushed on to data streams on the platform, which can be uniquely identified by their URIs.

The authorization mechanism of Phant is based on API Keys, which the user or device must pass to the platform to gain access to the data. However, Phant is still in development stages, hence the data pushed on to the platform hosted at data.sparkfun.com is accessible publicly. However, for our use case some of the data we were monitoring was not strictly required to be private, hence Phant was an adequate choice in spite of this shortcoming. We also hosted the platform on our own server, where the data was private and required authorization for access. The Phant platform hosted on the SparkFun server also offers tutorials to publish and visualize the data using Google Charts API.

Parse: Parse is another open-source cloud platform that we have chosen to conduct our study for the use case. The platform offered by Parse was at one time a proprietary cloud platform. However, they decided to terminate their platform hosting services from January 2017 and in turn offer the platform as an open-source version. Hence, the platform is well developed and feature-rich among most other open-source platforms. Parse uses the HTTP protocol for device and cloud communication and the data is exchanged in the form of JSON objects. The authorization of data is handled using access control lists and user groups and API keys as well.

We also chose Parse for our use case to study some other features offered by Parse. Parse offers the option of adding cloud code, which are functions which can be written on the cloud platform and can be executed by invoking them from an application built on the platform or from devices. The cloud code function can serve many purposes by extracting the data stored on the platform and using the data for different applications like visualization, processing and notifications. Parse also offers push notifications to devices with live query, where a user can register for notifications based on an object. When the object is updated or modified, push notifications are sent to the respective user. The visualization of the data for the Parse platform is done using Google Charts API by building a web-service leveraging the Parse platform.

5.2 Use-case scenario

In this section, we present a use case for the performance evaluation of the cloud platforms. For our use case, we monitor a heritage location called 'Circo Massimo' in the heart of Rome. Circo Massimo, once stood as a stage for entertainment with enactment of Roman battles and conquests in ancient times. Beneath the grounds of Circo Massimo lied a mithraeum, an ancient temple erected by the worshippers of Mithras. However, over time, the site of the mithraeum has suffered wear and tear and is currently in ruins, requiring regular maintenance. The site of the mithraeum is monitored and maintained by archaeologists from the university of Trieste. Furthermore, they wanted to improve the monitoring of the site for a deeper understanding of its current condition with the Internet of Things.

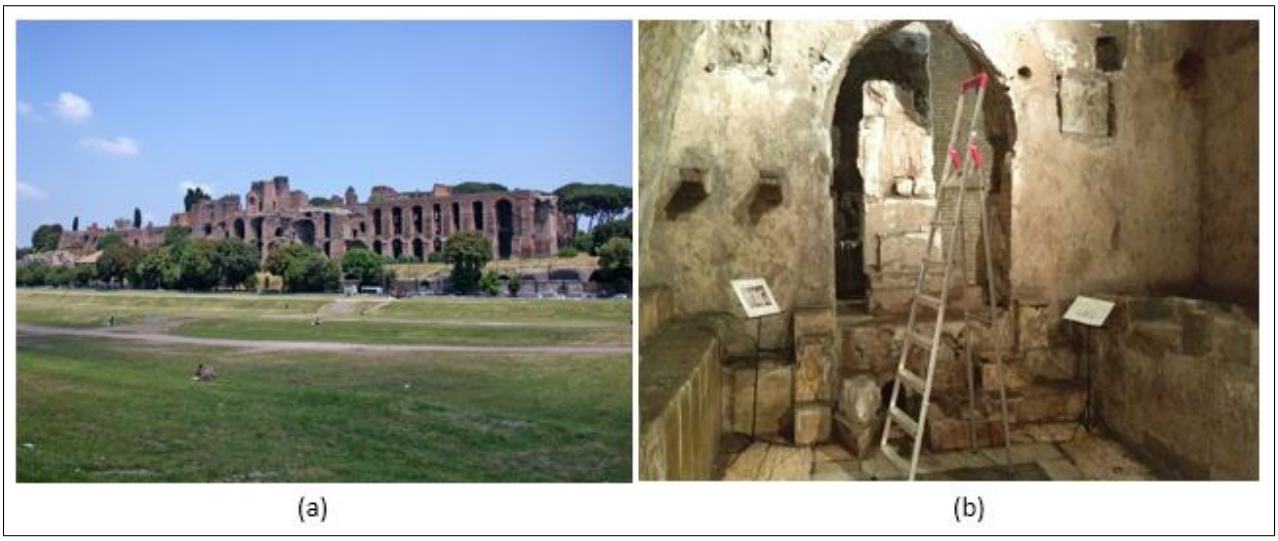

Figure 5.1: The site of Circo Massimo.

Figure 5.1illustrates the site of Circo Massimo in two different views. In Figure 5.1(a), the part of the site for the enactment of wars in the past is depicted, where tourists are allowed to roam freely. On the other hand, Figure 5.1(b), depicts the basement of the site, the mithraeum, which is inaccessible to tourists. This part of the site is currently monitored by the archaeologists. We wanted to monitor the environment of this underground part of the site with a set of sensors. Furthermore, we also wanted to monitor the health of the structure and its susceptibility to traffic moving above the ground.

Requirements for the deployment: The monitoring of the heritage site required us to design a solution which would suit the needs of the archaeologists. The archaeologists had three basic requirements. Firstly, they wanted the data gathered from the sensors to be stored and visualized such that they can see the data individually from each sensor and also in different time windows. For example, they could have chosen to see the data visually from the temperature sensor on a particular node for the past week. Secondly, they required basic processing on the data gathered from some of the sensors. For example, they could have chosen to ask for the maximum and minimum values of humidity in a particular time window. Finally, they also wanted the data to be shared among people who were involved in the project and also make the data available to the public. Keeping these objectives in mind, we had to choose suitable hardware for the edge devices and a cloud platform which could be used to satisfy these requirements.

Challenges involved: Deploying sensors in a constrained environment comes with a set of predictable and a set of unforeseen challenges as well. Some of these challenges can be foreseen while testing the devices in the lab environment while some of them crop up due to the strong influence of the environment on the devices being deployed, triggering bugs and performance degradation. One of the primary challenges that we faced was connectivity to the internet from the basement of the site. Connectivity played a major role in our case since it was pivotal to send and receive data from the cloud platforms. However, the site was underground without any presence of smaller base-stations or access-points in vicinity. The site was only accessible by a metal door, which when closed, severely degraded the connectivity from the inside. Furthermore, the devices once deployed were to be left there without any maintenance. Hence we needed to ensure that the devices function correctly over a long period of time with the exception of unforeseen events. The lifetime of the devices was also a major concern, since visiting the site and replacing the batteries frequently was inconvenient and a costly affair. The sensors attached to the devices consumed different amounts of power. For example, a carbon dioxide sensor consumed more power than a temperature sensor on the device. Thus, we also had to design the configurations of the devices with different sensors to optimize the battery consumption of each device. We were also required to maintain the integrity of the site such that our deployment does not in any way cause harm to the site and its surroundings. For example, even though we wanted to measure the vibrations on the site, drilling into the walls to place the sensor nodes was not a viable option. With these challenges involved, we proceeded with the design and implementation of our solution described in the following sub-sections.

5.2.1 End Devices

The end devices are devices at the edge of the architecture, responsible for collecting data through the sensors the devices are equipped with. The data from these end devices can be sent to a gateway or directly to the cloud platform depending on the architecture chosen for data accumulation. For our deployment we used the Libelium Waspmote as the end device for sensing and collecting data. We used 10 motes marked with an ID in various configurations, equipped with different sensors and a radio module to communicate following the 802.15.4 standard. These motes also sensed data at different intervals to maximize the lifetime of the motes. Primarily, we wanted to monitor the temperature, humidity, ambient light, air quality based on carbon dioxide content and soil temperature values from the site and also monitor the vibrations that

the pillars suffered using the built-in accelerometer. In accordance with these requirements we equipped the sensors with the configurations illustrated in the following table.

Sensor type	Node ID	Rate	Data transmission
Accelerometers - Sample, Light, Humidity, Temperature sensors	1,2	single sample every 10 minutes	raw sample reading
Accelerometers - Burst	3,4	10 secs at 100 Hz every 30 minutes	statisctical measures and FFT values
Carbon-Dioxide sensor	7,8,9,10	single sample every 15 minutes	raw sample reading
Light, Humidity, Temperature, Soil Temperature sensors	5,6	single sample every 10 minutes	raw sample reading

Table 5.1: Configuration of the end devices.

Waspmote nodes with IDs 1 and 2 had identical configurations and were equipped with humidity (808H5V5), temperature (MCP9700A) and Light Dependent Resistor (LDR) sensors. The accelerometer was used to get raw values at an interval of 10 minutes along with values from the other sensors. The light intensity was measured in voltage across the LDR and was then converted into LUX, the unit to measure light intensity. The temperature and humidity sensors produced raw values in degrees centigrade and percentage of relative humidity respectively. For Waspmote nodes with IDs 3 and 4, the nodes were not equipped with any sensor other than the built-in ones. These nodes were used to get the values from the accelerometer in bursts of 10 seconds at 100Hz. This data was furthermore processed to generate Fast Fourier Transform (FFT) of these values along with the mean, median, maxima and minima values over a period of 30 minutes. Due to the intensive processing on these nodes, no other sensors were placed with these nodes. The computation of the FFT was done on the nodes instead of a gateway device, as sending these values to the cloud platform or a gateway device would involve turning on the radio for longer periods which would lead to severe reduction in the lifetime of these nodes. Waspmote nodes with IDs 5 and 6 were equipped with temperature probes (PT1000), humidity, temperature and LDR sensors. The temperature probes were used to monitor the temperature of the crevices in the pillars of the site. The temperature probe also generated raw values in degrees centigrade which were used directly from the sensor. These nodes gathered data every 10 minutes. Waspmote nodes with IDs 7 till 10, were equipped only with the Carbon Dioxide sensor (TGS4161) due to the high power consumption of the sensor. The data gathered from the sensor was in voltages, sensed at the pin to which the sensor was attached. This value was converted to ppm level of CO2. These nodes gathered samples every 15 minutes.

The nodes were put into hibernation mode when they were not sending or gathering in any data to save energy. In the hibernation mode, the node is detached from all parts of the board except the Real Time Clock (RTC), which is powered by the secondary battery. The data accumulated from all the nodes were sent to the gateway device as soon as they were gathered. The network was designed as a one-hop network, such that all the end devices were able to communicate with the gateway directly. The data (with the exception of nodes with IDs 3 and 4) was sent in single XBee packets defined by the Libelium library which had a size of 100 bytes for unencrypted broadcast. The packets were sent to the gateway device over 802.15.4 for

further processing and sending the data to the cloud platform. Nodes with IDs 3 and 4 gathered data from the accelerometers and then sent the FFT values of the accelerometer data in 7 packets for each of the axes. The seventh packet also contained the mean, standard deviation, minima and maxima values for each axis. Figure 5.2 illustrates the positioning of nodes on the floor plan of the basement in Circo Massimo.

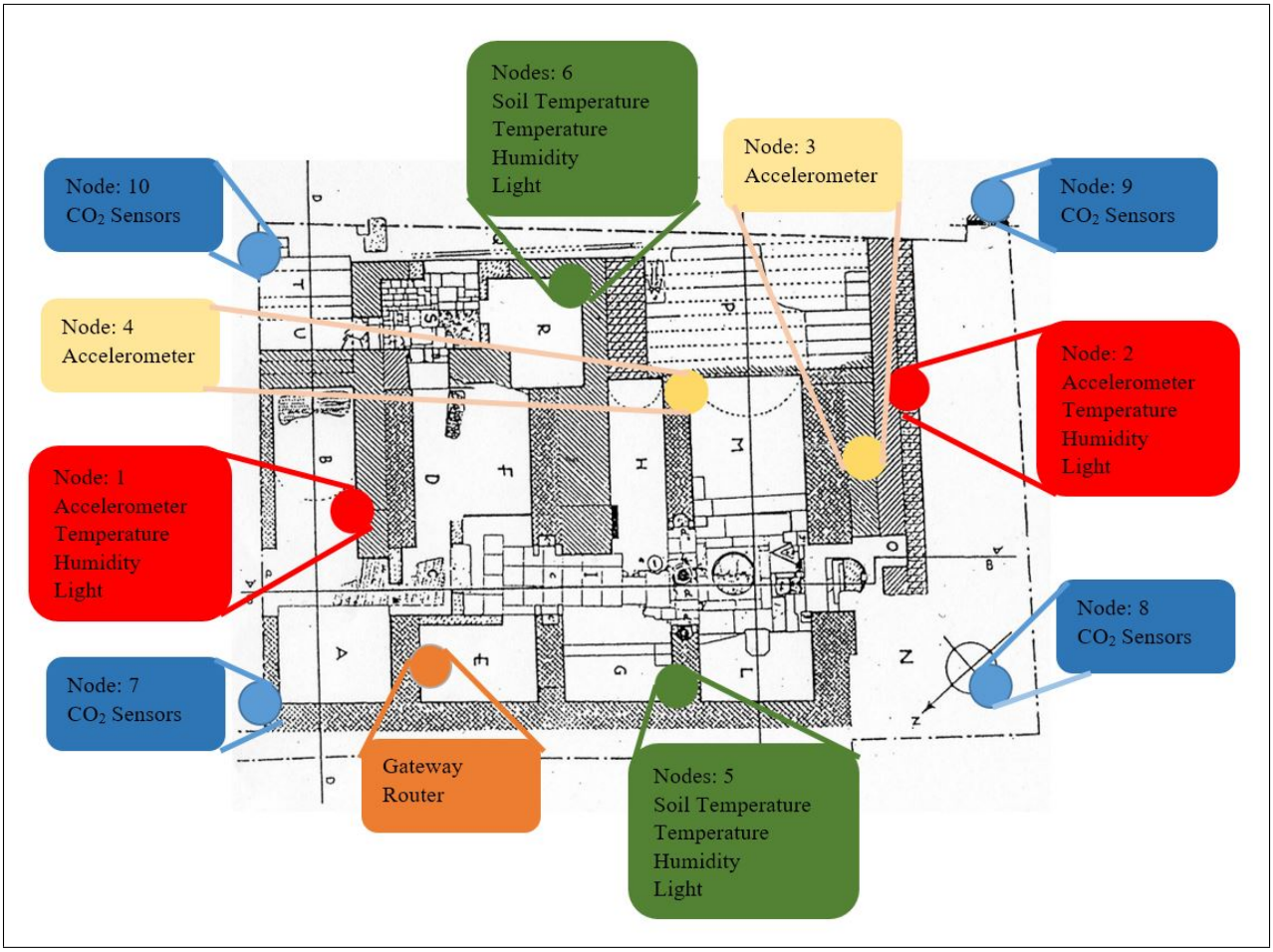

Figure 5.2: Floor plan of Circo Massimo with node positions.

5.2.2 Gateway

In section 2.2 we discussed about the two kinds of architecture, first, involving a gateway used between the end-devices and the cloud platform and the second, sending the data directly from the end devices to the cloud platform bypassing the gateway altogether. In our use case, we decided to design the architecture with a gateway device between the end devices and the cloud platform. The rationale behind this decision were the following.

Firstly, the connectivity inside the basement was very poor and intermittent. The signal was often unavailable or not strong enough to send the data reliably to the cloud platform. Hence we needed to store the data while the connection was down or unreliable. A gateway device served as a buffer to store the data when the connection was unavailable and then send the data when the connection was back up. Secondly,

even with 802.15.4 we got a packet loss rate of about 5-10% for various nodes involved to get the data from the end devices to the gateway. This packet loss could have been mitigated by introducing a multi-hop network where the edge devices form a network to communicate with each other to deliver the data to the gateway. However, we wanted to bypass the complexity of using a multi-hop protocol and hence, adhered to the design of using a single hop to the gateway approach. From Figure 5.2we can observe that the nodes were at different distances from the gateway device where the router for connecting the gateway to the internet was co-located. The maximum packet losses were observed from the nodes furthest away from the gateway namely nodes with IDs 9 and 10. Without the gateway, with an unreliable connection to the internet, the packet drop rate would have been much higher. For these reasons, we used a gateway device in the form of a Raspberry Pi 2 to communicate with the end devices.

The Raspberry Pi was connected to a Libelium Gateway device serially to receive the data from the Waspmote nodes. It was also equipped with a TP-Link Nano-USB adapter to connect it to the internet. Thus the gateway device was capable of communicating both in 802.15.4 serially through the Libelium gateway and connect to the internet over Wi-Fi. The communication and data exchange was handled on the gateway with some scripts scheduled to run at different times.

The gateway was used to run a few scripts to store the data received and then forward the data to the cloud platform of our choice. Some of these scripts were periodically run at a certain interval and some of these scripts were run at a certain time in the day. The scheduling of these scripts was handled by using 'cron', a software utility for scheduling events in Unix-based operating systems. The scheduling was performed by cron by reading a file to which the user inputs the schedule.

We used cron to execute a script every time the Raspberry Pi booted, to initiate other scripts which would continue running throughout the period that the gateway was turned on. It was used to connect the gateway to an overlay private network created using Zero-Tier and initiate another script to receive the data serially from the gateway and write it to a local file. The rationale behind the use of Zero-Tier is explained in section 5.2.3. We also ran another script at startup, to check if the gateway was connected to the cloud platform and accordingly maintain a Boolean variable denoting the status of the connectivity.

We ran a script periodically to read data from the local file generated by reaper.py and parse the data in accordance to the requirement of the platform. For example, we used this script to parse the data for Parse and Amazon AWS to a format which facilitated creation of JSON objects. Furthermore, we used another script to send the parsed data to multiple platforms, to which the gateway was connected.

Data storage: The data received from the Libelium Gateway was in the form of XBee packets which were parsed by the reaper.py script. The payload of each packet was simply copied from the packet and added as a line to a locally stored text file. Thus, each line of the file translated to a set of data generated at a particular time instant, by a single node. The data consisted of the values of the readings from sensors the node was equipped with along with the battery level and ID of the node. Furthermore, for formatting the data in accordance to the platforms to which the data was being sent, we used the parser.py script on this data to generate a file for each platform. In this file, each line corresponded to data from a single sensor, defining the timestamp of data generation, the ID of the node and the sensor value and type. The data

storage formats are shown in listing 5.1 for node 1. The parsed file facilitated the creation of key-value pairs in JSON objects by using separators and also setting topics in message passing protocols based on the kind of sensor the data was generated from.

Listing 5.1: Trace of local file storing raw data.

```
1  //Data written by reaper.py
2  TS:1466573455.0#ID:1#BAT:98#AX:126#AY:79#AZ:71#TE:18.06#HU:76.83#LT:0.00
3
4  //Parsed data from parser.py
5  TS:1466573455.0#ID:1#BT:98
6  TS:1466573455.0#ID:1#AX:126
7  TS:1466573455.0#ID:1#AY:79
8  TS:1466573455.0#ID:1#AZ:71
9  TS:1466573455.0#ID:1#TE:18.06
10 TS:1466573455.0#ID:1#HU:76.83
11 TS:1466573455.0#ID:1#LT:0.00
```

We designed the gateway to be scalable, such that the gateway could send data to multiple platforms at a time and new code for new platforms to send and parse the data could be added without changing the scheduling for Cron. Adding a new platform entails modifying the list of commands to be executed on the scripts parser.sh and send_data.sh and adding the code for the new platform.

5.2.3 Connectivity

Connectivity among the devices, the gateway and the cloud platforms played a pivotal role in our use-case application, due to the unforeseen issues that we encountered while we deployed our network of devices. In this section we address the issues we encountered in device-gateway communication and gateway-cloud communications and how we overcame these issues.

Device-gateway communication: The communication between the devices and the gateway was handled following the 802.15.4 standard. The devices were equipped with external antennae to boost the signal transmission and receipt. When we initially tested the devices in the lab environment, the distance between the nodes were smaller, hence packet drops were not encountered. However, when we actually deployed the network on the site, the placement of the gateway was an important aspect for getting minimal packet losses. The other constraints that we had to consider was the proximity of the gateway to a power source and to the wireless router as well. Packet losses, especially from the nodes computing the FFT values from the accelerometer were detrimental to our application. Since it was a difficult task to extrapolate the values even if a single packet was lost, we had to place higher priority on placement of the gateway with respect to nodes 3 and 4. Keeping these constraints in mind, we placed the gateway near the door as specified in the floor plan in Figure 5.2. We still encountered packet losses from the nodes farthest from the gateway. These nodes were nodes with IDs 9 and 10. Since they were monitoring only air quality, we changed their positions bringing them closer to the gateway to minimize packet loss.

Gateway-cloud communication: The connectivity with the internet was setup using a Netgear AC785-100EUS Mobile Router. The device offered MIMO Dual Band wireless connectivity. Hence we used 2 antennas to boost the signal strength and created a Wi-Fi network in both 2.4 GHz and 5GHz frequencies. While testing with the devices and the gateway in the lab environment we used 4G to create the Wi-Fi network without the use of external antennas and connect the gateway device to the cloud platform on the 2.4 GHz band. However, inside the site where we deployed our network, the best connectivity that we were getting was Edge without the use of external antennas. The packets we needed to send from the Gateway to the cloud platforms were small and sparse in frequency, thus Edge connectivity was sufficient for our application. However, we tried to reach higher reliability with the connection by adding the external antennas, by changing the position of the router inside the site and by switching channels for the Wi-Fi network. We co-located the router with the gateway to use a singular power source and also found the co-location to be optimized in terms of signal strength and reliability. With the 2.4 GHz band, Edge connectivity and a channel selection of 11, we achieved a reliable connection with average latency of up to 100 milliseconds. Occasionally, we had small periods where the latency went up to a few hundreds of milliseconds.

In case of communications with the cloud platform for open source platforms, we hosted the platforms on our server inside the Politecnico network. Thus to access the server from a device not connected to the Politecnico network, we had two primary options to choose from. Firstly, we could have added the Gateway device to a VPN to tunnel the data into the server. Secondly, we could have used Zero-Tier, which is a Software Defined Network (SDN) utility to create private networks where devices can be added. Furthermore, the devices can be accessed as if they were on a local network. We tested both methods for our use-case scenario. For the first case, we used OpenVPN, an open-source software to implement a VPN for creating end-to-end secure channels of communication. For connecting to the Politecnico network, we used the credentials offered by the university on OpenVPN and established communication between the gateway device and the server. For the second option, we installed Zero-Tier on the gateway device. Between these two methods to connect the gateway to the cloud platform, Zero-Tier was easier to set up. Zero-Tier was also more reliable than using OpenVPN, since we encountered frequent disconnections of the gateway from the VPN. Hence we chose Zero-Tier as our preferred method to connect the gateway to our server.

5.3 Experience with cloud platforms

In the previous section, we discussed about the deployment of the network of end-devices along with the gateway and setting them up to send the data to the cloud platforms. With the gateway receiving data from the end devices and equipped to connect to cloud platforms, in this section we discuss about the experience of using each cloud platform selected in section 5.1 with the gateway and how each of these cloud platforms can be leveraged to fulfill the requirements specified by the archaeologists. We study the use of two open source cloud platforms in the form of Parse and Sparkfun and two proprietary platforms in the form of Azure IoT and Amazon AWS. We discuss about the steps involved in fulfillment of the requirements from the archaeologists specified in section 5.2 and describe these steps in detail for each platform. For all platforms, we elucidate the process of setting up connectivity with the server, handling security semantics, data storage and processing on each platform and visualization of data. Furthermore, for the open-source platforms, we discuss about deployment of the platform on our own server.

5.3.1 Deployment of the platform

In this section we discuss about the deployment of the open-source platforms on our own server. The open-source platforms are usually available for cloning as repositories and can be installed on a server of the user's choice. Unlike proprietary platforms, where the platform is already deployed and ready-to-use, open-source platforms have the added complexity of installation and maintenance of the platform on the server. On the other hand, it also brings more control to the user by allowing the user direct access to the data and also the option of modifying parameters on the platform which otherwise would be inaccessible for proprietary platforms. For each platform we discuss about the requirements for deploying the platform and the complexity involved in deployment of the platform on the server.

5.3.1.1 Parse

Parse is an open-source platform available for cloning from a GitHub repository. Till March 2017, their services are also offered on their own deployment of the platform at 'https://parse.com/'. For our use case we have tried to use both the online deployment hosted on the Parse website and a cloned deployment of the platform on our own server. Since the deployment of the platform on https://parse.com/ is getting discontinued, we primarily focus on our own deployment of the platform.

Requirements: The Parse platform is built based on JavaScript and can run on any infrastructure which can run Node.js. The platform is implemented with Express, a web application framework for Node.js. Thus, the basic requirement of running the Parse platform server is only programming framework dependent and thus, can be deployed on different operating systems and architectures, given they are capable of running JavaScript and Express. The Parse platform also requires an underlying database for storage of the data on the platform. The preferred database for Parse is MongoDB, a noSQL database. Thus, one of the complexities involved to deploy the platform, is to install the MongoDB server and configure it to work with the Parse platform.

Implementation: The repository offered by Parse contained a sample implementation of the platform. Thus, the complexity of deploying the platform on the server for our use case, was limited to modification of the parameters defined in this implementation according to the requirements of our use case. The parameters that were defined for the sample implementation included the URI for the database, the application ID, the master key and the server URL.

There were two options for deploying the MongoDB database for the Parse platform. Using the first option, we could have deployed the database locally on the same server as the platform and pointed to the local instance of the base. This option is more suitable when deploying a sample application where the total space available on the server is sufficient. This implementation is also simpler for the user as the URI can be defined based on the localhost of the server. In the second option, we could have deployed the MongoDB database on another server with expandable storage space and passed the address of the MongoDB instance as the URI for the database. In the second option, the database and the platform is decoupled and offers the user more degrees of freedom. For example, a user may choose to have a local implementation of the server while deploying the database on a different cloud storage platform. Since the data generated by our

use case was sparse, we used the first option for our application. The database URI can be also defined as an environment variable or can be hard-coded into the configurations file. While the first option makes it easier to change the database URI, the second option offers a simpler approach for users not accustomed with handling environment variables.

The application ID is used to specify the name of the application being deployed along with the corresponding master key which is used by a device or a user to access the application. For the deployment of the Parse platform on the Parse website, these two values are automatically generated as 40-character long strings when an application is created. However, the local deployment of the platform offers more freedom to the user, where the user can choose a shorter application ID and leave the master key blank as well if the required security is less stringent. For our use case we used 'RomeApp' as our application ID and left the master key blank. The server URL defined for configuration of the platform, defines the URL and the port on which the Parse platform is running. For our use case we used Zero-Tier to share a private overlay network between the gateway and the platform and used a local IP address as the server URL. The private network for the free-tier supports up to 100 devices. Thus, when deploying the platform for a larger set of devices and applications, the deployment would entail use of a public IP address accessible globally or a paid subscription to Zero-Tier.

5.3.1.2 Phant

Phant is an open-source platform available as a GitHub repository. Phant is also hosted by SparkFun Inc. at 'data.sparkfun.com' which can be used as an endpoint to communicate with the Phant platform. Similar to Parse, we tried to study the platform deployed on the SparkFun server along with a local deployment of the platform on our server.

Requirements: The Phant platform is available as an npm package based on Node.js and requires the latest version of Node.js to run. Thus, for basic implementation of the Phant platform, the complexity of cloning a repository from GitHub is bypassed. A user can simply install the latest version of Node.js and install Phant as an npm package. However, for advanced users who wish to modify the code for the platform can clone the repository, configure the platform and then run the code for the platform. Similar to Parse, Phant can be also run on various operating systems and architectures capable of running Node.js.

Implementation: The simplest deployment of the Phant server involves installing the Phant npm package and starting Phant. The platform automatically starts to serve users at the default ports, offering an HTTP server and a telnet connection to the platform. As discussed in section 4.1.6, Phant makes use of HTTP streams called data streams for storing the data. The creation of these data streams are handled using the telnet connection to the Phant platform. Thus, only users who have access to the telnet connection can create the data streams on the platform. Thus, for users who are not working on a Linux/Unix environment, this entails the installation of a Telnet client to configure the data streams for Phant. The creation of each data stream associates it with public and private keys which are used for reading and writing to the data stream. For our use case we created data streams corresponding to each type of sensor deployed in the use case.

Unlike Parse, where the configurations of the server are handled from a single file, here different mechanisms

are used to configure different parts of the server. For example, if a user wants to change the ports for the platform, this can be done by adding a new file and defining the variables separately in them. The data is stored on Phant as a file for each data stream. The schema of these streams are predefined according to Phant documentation. The streams are accessed based on input and output HTTP streams while the consistency of the streams are maintained by a stream manager class. However, this complexity of handling the streams are hidden from the user by providing the Telnet interface to interact with the data streams.

5.3.1.3 Discussion

The implementation of Parse and Phant platforms are quite similar in terms of the framework that they use. Both platforms are built using JavaScript and thus, the installation for each platform is module dependent and are installed using npm. However, in terms of the complexity involved in creating a simple application to store data, Phant is easier to use and quicker to get started. On the other hand, Parse requires installation and configuration of MongoDB separately for the platform. However, this turns out to be an advantage for the user, since the user can not only use the Parse platform to access the data on the platform but also use a MongoDB client to access the data on the platform. On the other hand, since Phant uses their own data structure and storage files to store the data, accessing the modifying the data without using the methods provided by the Phant platform can prove to be a difficult task. Migration of the platform is also simpler for Parse in comparison to Phant. The deployment of the platform can be moved from one platform to another by simply pointing to the same MongoDB database or replicating the MongoDB database. On the other hand, for Phant, the user has to back up the files manually for the data streams. On Parse, new collections for storing data on MongoDB are created when a device makes a request to the server with a new class name. While on Phant, new data streams can be created only through the Telnet interface for the server. Thus, Phant only allows creation of streams from the server itself while Parse also allows client devices to create new classes.

5.3.2 Gateway-cloud interaction

In this subsection, we discuss about setting up the end-to-end connectivity from the gateway device to the cloud platforms. We discuss about the ease-of-use for the libraries provided by the platform, the complexity of handling security semantics and the data schemas used for each platform. For each of the platforms, first we discuss the requirements and challenges to setup the connectivity to the platform using the libraries offered by the platforms. Then, we discuss the security requirements of the use-case and compare them with the security measures required for each cloud platform, followed by the data schema we used for each platform to send data to the cloud platforms.

5.3.2.1 Parse

Connectivity: Parse offers libraries in multiple programming languages to connect devices to the cloud platform. For our use case we studied the EmbeddedC and JavaScript libraries and used them to communicate with the cloud platform. Initially we used the EmbeddedC SDK to communicate with the Parse server. We managed to connect with the platform hosted on the Parse server and send data quite easily by modifying the sample provided on GitHub. However, while we tried to modify the Parse URL and send the data to

the platform deployed on our own server, we could not set up the connectivity to the platform. We raised an issue regarding the same with the Parse community to solve the issue. On the other hand, the platform being written in JavaScript, the JavaScript library is easier to comprehend and use for the platform. The library is also well-documented with the data structures and objects clearly defined. Before working with the Parse platform, I was not familiar with JavaScript. However, with the tutorials provided by the developers of the platform and the documentation of the library, I was able to set up connectivity with both the platform deployed on the Parse server and our own deployment of the server.

For the connectivity to the platform, Parse uses HTTP as the underlying protocol. The data from the devices are sent as JSON objects, to specific classes on the platform. These classes have their own endpoint URIs on the platform. For our use case we used a different class for each of the types of sensors that we had deployed. For example, temperature data from all nodes were gathered and sent to the Temperature class on the Parse platform. We determined the class from the parsed data on the gateway, based on the type of sensor from which the data was received. The data from each line of the parsed file was packed into JSON objects and then sent to the respective class. When a class does not exist on the application deployed on the platform, the class is automatically created. For example, the first data point sent to an endpoint for humidity would create a class humidity and add the data point to the class. Thus, the adding of the classes is handled by the client devices which makes it easier to add new classes to the application by simply posting data to a new class.

Security: The basic requirement for authenticating a device to access an application on Parse is the knowledge of the application ID and the master key for the application. For the platform deployed on the Parse server, these values are generated automatically when the application is created on the Parse platform. Along with these values, the platform also generates other keys for fine fine-tuned access to the applications. For example, instead of sharing the master key, the developer of the application can only share the REST API key which allows a client device to only make REST API requests. However, for the deployment of the Parse platform on our server, only the definition of the Application ID and the master key was sufficient to create an application.

For the use case, we defined only the application ID on the platform for the deployment on our server and left the master key blank. The rationale behind this was, the application ID was not public and hence using the master key in addition to the application ID as a security measure added more complexity to our application. For setting up the device to communicate with the Parse platform, we passed the application ID as a parameter when initializing the endpoint for the platform to authenticate the device. Moreover, we also sent data to an application deployed on the Parse server. For that application, we had to pass both the master key and the application ID to authenticate the device.

Data schema: The data sent to the Parse platform is in the form of JSON objects. However, the schema used for each class is left as a choice to the user. The user can choose the key value pairs to send in the JSON objects according to the requirements of the application. For the use case we sent the timestamp of the data when it was received, the ID of the data and the value of the sensor reading. The JSON object was stored on the platform in classes in accordance to the type of sensor the data was received from. Hence, the type of sensor was not included in the data schema. When the data is stored on the platform, the key value pairs

sent to the platform are stored along with the timestamp at which the object was created on the platform and the timestamp at which the object was last modified.

The underlying database for Parse being a noSQL database, there is no fixed schema for each class. Hence, we could also add additional attributes to each data point on Parse. We used this to mark some of the data points with a Boolean variable to denote if they were sent at a later time than at which the reading was taken from the sensor due to the connection being unavailable.

5.3.2.2 Phant

Connectivity: The connectivity between the devices and the Phant platform is handled over HTTP. For our use case we set up connectivity with the deployed platform on the Sparkfun server and also with the deployed platform on our own server. Phant offers client libraries in a mulitple programming languages to communicate with the Phant platform. The libraries offer simple methods to post and retrieve data from the cloud platform. We studied the Python and Node.js libraries for Phant and sent data to both the deployments of the platform using these libraries. The requirements to communicate with the Phant server is quite simple. The user needs to specify the private key and the public key for the data stream the data is being sent to and specify the data as a key-value pair in a JSON object. The end-point for sending and retrieving the data is independent of the data stream from which the data is being fetched or sent to. Thus given these simple requirements, Phant is a platform where setting up the communication with the platform is quite simple and can be achieved even without the use of libraries. A user, given the knowledge of the private key and public key, can communicate with the platform with ease. The user can simply create a JSON object with the keys and data-points to report the data to the Phant platform endpoint by leveraging a library offering methods to perform curl requests.

Security: The authorization to access a data stream on the Phant platform is handled using two API keys generated automatically when a data stream is created. One of the keys is for reading the data stream, called the public key and the other is for writing to the data stream, called the private key. A device can read the data stream if it passes the public key in the payload of the HTTP PUT request performed to read the resource while it needs to pass the private key in the payload of the HTTP POST request performed to write to a resource.

For the use case, we stored the API keys for each data stream corresponding to a type of sensor in a list and used the required keys to write to the data streams in accordance to the type of sensor values we wanted to store on the platform. We studied the methods to create and delete the data streams from the Phant platform. Creation of data streams on the Phant platform is not controlled by any security mechanism. A data stream creation simply requires a user to be able to establish a telnet connection with the Phant platform. However, deleting a stream requires the user to pass the delete key generated when the data stream is created.

Data schema: The data on the Phant platform is passed as a JSON object to the platform. The number of fields added to the JSON object depends on the fields defined on the data stream schema to which the data is being sent. For example for each sensor type we created a data stream. On each data stream we had 3

fields to store the timestamp, the value of the sensor reading and the node ID. We tried to insert additional fields other than the ones specified on the schema for the data stream, however, the platform only accepts values for fields which are defined on the schema.

5.3.2.3 Amazon AWS IoT

Connectivity: The connectivity with the Amazon AWS IoT platform is handled over multiple protocols namely HTTP, MQTT and HTTP websockets. The core of the AWS IoT platform is based on the MQTT broker. Even with HTTP and websockets the data is sent to specific topics on the MQTT broker using the standard REST API verbs. For our use case we used MQTT as our preferred protocol and Python and C libraries for communicating with the AWS platform. With the MQTT protocol the data is sent in the form of JSON objects to the AWS IoT broker. The broker has different end-points depending on the area the user chooses for using AWS IoT. For example, for our use case we used an endpoint from EU (Ireland). The user has to specify this endpoint for the correct region to interact with the broker. Given the large number of services Amazon has on offer, it may come across as a difficulty for a non-technical person to find the endpoint to the MQTT broker on the AWS IoT console. However, to make things simpler, Amazon AWS offers quick setup guides for Raspberry Pi and Arduino.

We setup connectivity with the MQTT broker using both the Python and C libraries. For our use case we defined a unique MQTT topic for each type of sensor along with the ID of the node. We formatted the topic as '/Sensors/Type_of_Sensor/nodeID'. We packed the data into JSON objects from our parsed file with Python easily, however with C, it required more effort to pack the data into JSON objects. AWS IoT offers an MQTT client on their website to subscribe to topics specified by the user. This client is handy when testing the libraries for sending the data to the AWS IoT broker and offers an easy mechanism for troubleshooting.

Security: The connection of a client device to the AWS IoT platform requires the user to create a thing on the AWS IoT platform. On successful creation, three certificates namely a root CA certificate, a public key and a private key are generated for the device. When the device connects to the AWS IoT platform, it must have these certificates stored in the memory to access the services offered by the platform. This part of the security mechanism consists of authenticating the device to access the AWS IoT platform. For the use case, we registered our gateway device on the platform as a thing and placed the certificates on the device. We used environment variables to store the location of these certificates when connecting to the AWS IoT platform.

The storage and processing of data on the AWS IoT platform is handled by accessing different modules like AWS Lambda for processing data, AWS DynamoDB for storing data and AWS S3 bucket for storing files. Every authenticated device requires authorization for accessing these modules in addition to the AWS IoT module. The authorization to access these modules are handled based on user policies attached to the device. For the use case we used the policy in listing 5.2.

Listing 5.2: User policy for AWS IoT.

```
{
"Version": "2012-10-17",
"Statement": {
"Effect": "Allow",
"Action": [
"dynamodb:PutItem",
"iot:*",
"s3:PutObject",
"lambda:*"
],
"Resource": "*"
}
}
```

For the use case we used an AWS S3 bucket to store backup files from the gateway and hence, we required write access to the S3 module. Similarly we required write access to the DynamoDB database which was also specified in the user policy along with access to AWS Lambda. We created the policy using the Amazon AWS console which allows users to choose the modules for a policy and then attach the policy to a device. The policy can be also created by user by writing the key-value pairs corresponding to the permissions in a JSON object and attaching the policy to the device using the AWS command line interface. However, the second process requires the user to be familiar with a command line interface.

Data schema: The data is sent to the AWS IoT platform in the payload of an MQTT or HTTP request. When the message is received by the AWS IoT broker, the message can be in the format of a string or a JSON object. The schema of the JSON object is left as a choice to the user. For the use case, we sent only the value of the sensor and the timestamp. We did not send the type of sensor or the ID of the node, since the data was published to a topic on the platform which stated the type of the sensor and the ID of the node from which the data was generated from.

5.3.2.4 Microsoft Azure IoT

Connectivity: The connectivity with the Microsoft Azure IoT platform is set up by communicating with the Azure IoT Hub on the platform using protocols like HTTP, MQTT, AMQP and websockets. Communication with the Azure IoT platform, requires a user needs to first create an instance of the Azure IoT Hub and name it uniquely. The creation of the Azure IoT Hub creates a unique end point address for the IoT hub. For the use case, we created an Azure IoT Hub called 'romedeployment' to send the data from the gateway to the Azure IoT Hub. The Azure IoT Hub supports multiple types of endpoints for the supported protocols.

Azure IoT offers libraries in multiple languages for communicating with the IoT Hub. Some of the libraries including Node.js and Python offer a feature which stands out from other platforms. These libraries for Azure IoT supports all of the protocols together to connect to the IoT Hub. In other words, a user can simply

specify the type of protocol while connecting to the IoT Hub and the underlying complexity of setting the endpoint is handled by the library itself. The libraries also offer callbacks for various events like confirmation of a message sent to the IoT Hub or uploading a file to the IoT Hub making troubleshooting easier for the user. For communicating with the IoT hub, we used the Python and Node.js library to connect to the IoT hub leveraging HTTP, MQTT, AMQP. We used the same code for sending the data to the Azure IoT Hub and made changes to the parameter for defining the protocol and dependency for the protocol. We used the callback functions to ensure delivery of the messages to the platform and storage of the data on the platform.

Security: Setting up communication between the Azure IoT Hub and the device requires registration of the device with the Azure IoT Hub. When the Azure IoT Hub is created, a shared access key and a connection string is generated corresponding to the Azure IoT Hub. There are two ways to register a device with the IoT Hub. For Windows users, Azure IoT offers an application called Device Manager with a Graphical User Interface (GUI), to register a new device by passing the shared access key and connection string to the tool. However, for non-Windows users, the only option available is to the use the command line interface to perform a request to the Azure IoT platform and pass the shared access key and connection string as parameters to register a device.

Devices are authenticated on the platform by passing shared access signature (SAS) tokens while performing requests to the Azure IoT Hub to send and fetch data using various protocols. The SAS tokens are generated from the Device Manager for Windows devices and through the command line interface for non-Windows devices. Authorization to offer read or write access to resources on the Azure platform can be granted to devices by defining shared access policies. These policies can be defined from the Azure portal or by performing API requests to the Azure IoT platform. The first option to add the policy on the Azure portal is more user-friendly and is suitable for non-technical people since this method offers a GUI to change the user policies. On the other hand, the use of the REST API allows the user to control the permissions over the internet without the use of a browser. This method is not as well documented as the method for using the Azure portal and requires a thorough understanding of the REST API framework for handling the permissions.

Data schema: Azure IoT Hub supports data in the form of both strings and JSON objects. There is no specific schema defined or any attribute specified which is necessary in the JSON object. When the JSON object is sent, the data is already formatted and allows for easier parsing on the Azure IoT Hub. However, if the data is passed as a string the complexity of parsing it is offloaded to the Azure IoT Hub. For the use case, we sent the timestamp of the data, the type of sensor, the ID of the node and the value of the sensor reading in a JSON object.

5.3.2.5 Discussion

Connectivity: In this discussion, we study the ease-of-use for the platforms to setup connectivity from the gateway with the platform. The documentation primarily focuses on the connectivity to the platform deployed on the SparkFun server and not on the local deployment on the server. For example, the creation of streams using the SparkFun deployment is illustrated clearly while there is no clear documentation on how to use Telnet for the creation and deletion of data streams on the local deployment. For Parse, the

terminology used on the platform for data structures and APIs are discussed in an elaborate document. The APIs for connecting to the platform are well described with sample applications to connect to the platform using multiple operating systems like Linux, Android and IoS. AWS IoT offers quick start guides and samples for devices like Arduino and Raspberry Pi and users can start building applications on top of these samples. The developer guide for the SDKs in different languages are useful and offer a step by step approach to setting up the devices for connecting to the platform. The Azure IoT Hub documentation provides a clear distinction among the use of different protocols to send data to the Azure IoT Hub. The samples provided along with the SDKs facilitate setting up connectivity with the IoT Hub.

If we look at sending the data to the platforms using the HTTP protocol, the overall process of setting up the connection with the Phant platform is intuitive for simple applications and can be achieved without using libraries. For Parse, the sample application provided to connect the device to the platform was well described. Setting up connectivity to the platform, only required changes in configurations, in terms of the database URI, platform endpoint and port. The endpoint for Phant is consistent for messages to all data streams while that of Parse changes based on the class the data is being sent to. However, when handling a large number of classes and data streams on Parse and on Phant respectively, Phant requires the storage of all the private and public keys to communicate with the data streams while on Parse, simply changing the name of the class on the endpoint allows the user to switch between classes. Thus, for larger number of data streams or classes, data handling is easier for Parse. For Amazon AWS, there are no dedicated libraries to communicate with the platform over HTTP, only the structure for the endpoint address is provided for a given topic on the MQTT broker. Setting up communication with the HTTP endpoint is left to the user.

The use of MQTT on AWS IoT is facilitated by the samples provided to connect with the platform. The use of MQTT for AWS IoT is made simple with the online AWS MQTT client provided on the AWS console to check messages received. The parsing of topics for the MQTT messages is intuitive on the AWS IoT rule engine for storage and processing of data on different modules. On the other hand, for Azure IoT, the same library can be used for all protocols, thus switching among protocols simply requires a change in the variable defining the protocol being used. The complexity of using AMQP and MQTT by defining the endpoints on Azure IoT is handled by the library itself.

One of the issues that we encountered while setting up the gateway for Azure IoT and AWS IoT is that these platforms offer a lot of services and modules that contribute to building complex solutions. For example, AWS IoT offers thing shadow and shadow managers to have a distributed data sharing model between the device and the platform. The libraries that they offer, often have all these modules integrated in them such that they can be used from the sample code they provide. However, a user looking to build a simple application where these modules are not required, may feel intimidated by the large number of options at hand. It may puzzle the user as to which parameters to modify and which parameters to leave alone for their application. For example, the Python MQTT library offered by AWS IoT offers classes to use all the features of the platform along with the basic publish-subscribe methods. Hence, for users looking to setup a simple application, we recommend using a third party open-source library which only offers methods to publish and subscribe to topics on the AWS IoT broker [6].

Security: In this section, we discuss the experience of setting up security measures for each platform and compare them with the requirements of the use case. In terms of security, our requirements for the use case were basic. The primary security requirement that we had was that no user was able to gain write access to the data stored on the cloud platform to modify the data gathered from the gateway. Keeping this requirement in mind, we study the security measures of the platforms.

For the Phant platform, the security was handled using API keys per data stream. This entailed storage of the both the public and private keys for each data stream to access each stream. Thus the requirement of preventing other users from modifying the stream was satisfied by simply keeping the private keys secret for the data streams. However, when we create an application using a large number of data streams, the number of keys to be stored also increases, since there is no user level security which can be used to allow access to all data streams together. On the other hand, Parse uses a master key and an application ID to offer access to all classes stored for the application. The security can be further fine-tuned by adding object level permissions using an Access Control List when necessary. However, for applications where stringent security measures are not required Parse stands out as a preferred choice. For the use case, using only the Application ID and keeping the ID secret satisfied the security requirements. All the streams we created on the Sparkfun deployment of the platform were public and accessible for only reading. Thus, for our use case where we wanted to share the data with the public, sharing the web page link to the data on the Sparkfun deployment of Phant could have been an option.

The security measures provided by Amazon are stringent and offer a distinction between authentication of the device and authorization to resources. For simple applications like our use case, implementing all these measures is an overkill and Amazon AWS does not offer a method to bypass some of these security measures. The security measures for Azure IoT are based on getting connection strings from the platform and using them to register the device. The process is simple for Windows users using the Device Manger tool offered by the platform. The device should not necessarily be registered from the device itself and can be registered from other devices as well. Even though, we were using a Linux based gateway device, we used the Device Manager on another device to add the gateway while it was already deployed in the site of Circo Massimo.

Data schema: The data schema for all the platforms are primarily based on JSON objects with the choice of key value pairs in the hands of the user. However, there are subtle differences in the schemas we chose based on the data that was sent along with the data to the platform due to the protocol we used or due to the kind of storage the platform offered. For Parse, the name of the classes was used as a way to recognize the type of sensor the data was generated from. Hence, adding the type of sensor to the JSON object was redundant. Even though a creation timestamp is added to the object when it is saved to the MongoDB database on the platform, we still added the timestamp when the sensor generated the data to the schema for the following reason. Due to unreliable connectivity in the site of deployment, the Pi would often store the data before sending it to the platform when connection was unavailable. Thus, the timestamp at which the data was generated from the sensor often differed from the timestamp at which the data was created on the platform. Similarly for Phant, the platform adds a timestamp of creation to the object when it is stored on the data stream, but we chose to send the timestamp of the sensor reading for the above reason.
The use of strings to send data to the Azure IoT and AWS IoT platforms offers an advantage of bypassing

the complexity of creating a JSON object using low level languages. The complexity of getting the data by parsing the string is bypassed to the platform. However, we used JSON objects instead of strings, to communicate with the platform, since the data was already parsed on the gateway.

5.3.3 Storage

In this section we discuss about the storage of the data sent to the platforms from the gateway. For each platform, we study how the data is stored in the backend of the platform and how the data is accessible to the users after the data is stored on the platform.

5.3.3.1 Parse

The data on the Parse platform is stored in the MongoDB database instance specified on the configuration of the deployed platform. For the server deployment, we used a local instance of a MongoDB database running on the same server. For each type of sensor, we created a class on the platform which translated to a collection on the MongoDB database. A set of documents from the collection for Humidity is shown in figure 5.3 .


```
{
        "TS" : "1478785264",
        "ID" : "5",
        "value" : "94.20",
        "_id" : "bD6nrq1GLC",
        "_created_at" : ISODate("2016-11-10T13:45:14.850Z"),
        "_updated_at" : ISODate("2016-11-10T13:45:14.850Z")
}
{
        "TS" : "1478785253",
        "ID" : "6",
        "value" : "96.28",
        "_id" : "h0TyFp6UcD",
        "_created_at" : ISODate("2016-11-10T13:45:14.865Z"),
        "_updated_at" : ISODate("2016-11-10T13:45:14.865Z")
}
```


Figure 5.3: Humidity data stored on MongoDB instance in Parse.

We accessed the data on the Parse platform using two interfaces. We read the data using the Parse APIs by passing the application ID and accessed the data. We also installed a MongoDB client on the server where the MongoDB database instance was running to gain access directly to the collections stored on the database.

5.3.3.2 Phant

The data sent to the Phant platform for the use case was stored on data streams. Data from each sensor type was stored on a data stream. We created each data stream with a name according to the type of sensor, a description of the type of the sensor, fields named timestamp, ID and reading for storing the values parsed

from the JSON object while we added no tags for each data stream. The data schemas for the streams are stored locally in a file on the server.

5.3.3.3 Amazon AWS IoT

The data on the Amazon AWS IoT platform is received by the MQTT broker with association to the topic to which the data was published. The broker in itself does not have the capacity to store data. When the data is received on the broker, a topic specific rule is triggered to pass the data to some other AWS module which can be then leveraged to store the data. For the use case, we created noSQL based DynamoDB databases on the AWS platform for each type of sensor. Furthermore, we set a rule for inserting the data from the payload of the message to a DynamoDB database based on the topic which was parsed to get the type of the sensor. For example, for humidity data, we set the rule for inserting the data into the DynamoDB database for humidity when the topic was '/Sensor/Humidity/*'.

5.3.3.4 Microsoft Azure IoT

The data published from the devices to the Azure IoT platform is received on the IoT Hub. However, the IoT Hub is not responsible for storage of the data. We needed to use other modules on Azure in addition to the Azure IoT Hub to store the data on Azure. Azure handles passing of data between modules as streams of data and offers a module called Event Hub to convert the data from the IoT Hub into a data stream. Azure also offers a module for storage of data called Azure Blob storage. The IoT Hub exposes an endpoint which is compatible with the Event Hub module. Thus we created an Event Hub to get the data from the IoT Hub and convert it into a stream. The output data stream from the Event Hub was connected to an Azure blob storage instance to store the data.

5.3.3.5 Discussion

The data storage on Parse with MongoDB is accessible through a MongoDB client and the REST API offered by the Parse platform as well. This offers a flexibility for users to choose an access method they are comfortable with. However, this flexibility comes across as both an advantage and a disadvantage. For users familiar with MongoDB, use of the MongoDB client is easier for querying the data and deleting and updating the data. This is due to the fact that the user has direct access to the database and can use standard MongoDB queries on the database. The access through the REST API requires passing the application ID for authentication and then using the API to access the data. A user may be not as familiar with the API for Parse as much as using a MongoDB client, which is more widely used. Thus, the MongoDB client can be used for data validation and can be used to remove data from the platform which is anomalous after they have been passed on to the platform. For example, in our use case deployment, we had a temperature sensor malfunctioning, sending values of over 100 degrees. We deleted these anomalous values by directly accessing the MongoDB database instead of using the API for Parse to access the data. However, the drawback of having the secondary access is that, the MongoDB instance being decoupled from the deployment of the platform, the access to the MongoDB instance using a client should also be secured using certain security measures.

The handling of data on both Azure IoT and AWS IoT is similar in terms of how the data is pushed to an entity which is solely responsible for gathering the data from the devices and not for storage and processing. However, this is owed to the face that both the platform supports multiple endpoints for different protocols whereas the open source platforms only support HTTP. Thus, storing the data received on these brokers is an added complexity for these platforms.

The documentation and tutorials offered by both platforms suffice in storing the data to a desired storage module. The documentation of Azure IoT elucidates the process of storing the data clearly, however, users must be familiar with terms like blob storage, pipelining and streaming in order to have a thorough understanding of the system. The documentation provided for AWS IoT, illustrates the use of the IoT rule engine to set a trigger for inserting the data into a DynamoDB database. However, at the time of writing, the use of a trigger to insert the values from the AWS IoT broker to the DynamoDB database only inserts two attributes from the JSON object. This issue was a drawback of the DynamoDB storage module offered by Amazon AWS. Thus, for our use case we could store only the timestamp and the value of the reading, while the ID of the node had to be dropped when simply using the rule trigger. To store all the key value pairs from the JSON object we needed an added layer of complexity of parsing the data using AWS Lambda and then storing the data on a DynamoDB database. The modules for storage on AWS IoT are illustrated in figure 5.4 .

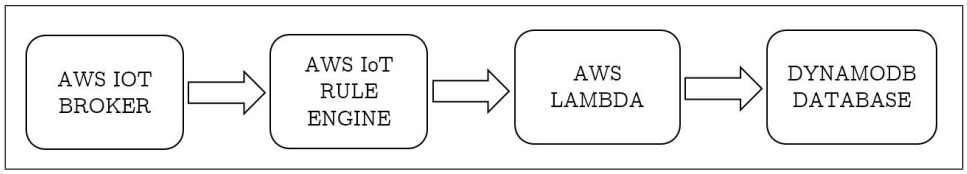

Figure 5.4: Storage of data on Amazon AWS platform.

Similarly, for Azure, the data has to be passed from the broker to a module to stream the data and then store the data on a storage module. For complex applications, where large amounts of data are produced and requires processing on different modules, the streaming of data proves to be an advantage to push the data to multiple modules at a time by connecting the streams. However, for simpler applications like our use case, the task of storing data comes across as quite cumbersome for these platforms.

5.3.4 Processing

In this section we discuss about processing of data for our use case on the platforms. According to our requirements we needed basic processing of data like calculation statistical values namely mean, maximum and minimum values of the data stored for particular types of sensors. Some of the processing was already done on the gateway like calculating the mean, minimum, maximum and standard deviation of the values generated from the accelerometer.

5.3.4.1 Parse

Parse offers libraries to build applications leveraging the Parse REST API for processing data stored on the platform. For the use case we used the JavaScript SDK to build a new application on the platform to process

the data periodically. This new application required the access to the data stored by the other application we used to get the data from the gateway. Thus for fetching the stored data we performed GET requests using the Parse API by passing the Application ID as a parameter for authentication. We performed a query on the data using the Parse API, by passing the name of the class from which the data was to be fetched and the starting and ending timestamp of the data in accordance with the periodicity of the processing, as parameters. After fetching the data stored on the platform, we performed basic statistical analysis on the data and stored the data on a new class. Parse also offers an option to invoke functions stored on the cloud from devices for processing data. For processing the data, the use of cloud code was also an option, to fetch and process the data on the cloud code function. Our requirements were simple and hence, this approach would add to the complexity of the solution. However, for studying the cloud code implementation, we followed a tutorial offered on the platform and used the cloud code to periodically invoke a function to calculate the statistical values. This approach entailed setting up the cloud code module by making changes to the configuration file for the platform deployed on our server and adding the function to calculate the statistical measures from the application to the cloud module.

5.3.4.2 Phant

Phant is still a platform under development and does not offer a framework to build an application on the platform itself. However, the data can be fetched from the platform using the public API keys and can be used to process the data on an application developed on some other platform.

5.3.4.3 Amazon AWS IoT

Data received on the AWS IoT platform can be processed in two ways. The data can be stored on a database first using the method specified in 5.3.3.3 and then processed by fetching the data from the database. The other option can be passing the data to the AWS Lambda module as soon as the data is received and process the data before storing the data to the databases. For the use case, we simply wanted to store the raw data generated from the sensors and then later perform processing on the stored data. Thus, we found the first method more suitable where we stored the data on the platform first and then processed the data by fetching the data from the DynamoDB database. The output of the processed data was stored on another DynamoDB database.

5.3.4.4 Microsoft Azure IoT

Data processing on Azure IoT requires the data to be passed from the Azure IoT Hub to an Event Hub to stream the incoming data. The data is then passed to a Stream Analytics module which takes the stream output from the Event Hub as input and generates a processed data stream as the output. We created a new Stream Analytics module to process the data received on the Event Hub and used it to perform SQL queries on the data received over a time window. Furthermore, we performed operations like calculating the maximum, minimum and average values in the time window using SQL. The output of the Stream Analytics module was stored on another Blob Storage for processed values. The modules for storage and processing are illustrated in figure 5.5 .

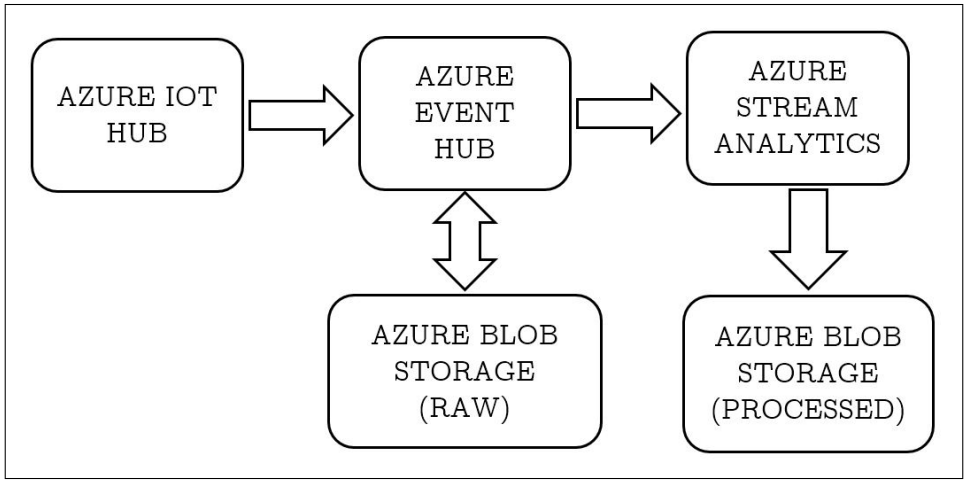

Figure 5.5: Storage and processing on Microsoft Azure IoT.

5.3.4.5 Discussion

The processing requirements for the use case were quite simple, since we wanted to calculate only statistical values from the data. For Parse, the data was retrieved from the application leveraging the REST API. The sample codes presented on the documentation was well explained to perform a query on classes stored on the Parse platform. We used the code and modified it accordingly to fetch the data from our application and process it. On the other hand, given these simple requirements for basic statistical computations in a data set, the steps involved to process the data were complex for Azure IoT and AWS IoT. Data processing on these platforms require the user needs to have an understanding of data pipelining and streaming through multiple modules. For AWS IoT, there is no standard set of functions to use to calculate these statistical values. Amazon offers the AWS Lambda module on which a user can write a script to fetch the data from the DynamoDB database and process it. There is no sample template offered by Amazon to fetch the data from the DynamoDB database through the Lambda module, which makes it difficult for users to familiarize with the use of AWS Lambda to access the DynamoDB database. On Azure IoT, the data is passed through the Event Hub and then processed on a Stream Analytics module. The documentation illustrates steps on interfacing the IoT Hub with the Event Hub and then with the Stream Analytics module. We found tutorials from Microsoft developers in addition to the documentation to be useful to interface these modules with each other. The Stream Analytics module allows the user to parse the data from JSON objects using SQL queries, which is advantageous since SQL is widely used. Furthermore, using the SQL queries, we could also directly process the data to get the statistical values we required. Hence, the data was easier to process on Azure IoT in comparison to AWS IoT for the use case.

5.3.5 Visualization

Visualization of data generated from the sensors deployed was one of our primary requirements. We wanted the data to be visualized on a platform which could be shared with the archaeologists and eventually offer public access. Given these requirements, we study the platforms in the following subsections.

5.3.5.1 Parse

The Parse platform does not offer any module to visualize data stored on the platform. We used Google Charts API, a tool offered by Google to plot charts from data, to build charts for each type of sensor. We created a simple webpage for the user to enter the type of sensor to view the data from, the ID of the node and a time window in which the user wanted to view the data. From the user input, we used the JavaScript SDK to query the platform using the ID of the node, the type of sensor and the timestamps as parameters. We created a data storage object compatible with the Google Charts API to store the data and used the object to generate a chart for the webpage. Figure 5.6 illustrates a sample chart for all nodes showing the respective humidity values plotted from the data stored on the Parse platform.

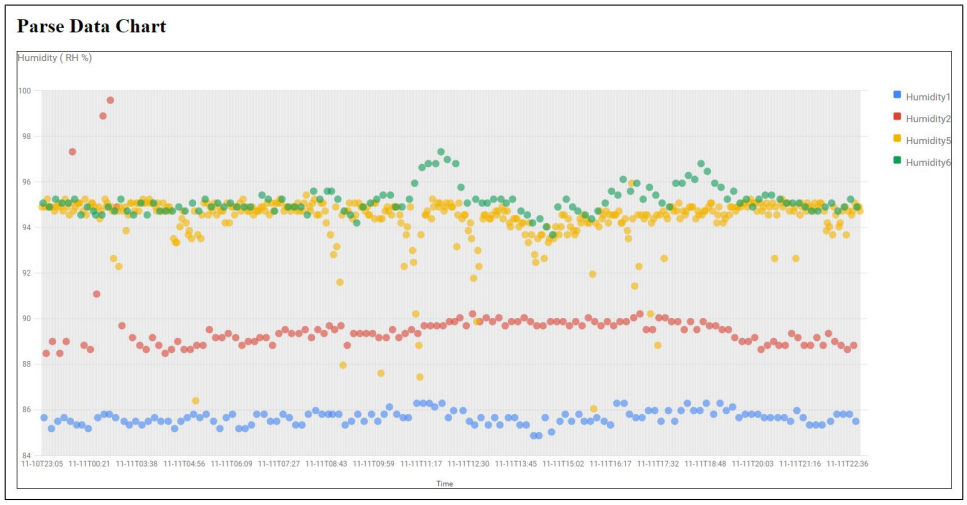

Figure 5.6: Humidity values from the use-case for 2 days.

5.3.5.2 Phant

The Phant platform offers a tutorial to plot live data from the platform by leveraging the Google Charts API. We created another webpage to plot live data from the the data streams using the tutorial offered by Phant. We took the type of sensor and the node of the ID as an input from the user to get the data from the specific data stream. We queried the platform to produce only the first page of the data stored on the corresponding data stream, which translates to the most recent data stored on the data stream. We created a data storage object for the Google Charts API to stored the data obtained from the query. We used the data storage object to plot the chart and display it on the webpage.

5.3.5.3 Amazon AWS IoT

At the time of writing, Amazon AWS offered a business intelligence tool called Amazon QuickSight for visualization of analysis of data. However, the module was offered only to business accounts which was beyond the scope of our project.

5.3.5.4 Microsoft Azure IoT

Azure IoT offers a Business Intelligence tool for data visualization and analysis. However, the services being available only to business accounts, we could not use the tool for the use case. Instead, we used a sample webpage created by a group of developers at Microsoft available as a repository on GitHub. The webpage is compatible with a Stream Analytics endpoint and accumulates the data from the Stream Analytics module to plot it in real time. For the use case, we gathered the data on the Stream Analytics module as mentioned in section 5.3.4.4. We pipelined the output from the Stream Analytics module and connected it to the website to plot data from the sensors on the webpage hosted on the Azure platform.

5.3.5.5 Discussion

We looked for options on many cloud platforms for visualization of data and there are few platforms which provide visualization which offer a free tier. Thus, the combination of an open source platform and the Google Charts API worked well for our use case where we stored the data on the platform and plotted the data on a webpage we developed. We used the Parse platform for creating the time window based webpage and Phant for plotting the data live from the data streams. We could have also used Phant to plot data from a specific time window and Parse to plot real time data. The query models for both platforms are quite similar, based on HTTP requests with parameters specified along with the GET request. On Phant the data is limited by pagination while on Parse the data is limited by the number of data points. The maximum number of points gathered from each query on Parse was 1000 and hence the displayed data from a time window was with less than 1000 data points.

The business intelligence tools from Amazon AWS and Microsoft Azure do not offer free tier services for visualization of data. However, the sample webpage we used for displaying the data from the IoT Hub was setup easily since it was compatible with the Stream Analytics module. Hence we extended the Stream Analytics module used for processing the data as discussed in section 5.3.4.4 to visualize the data on the webpage.

5.3.6 Cost

The pricing of the cloud platform plays a major role when selecting a cloud platform for a use case. The requirements for different applications are different in terms of storage, processing and visualization, which leads to different pricing of the solutions built on the cloud platforms. In this section we will discuss the costs endured for our use case including costs for hosting a platform and costs for using the services offered by the platform.

5.3.6.1 Parse

Parse being an open source platform, the services offered by Parse did not incur any costs for our use case. Thus, the number of messages exchanged with the platform or the amount of data processed on the platform did not contribute to the costs involved. The platform was hosted on a virtual machine deployed on our server along with the MongoDB instance. The platform required storage space on the virtual machine to store the data generated from the devices on the MongoDB database.

5.3.6.2 Phant

Similar to Parse, there were no costs involved for using the services offered by Phant. The platform deployed on the SparkFun website also does not involve any costs at the expense of all data streams being publicly accessible for viewing. For the Phant platform we deployed on our server, we ran the platform on the same virtual machine which ran Parse and used storage space on the server to store the data.

5.3.6.3 Amazon AWS IoT

The pricing model of Amazon AWS IoT is based on incurring costs in accordance with the amount of usage on a per module basis. Each module offers a free tier of usage beyond which costs are accrued for using the services. For the use case, the number of messages the gateway exchanged with the server was below the limit of 5 million messages in the free tier for AWS IoT. The storage of the backup files on the S3 bucket and the use of the rule engine was within the free tier as well. However, the storage of the data on DynamoDB databases was beyond the amount provisioned for the free tier and incurred costs of up to 15€ per month. The processing of data on AWS Lambda was also within the free tier of usage.

5.3.6.4 Microsoft Azure IoT

The costs incurred on Microsoft Azure is dependent on the amount of usage beyond the free tier for various modules. The usage of the Azure IoT Hub to receive data from the gateway was within the free tier for our use case. The blob storage we used for storing the data was also within the free tier of usage. However, the Stream Analytics module usage for processing and parsing data from the IoT Hub incurred costs based on the units of data processed and added up to about 20€ per month.

5.3.6.5 Discussion

The costs incurred for Phant and Parse were only based on the storage space required to deploy the platforms. Hence, this kind of cost is easier to estimate if we can foresee the amount of data generated by the devices per day and plan the costs accordingly. However, the costs on platforms like Azure IoT and AWS IoT are driven based on the services offered, like the units of processing required or the amount the data streaming used for the application. These costs are more difficult to estimate before the deployment of the platform. However, we do have a tradeoff between the costs involved in hosting a platform and storing the data and the costs involved for the usage of services, the proprietary platforms offer, beyond the free tier. Thus, if applications require storage and processing of data on a smaller scale where the usage of services fall in the range of the free tier, the use of proprietary platforms like AWS IoT and Azure IoT are preferable. On the other hand, if the services provided by the platform are required at a higher scale, like performing a large number of queries and streaming of data between modules, the usage of Parse is preferable, where there are no costs involved in using the services of the platform itself.

5.4 Discussion

In this section we discuss about the solutions we built, based on each cloud platform we used and compare them by their suitability to our real-world use case. For the deployment of the open-source platforms, Phant

was easier to deploy since it is available as an npm package for installation. However, when modification of the configuration is necessary, like in our case, Parse was easier to handle for the use-case scenario. In terms of the connectivity to the cloud platforms, the easiest to set-up connectivity was Parse, since it involved the use of only the application ID as a parameter for authentication of the device. We had this advantage on Parse because of the ability to modify the security requirements according to the use-case scenario being considered. The other advantage with Parse was the use of a single endpoint for sending the data to with changes made only to the endpoint URI to modify the class name to which the data was being sent. On the other hand Amazon was much more difficult to set up due to its strignent security measures based on certificates which was an overkill for our use-case scenario. Since we already had our data parsed to create JSON objects, the advantage of being able to send data as plaintext to AWS IoT and Azure IoT was nullified. If we had used the end devices to send directly to the cloud platforms, this would have come across as a handy advantage for our use-case scenario.

In terms of storage of data, Parse was also the easiest platform for achieving storage of data due to the use of MongoDB as an underlying database. Parse was also particularly advantageous in our use-case since the database implementation is decoupled from the platform deployment, it helped us migrate the platform from a local implementation on a virtual machine to that on the server by migrating our MongoDB instance where the data values were stored. However, with Azure IoT, the storage of data required passing the data through multiple modules which was quite cumbersome given our basic storage requirements. The storage on AWS IoT also required the data to be passed over multiple modules, however, the documentation was easier to comprehend and the storage of the data on multiple databases was easier handled thaan the Azure Blob storage. For processing of data, the API to retrieve data from the platform was simplest for Phant, however, it did not offer an application framework to build an application for data processing. Hence, data processing was also most suitable on Parse in comparison to the other platforms given the clear documentation and tutorial for the JavaScript API. Similar to the storage of data, the processing also entailed the use of multiple modules on Azure IoT Hub and Amazon IoT. However, the processing using SQL was simpler and suitable to our use case in case of Azure.

In terms of data visualization, the implementation of the Google Charts API was similar for both Parse and Phant. However, the access of data on Phant is paginated and the requests for getting data on Phant is simpler in comparison to Parse. Hence, Phant was the easier on between Parse and Phant for setting up the visualization. We could not access Amazon AWS Quicksight or Azure Power BI for the data visualization. However, the open-source web service offered for Azure was useful in visualizing the data on the Azure platform. In terms of cost, we used both Parse and Phant without incurring any costs. However, we would rate Parse as a platform more efficient than Phant in terms of cost, since the storage module can be decoupled from the implementation of the platform. This allows the user a degree of freedom to store the data in another location to minimize the costs of data storage. The pricing of Amazon AWS IoT was cheaper for our use-case scenario since the processing we performed was under the free tier limit, while the pricing was higher for processing on Azure IoT Hub. In table 5.2 , we have depicted the platforms for each of the parameters we have defined for the comparison in the order of most suitable to least suitable in terms of the discussion above.

From the table, it is evident that for our use-case Parse stood out as the most suitable platform. This is

Parameter	Platforms			
Deployment	Parse	Phant	-	-
Connectivity	Parse	Phant	Azure IoT	AWS IoT
Storage	Parse	Phant	AWS IoT	Azure IoT
Processing	Parse	Azure IoT	AWS IoT	Phant
Visualization	Phant	Parse	Azure IoT	AWS IoT
Cost	Parse	Phant	AWS IoT	Azure IoT

Table 5.2: Suitability of cloud platforms for use-case scenario.

primarily because of two reasons. Firstly, the requirements of the use-case were simple in terms of storage and processing. Thus, the modules offered by the proprietary platforms were too cumbersome to setup for such a simple requirement. Secondly, the open-source nature of Parse, allowed modification in terms of security, which allowed us to modify the security requirements to our advantage for the use-case scenario. However, we must take into consideration that we considered the deployment of the open-source platforms as intuitive, given we are technically sound to handle the deployment. A non-technical user may prefer to use the proprietary cloud platforms instead of setting up the open source platforms for this use-case scenario. Thus, we can observe from this experience that there are many tradeoffs involved for the choice of cloud platforms and there is not singular choice which can be pointed out as optimal for a given use-case scenario.

Chapter 6

Conclusion and future work

Cloud services for the Internet of Things have grown rapidly with the application to Internet of Things to multiple domains. Among these services, Platform-as-a-Service has gained prominence offering users a platform to gather data from the end devices and gateways to store the data and build applications on top of the platform to process and visualize the data. However, given the large number of platforms available, there is a lack of detailed standard or taxonomy based on which the features of the cloud platforms can be compared. While studies have been conducted to compare features of various cloud platforms, none of them offer a comparison among the platforms based on a real use-case scenario. In this thesis, we have addressed this issue with two distinct contributions, first, a detailed study of the cloud platforms with the definition of a novel taxonomy which is applicable to all cloud platforms and second, a comparison of the cloud platforms based on a real-world use case.

In the first part of the thesis, we conduct a qualitative study of the cloud platforms for the Internet of Things. In chapter 2, we discuss about the challenges involved in storage and processing of the large scale data generated by the Internet of Things and the role of cloud platforms in addressing these issues. We define the entities that comprise the architecture of the Internet of Things in section 2.1, while in section 2.2 we illustrate the types of architecture that are used to build applications using these entities. In section 2.3, we describe the role of protocols used for connecting the things to the cloud platforms. We classify the protocols according to their messaging model and study some of the frequently used protocols in detail. In chapter 3, we define the taxonomy to provide a standard framework for qualitative evaluation of the services offered by the cloud platforms for Internet of Things. We provide a descriptive explanation for each of the terms we have defined and illustrate them with examples. In chapter 4, we present an overview of the cloud platforms available along with their features. Among these platforms, we select a subset of platforms offering Platform-as-a-Service (PaaS) and describe these platforms in detail based on the taxonomy we have defined.

In the second part of the thesis, we select four cloud platforms, two open-source namely Phant and Parse and two proprietary namely Microsoft Azure IoT and Amazon AWS IoT, to study their diverse set of features and compare them based on a use-case scenario. In section 5.1, we assert the rationale behind our selection of the four cloud platforms for comparison. In section 5.2, we describe the use-case scenario involving a real deployment of a sensor network to monitor a heritage site in Rome. We discuss about the requirements

and goals of the deployment as well as the challenges involved in carrying out such a deployment. With the selected platforms, we discuss the implementation of an end-to-end solution for the use case, in accordance to the requirements in section 5.3. We deploy the solution using each of the selected cloud platforms and present them in detail. We compare these platforms based on their documentation, ease-of-use, services offered and the role they played in building the solutions for the use case.

Based on our use-case scenario, we have discussed the suitability of each of the platforms, that we have used to build a solution, depending on how they facilitated the fulfillment of requirements of the use case. In particular, Parse has stood out as the platform, most suitable for our use-case among the 4 platforms we studied. This is owed to the fact that Parse being an open source platform was deployed on our own server. Hence, it allowed us to modify the security requirements according to our use-case to make it simpler according to our requirements. The use of a decoupled instance of a database from the deployed platform also acted as an advantage for Parse. On the other hand, given the simple requirements that we had for the use-case in terms of storage and processing, the proprietary platforms required more effort to perform simple objectives like receiving and storing the data. However, the proprietary platforms are designed for handling larger amounts of data and to perform complex processing as well. Hence, the data set from our use-case scenario was not suitable to study the storage and processing of large scale data on the proprietary platforms.

To extend this work in the future, we would like to study the platforms based on other use cases which would allow us to further study the diversity of platforms. We would like these use use cases to produce larger data sets requiring complex processing to study how the proprietary platforms perform in comparison to the open source ones. Another approach to the use cases would be to study the platforms with diverse non-textual data generated from the end devices and how the platforms facilitate processing of the data, given platforms like IBM IoT, Azure IoT and AWS IoT offer modules for applying machine learning and pattern recognition on the stored data. We would also like to extend this work by studying some more platforms like IBM IoT and Thingspeak in further detail for other use cases. Furthermore, there was a lack of a requirement for actuation on our use case. Thus, a study with an actuation requirement, would also yield an evaluation on the architecture required for handling the actuation and how cloud platforms would facilitate the same.

In conclusion, our work provides a detailed study of the features of 12 cloud platforms based on the taxonomy we have defined and a qualitative evaluation of 4 cloud platforms, based on the requirements of a real world use case. This work lays the foundation for quantitative evaluation of the cloud platforms to study their performance based. This work can be also used by researchers to gain familiarity with the cloud platforms and their features and select cloud platforms based on the taxonomy we have defined.

Bibliography

[1] Amazon cloud platform. https://aws.amazon.com/iot/.

[2] Azure cloud platform. https://azure.microsoft.com/en-us/suites/iot-suite/.

[3] Ibm cloud platform. www.ibm.com/internet-of-things/.

[4] mbed cloud platform. https://www.mbed.com/en/platform/cloud/.

[5] Parse platform. https://parseplatform.github.io/.

[6] Python library for connection to amazon platform. https://github.com/mariocannistra/python-paho-mqtt-for-aws-iot/.

[7] Sense-iot cloud platform. www.sense-iot.com/.

[8] Sicsthsense platform. sense.sics.se/.

[9] Sparkfun phant platform. phant.io/docs/.

[10] Thingplus platform. https://thingplus.net/en/platform-en/.

[11] Thingspeak platform. https://thingspeak.com/.

[12] Ubidots platform. https://ubidots.com/.

[13] Xively platform. https://www.xively.com.

[14] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 2010.

[15] Alessio Botta, Walter De Donato, Valerio Persico, and Antonio Pescapé. On the integration of cloud computing and internet of things. In *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*. IEEE, 2014.

[16] Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. *IEEE Transactions on Industrial Informatics*, 2014.

[17] Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. An iot gateway centric architecture to provide novel m2m services. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*. IEEE, 2014.

[18] Charalampos Doukas and Ilias Maglogiannis. Bringing iot and cloud computing towards pervasive healthcare. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*. IEEE, 2012.

[19] Pankaj Ganguly. Selecting the right iot cloud platform. In *Internet of Things and Applications (IOTA), International Conference on*. IEEE, 2016.

[20] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 2013.

[21] Alex Hernandez. An analysis of middleware platforms for the internet of things. Master's thesis, Politecnico di Milano, 2016.

[22] Oleksiy Mazhelis and Pasi Tyrväinen. A framework for evaluating internet-of-things platforms: Application provider viewpoint. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*. IEEE, 2014.

[23] Julien Mineraud, Oleksiy Mazhelis, Xiang Su, and Sasu Tarkoma. A gap analysis of internet-of-things platforms. *Computer Communications*, 2016.

[24] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: A survey. *IEEE Communications Surveys & Tutorials*, 2014.

[25] Riccardo Petrolo, Valeria Loscrì, and Nathalie Mitton. Towards a smart city based on cloud of things, a survey on the smart city vision and paradigms. *Transactions on Emerging Telecommunications Technologies*, 2015.

[26] Moataz Soliman, Tobi Abiodun, Tarek Hamouda, Jiehan Zhou, and Chung-Horng Lung. Smart home: Integrating internet of things with web services and cloud computing. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*. IEEE, 2013.

[27] Qian Zhu, Ruicong Wang, Qi Chen, Yan Liu, and Weijun Qin. Iot gateway: Bridgingwireless sensor networks into internet of things. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*. IEEE, 2010.

[28] Michele Zorzi, Alexander Gluhak, Sebastian Lange, and Alessandro Bassi. From today's intranet of things to a future internet of things: a wireless-and mobility-related view. *IEEE Wireless Communications*, 2010.