M.Sc. Engg. (CSE) Thesis

Smart Delivery of Push Notification to Secure Multi-User Support for IoT Devices

Submitted by

Md. Shamsul Arifin Mozumder

1015052011

Supervised by

Dr. Muhammad Abdullah Adnan

Submitted to

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Dhaka, Bangladesh

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

July 2021

Candidate's Declaration

I, do, hereby, certify that the work presented in this thesis, titled, "Smart Delivery of Push Notification to Secure Multi-User Support for IoT Devices", is the outcome of the investigation and research carried out by me under the supervision of Dr. Muhammad Abdullah Adnan, Associate Professor, Department of CSE, BUET.

I also declare that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Md. Shamsul Arifin Mozumder

1015052011

The thesis titled "**Smart Delivery of Push Notification to Secure Multi-User Support for IoT Devices**", submitted by Md. Shamsul Arifin Mozumder, Student ID 1015052011, Session September 2020, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfilment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents on July 24, 2021.

Board of Examiners

1.	Adnan Dr. Muhammad Abdullah Adnan Associate Professor Department of CSE, BUET, Dhaka	Chairman (Supervisor)
2.	[illegible] Dr. A. K. M. Ashikur Rahman Professor and Head Department of CSE, BUET, Dhaka	Member (Ex-Officio)
3.	[illegible] Dr. Md. Shohrab Hossain Professor Department of CSE, BUET, Dhaka	Member
4.	[illegible] Dr. A. B. M. Alim Al Islam Professor Department of CSE, BUET, Dhaka	Member
5.	[illegible] Dr. Md. Shamim Reza Professor Department of EEE BUET, Dhaka	Member (External)

Acknowledgement

First of all I would like to thank my supervisor, Dr. Muhammad Abdullah Adnan, for assisting me throughout the thesis. Without his continuous inspiring enthusiasm, encouragement, supervision, guidance and advice it would not have been possible to complete this thesis. I am especially grateful to him for giving me his valuable time whenever I needed, and always providing continuous support, motivation and endless patience towards the completion of the thesis.

I want to thank the other members of my thesis committee: Dr. A. B. M. Alim Al Islam and Dr. Md. Shohrab Hossain for their valuable suggestions. I am also thankful to Samsung R&D Institute Bangladesh Ltd. (SRBD) for supporting this research.

Last but not the least, I am grateful to my guardians, families and friends for their patience, cooperation and inspiration during this period.

Dhaka
July 24, 2021

Md. Shamsul Arifin Mozumder
1015052011

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Internet of things (IoT) with a cloud server has become popular nowadays and it's going to be used in almost every aspect of human life. All devices will be connected to the internet and can communicate with each other where cloud plays an import role in the IoT environment. However, often cloud-enabled IoT environments have potential security risks and do not have multi-user support. In this research, we discuss an IoT push messaging framework named CloudPush framework consisting of a client application, IoT devices, and a cloud system. In this framework, IoT devices can work on an ad hoc network and send event notifications to the client applications through the cloud. We show that CloudPush framework has vulnerabilities while maintaining multiple user accounts between a client application and IoT device in the cloud. The client application can receive unintended and unauthorized notification messages due to the lack of managing multiple accounts properly in the cloud server. To ensure stability in this framework while sending push notifications through the cloud by IoT devices, we discuss potential vulnerabilities and their solutions in this research. We demonstrate that the aggregated throughput of CloudPush framework is 12-15% better than IoTivity framework even though IoTivity does not support multi-user for an IoT resource and a client application. If IoT device's events are sent to multiple client applications i.e. events are distributed among client applications, then the throughput of CloudPush framework increases to 12-25% compared with the IoTivity framework because the CloudPush framework runs optimized searching algorithm in cloud and scales event notifications in both cloud server and cloud push service layer. For a secured multi-user support, notification message data is encrypted that makes the CloudPush system 3-5% slower but still, it performs 9-12% better than the IoTivity framework.

Chapter 1

Introduction

Many IoT platforms are being developed by giant companies like Apple, Intel, Cisco, Amazon, Samsung, etc. One of the IoT platforms is IoTivity framework [1], an open source project hosted by the Linux Foundation [2]. The IoTivity framework creates a standard by which IoT resources and different client devices connect to each other through the internet. IoTivity server updates resource state as an event and IoTivity client works as an observer to get notification of a resource change through IoTivity cloud. However, cloud enabled IoTivity framework has potential security risks and does not have multi-user support. Not only the IoTivity framework but also Google Cloud platform [3], AWS IoT [4], Microsoft Azure IoT [5], Watson IoT platform [6] etc. do not support multi-user feature. An IoTivity client cannot register a resource N times with N number of accounts in IoTivity cloud.

1.1 Significance

In this thesis, we discuss an IoT push messaging framework with multi-user support and measure its vulnerabilities and solutions. We name our system as CloudPush which delivers push notifications to IoT devices smartly and reliably. CloudPush framework consists of a client application, IoT devices and a cloud system. In a multi-user system, a client application can have multiple user accounts and register with an IoT device with those accounts in cloud and vice-versa.

A typical use case in an IoT environment is to notify a human user about an event or a certain condition that occur. The user can receive the notification on an end device that we call client application. It includes a user interface and the user can dismiss or acknowledge the notification. It receives notification messages sent by IoT resources through cloud. Cloud sends push notification using its push messaging service.

We have implemented client application in Android. Client application runs as a service in the background of a device. Firstly, it registers firebase messaging service to receive remote messages sent by IoT devices. Client app service also updates refreshed token by extending FirebaseInstanceIdService class. This token is used by FCM server to identify the client app device for sending push messages. Client app sends account id and password to Amazon cloud to verify given account id. Amazon cloud verifies account through account authentication server. After completing account id verification, Amazon cloud sends success notification to the client application device. Now client app is ready for easy setup with IoT devices. Client app starts easy setup process with an IoT device. After successful wifi-provisioning and cloud-provisioning, IoT device is successfully registered into the cloud and a new row is inserted in the DynamoDB table that contains client device id, account id, authentication code, IoT device id, server-key and token columns. Now the easy setup process is done. When IoT device generates an event according to IoTivity specification, it notifies Amazon cloud that an event has been generated from the IoT device. Then Amazon cloud sends push message to the client application through FCM. If multiple client devices register or perform easy setup with the same IoT device then all registered client devices will get push notification message.

In this research, we discuss some vulnerabilities that arise in the deployment of the CloudPush framework while supporting multi-user feature. Client application receives unintended and unauthorized notification messages through cloud from IoT device where user sensitive data can get exposed. We solve these issues by tracking the current user information in the cloud server and applying encryption in push notification message where the encryption key is generated based on client device id and current user information. In a multi-user system, a notification message triggered by an IoT device can be sent to client application multiple times if the same IoT device registers with the same client account multiple times. We also solve this issue by maintaining unique client application device ID in the cloud. Our solutions will work for all event-triggered multi-user push messaging system like Bluetooth Low Energy (BLE) many to many topology [7, 8], push messaging services, payment solutions, medical services etc.

1.2 Motivation

Usage of IoT devices is increasing day by day. As this system works with IoTivity and security and privacy of accounts, it will resolve a novel security problem of controlling of multiple accounts management for IoT devices using cloud server. Additionally, this system will prevent user sensitive data to get exposed that can be used for hacking. IoT devices are mainly working on ad hoc network. If it is connected to the internet and easily accessible by service makers or other resource clients, more rich scenarios will be possible. Moreover, Constrained Application Protocol (CoAP) is bi-directional RESTful protocol which is used for controlling or being controlled by IoT based devices and easy to convert between CoAP and HTTP [9, 10]. Lightweight CoAP based device can directly connect to public internet using CoAP over TCP protocol. If CoAP over TCP protocol based cloud is available, many Open

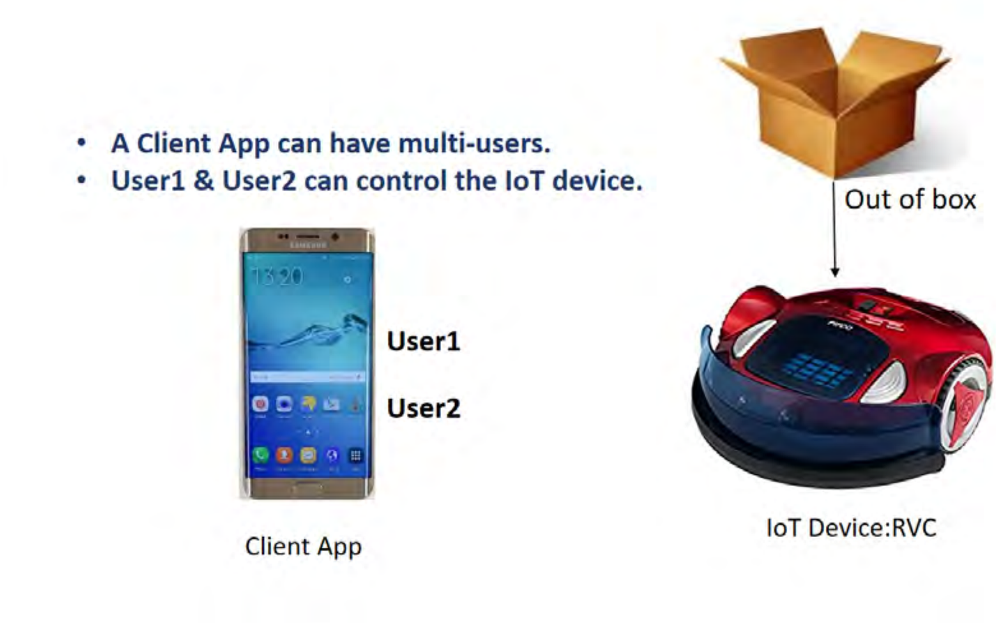

• A Client App can have multi-users.
• User1 & User2 can control the IoT device.
Out of box
User1
User2
Client App
IoT Device:RVC

Figure 1.1: Multi-user Interaction with a Single Smartphone

Connectivity Foundation (OCF) based devices can connect to the internet.

1.2.1 Unavailability of simultaneous multi-user support in existing IoT frameworks

Existing IoTivity, Google Cloud platform, AWS IoT, Microsoft Azure IoT, Watson IoT platform etc. do not support simultaneous multi-user feature. In a multi-user system, an IoT device can be controlled by multiple users.

Multi-user support

- **A single client application:** A client application can support multiple user accounts. Many user accounts can be registered for a single IoT device in the mobile, client application device. In Fig: 1.1, it shows that two user (user1 and user2) with a same smart phone, are controlling an IoT device Robot vacuum cleaner (RVC). In the underdeveloped or developing countries, many families can't afford to buy more than one smart phone. In this scenario, this feature will be really exciting.

- **Multiple client applications:** This system comprises of multiple smart phones (multiple clients) with a single IoT device. In Fig: 1.2, it shows that one smart phone, client app1 maintains two user accounts(user1 and user2) and client app2 has one user account(user3). All the users (user1, user2 and user3) are able to control a single IoT device. In this

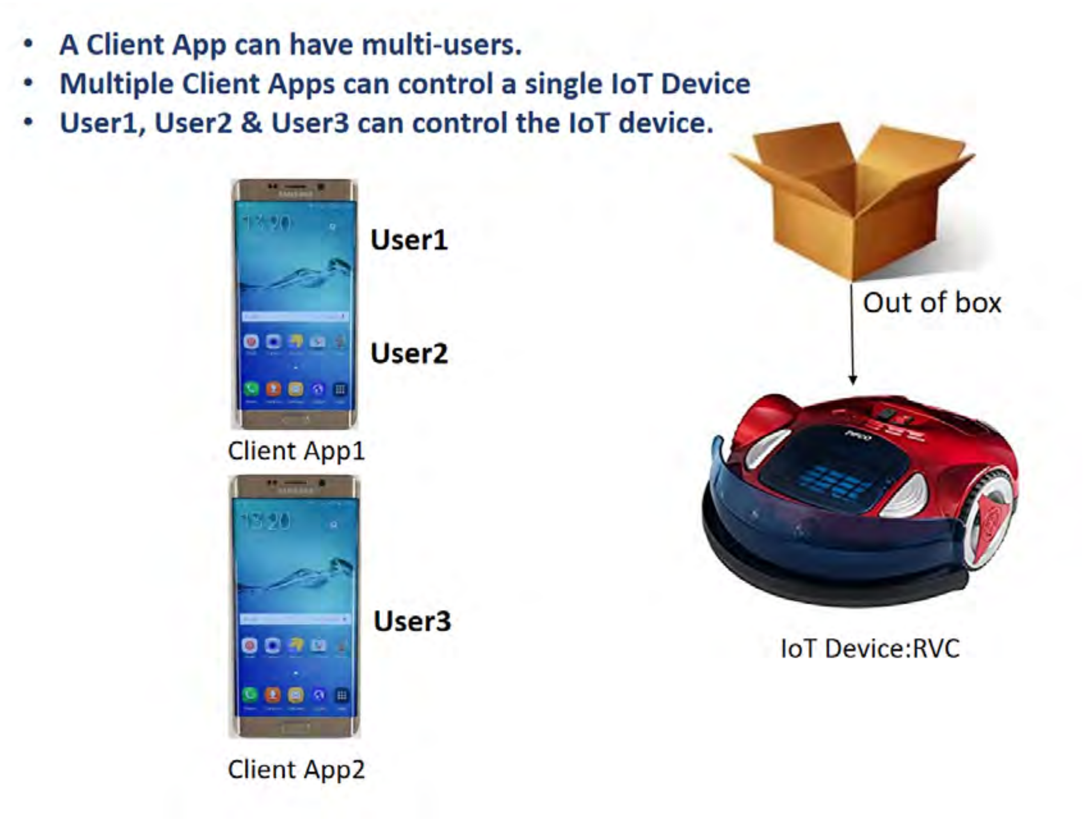

• A Client App can have multi-users.
• Multiple Client Apps can control a single IoT Device
• User1, User2 & User3 can control the IoT device.
User1
User2
Client App1
User3
Client App2
Out of box
IoT Device:RVC

Figure 1.2: Multi-Device Multi-user Interaction with a Single IoT Device

scenario, all registered users including multiple smart phones can control the IoT device (RVC).

Benefits

- A single IoT device can be operated with different combination of settings by multiple user accounts.
- Multi-user support in the IoT platform is cost effective as a single IoT device can be handled with so many options. In the underdeveloped or developing countries, this feature is a perfect fit.
- Productive feature if it is possible to maintain the same or better performance than single user and single IoT device environment.

1.2.2 Slow cloud response time of IoTivity Framework

IoTivity is an open source framework implementing OCF standards for the IoT software developments. The framework operates as middleware across all operating systems and connectivity platforms. It consists of four key components shown in figure: 1.3.

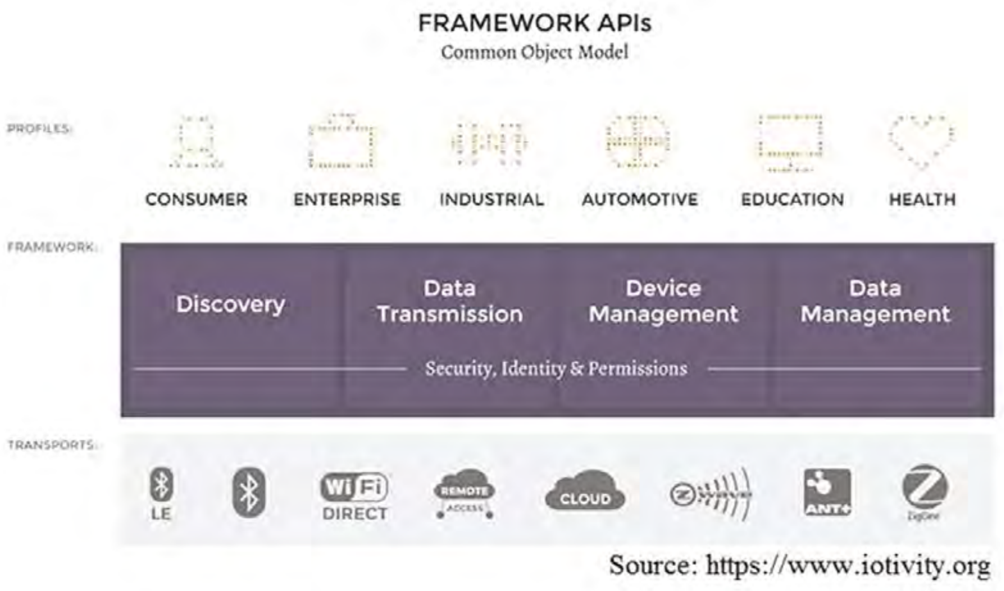


Figure 1.3: IoTivity common object model.

1. Device and resource discovery.
2. Data transmission.
3. Device management.
4. Data management.

The IoTivity discovery component supports multiple discovery mechanisms for devices and resources in proximately and remotely. IoTivity adopts the Constrained Application Protocol (CoAP) defined by the Internet Engineering Task Force (IETF) as a data transmission protocol. As CoAP is a lightweight alternative to Hypertext Transfer Protocol (HTTP), it can work with HTTP by using intermediaries to translate between two protocols. While the data management component supports the collection, storage and analytics of data from various devices, the device management component aims to provide a one-stopshop that supports the configuration, provisioning and diagnostics of devices in an IoT network.

IoTivity Cloud is an open-source platform that aims to extend accessibility of IoTivity devices. The IoTivity Cloud supports techniques such as HTTP to CoAP proxy and OAuth2 over CoAP to enable users to access their devices under their preference accounts over the cloud. Architecture of the IoTivity Cloud is depicted in Fig. 1.4. The platform includes IoT controllers who own IoT devices. To be widely used over the cloud, both controllers and devices must be registered to the cloud first. The devices read sensory data of the physical world, and then send the data to IoTivity cloud servers. Once the data have been published, IoT controllers can access them even they are not co-located.

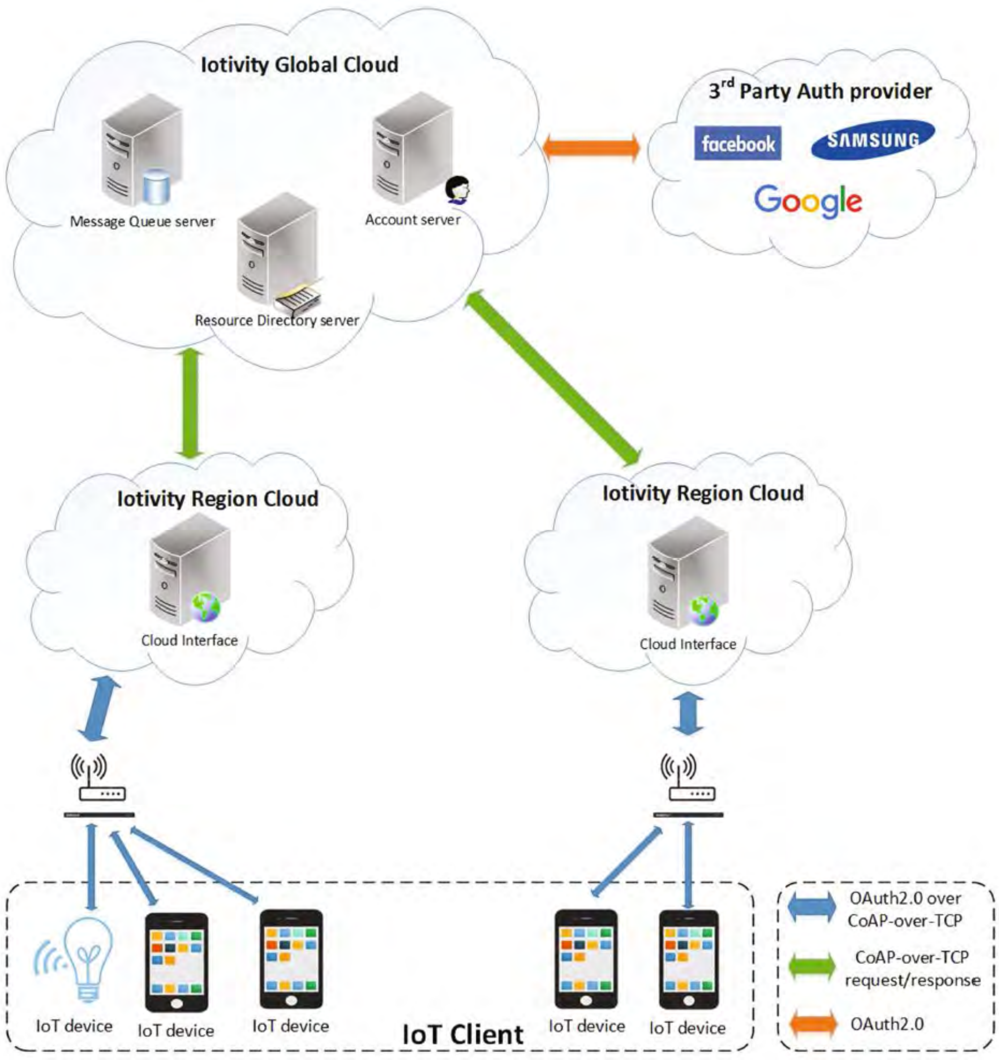


Figure 1.4: IoTivity Cloud architecture.

Most of IoTivity framework core components are developed in C and C++, but the IoTivity Cloud is developed in Java. In the platform, servers are separated into two levels: IoTivity Region Cloud (IRC) and IoTivity Global Cloud (IGC). IRC includes regional Cloud Interface servers that accept connections from both IoT devices and IoT controllers, receive sensory data and sends RESTful messages through connected pipelines to IGC. The IGC is the global cloud that includes Message Queue, Resource Directory, and Account servers. The functionality of each component is explained as follows:

Cloud Interface (CI):

A server acts as an interface of IGC. It is basically a proxy of Message Queue, Resource Directory, and Account servers. Additionally, CI handles the server side OAuth2.0 handshake protocol and the keep-alive messages from IoT devices to ensure the connectivity between the devices and the cloud. Last but not least, CI relays handler such that IoT devices and IoT controllers can communicate when they are connected to different regional clouds.

Resource Directory (RD):

A server supports device registration, discovery, updating, or deleting to IoTivity Cloud. It deploys MongoDB to manage the database of IoT device information.

Account server:

A server supports third party OAuth2.0 enabled authentication providers like Google, Facebook, and Github extending their users identity to IoTivity Cloud. After a user signs up to the Account server, his/her information and a corresponding access token will be stored in a database. Consequently, the user can register his/her devices to IoTivity Cloud for future use.

Message Queue (MQ):

A broker exposes an interface for clients to initiate a publish/subscribe interaction. The server is built on the top of Apache Zookeeper [17] and Apache Kafka [18]. Apache Zookeeper is an open source providing high performance coordination service for distributed applications. It is mainly used to track status of nodes, content topics, and messages, stored in an Apache Kafka cluster. Apache Kafka is a distributed publish/subscribe messaging system. It is written in Scala programming language and designed for processing of real time activity stream data, e.g. logs and metrics collections.

IoTivity client:

IoTivity client consists of IoT devices and IoT controllers. An IoTivity client handles the client side OAuth2.0 handshake protocol and sends keep-alive messages to CI periodically. The client is also able to send resource registration/discovery requests to the cloud.

Performance of IoTivity Framework

The testing results of IoTivity framework show that the CI can handle a load of 1000 IoT devices simultaneously and the maximum throughput reaches 595406.2 (msg/sec). The maximum throughput of the MQ is 4926.8 (msg/sec) which is much smaller than the throughput of the CI. What can be learned from this is the overall throughput of the architecture is mainly depended on the performance of the MQ. In addition, the throughput of the MQ drops significantly when we introduce a large number of IoT controllers into the system.

1.3 Contribution

1.3.1 Introduction of a smart new Cloud Push Notification Framework

We propose a smart new Cloud Push Notification Framework named **CloudPush**. This push messaging framework supports communication among "multiple users (client devices) or multiple accounts of a single user" with an IoT device. It consists of a client application, IoT devices and cloud server. Multiple accounts support in a single client application user device for an IoT device(s) feature is not available in the existing IoT platforms. That means an IoT device can be controlled by creating multiple accounts in a single client device (for example a mobile phone) in the CloudPush framework.

The purpose of CloudPush framework is to maintain a communication between IoT resources and a user (client application device) where IoT resources inform a user if an event occurs at its end through cloud servers. The user can also control an IoT device by sending proper instructions like on, off, move left, move right etc. operations from a client application. IoT device operates on both local networks (wifi, wifi-direct, bluetooth etc.) and remote network (through internet).

1.3.2 Cost effective solutions for controlling IoT devices

Multi-user support in the IoT platform is cost effective as a single IoT device can be handled with so many options. CloudPush framework supports multi-user feature that is a perfect fit in the underdeveloped or developing countries. User do not have to buy multiple smart devices rather one single device is able to control several IoT devices with different attributes. Details of multi-user feature is described in subsection 1.2.1.

1.3.3 An efficient and improved searching algorithm into the cloud database

We apply optimized runtime searching algorithm in cloud that find any component from cloud database in $O(search\_key\_length)$ time complexity. In Cloudpush framework, search items can be:

- Client DeviceID.
- Account ID.
- IoT DeviceID.

These three items are indexed in a search tree. A search tree structure is formed based on these three search items. Leaf of the tree contains a object associated with the cloud table row. When an item is searched from the tree, it can easily retrieve the corresponding row of the cloud table. It saves a significant amount of time while querying and improves throughput of CloudPush framework.

1.3.4 Issues and Security measures of the CloudPush framework

IoT presents a variety of potential security risks that could be exploited to harm consumers by:

- Enabling unauthorized access and misuse of personal information.
- Facilitating attacks on other systems.
- Creating risks to personal safety.

While supporting multi-user feature in CloudPush, we observe some vulnerabilities that can leak information to outsiders. Client application receives unintended and unauthorized notification messages through cloud from IoT device. As a result, user sensitive data can be exposed by others.

Vulnerability-1: Client device receives unauthorized message As CloudPush framework supports multi-user feature, a single client application can use multiple accounts in its device. For a client, only one account id is active at a time. Client application should receive only the notification from IoT devices that are registered with this current account id. Other registered accounts than current should not get notification message triggered by IoT devices. The issue is, client app's current log-in account id is not related to IoT devices but client app can still receive messages from those IoT devices.

Vulnerability-2: Client application receives multiple notification messages instead of a single message When a single client application device with multiple accounts register with a single IoT device then corresponding multiple rows are inserted in cloud table. For multiple accounts (multiple rows in cloud table) with single IoT device, a client application may get multiple messages wrongly.

Vulnerability-3: Same client device ID exists in multiple rows in cloud table If Client app and same IoT device register for multiple times in cloud somehow, then multiple cloud table's rows can be inserted. In this case, if an IoT device produces an error event then the client application will be notified multiples time. It causes performance issue of the CloudPush framework.

Solutions: The following solutions can also be effectively applied for other frameworks like *many-to-many topology, Bluetooth Mesh Network* etc.

- We solve CloudPush framework vulnerabilities by encrypting notification message data and implementing in cloud database layer by checking with currently active client account id.

- To prevent creating similar row multiple times, we have used unique ID instead of random device ID. Every device has a unique id. In case of Android devices, "TelephonyManager" class has an API "getDeviceId()" that provide unique device id.
- To prevent receiving multiple or unauthorized messages in client app for a single event generated by IoT device, we have kept another column into the cloud table that will maintain which one is current account. User can choose the current account in cloud table or last login account will be the current account. Only current account related client device will get message if relevant IoT device generates event.
- Another solution to prevent receiving multiple or unauthorized messages, we have removed all other records related to client app device in cloud table except currently logged in information row. Every time, client device login into cloud, it will create a new row and remove other row with client device id if row with that client device id exists. This approach will provide consistency. But each time sign-in into cloud, a new row will be added and removed old row with client device id. Every time row insertion and deletion is costly and time consuming operation.

1.3.5 Performance of the Cloudpush Framework compared to IoTivity

We validate our solution by setting up experimentation testbed consisting of a client application, Amazon cloud, IoT devices and Firebase Cloud Messaging (FCM) [11], [12] system. We also setup IoTivity framework from open source GitHub codes [13], [14] to produce same scenario like our CloudPush framework source code of CloudPush is available at GitHub: *https://github.com/devSpala/CloudPush* [1].

We compare the performance of CloudPush framework with IoTivity framework [1]. We experiment with a different number of clients and IoT devices in our CloudPush framework and IoTivity framework. We observe in the experiments that aggregated throughput is 12-15% better in CloudPush framework than IoTivity framework for a single client application working as a notification message receiver. The CloudPush framework performs even better 12-25% than IoTivity framework if notification events are distributed among different number of client application devices as we use separate cloud server and cloud push service layer that are scaled up depending on number of events requested by IoT devices and number of client application devices involved with the events during execution time and run optimized searching algorithm in cloud server. Notification message is encrypted for a secured multi-user support but it makes existing CloudPush framework with raw message 3-5% slower but performs 9-12% better than IoTivity framework for a single client. If events are distributed among multiple clients than performance improves about 9-23% than the IoTivity framework.

[1]source code of CloudPush is available at gitHub: `https://github.com/devSpala/CloudPush`

1.4 Organization

The organization of the rest of the thesis is as follows.

Chapter 2: Related Works This chapter includes a brief discussion of the previously proposed push notification framework. Finally we also analyze the existing subset-coding technique and its limitations for real world applications.

Chapter 3: The CloudPush Framework This chapter briefly describes the CloudPush framework and its related terminologies.

Chapter 4: Implementation This chapter includes a brief discussion of the implementation of the proposed model and workflow of the framework.

Chapter 5: Vulnerabilities of the Proposed Framework Vulnerabilities of the Proposed Framework are described in this chapter.

Chapter 6: Solutions Solutions of the vulnerabilities of the Proposed Framework are described in this chapter.

Chapter 7: Experimental Results and Discussion For evaluating the performance of our proposed framework, we have conducted experiment on synthesized data and also made an Android prototype for evaluating its feasibility in real world. This chapter briefly describes the experimental setups and their results.

Chapter 8: Conclusion and Future Works This chapter concludes the thesis and presents some future research direction.

Chapter 2

Related Works

We discuss performance evaluation of CloudPush framework, its vulnerabilities and their solutions in this research. In this chapter, we describe CloudPush framework with existing IoTivity platform, state-of-the-art with other IoT architectures, IoT security issues, Google Cloud Messaging (GCM) [15] security problems etc.

2.1 IoTivity cloud performance

A number of significant technology changes have come together to enable the rise of the Internet of Things. IoT is envisioned to contain billions of devices, including RFID devices, sensors, smartphones, cars and so on, in the near future. To make variety of smart systems such as smart city, smart health care, smart transportation and smart manufacture feasible in the IoT environment, such devices are required to generate a large amount of data and communicate together. There emerges the development of software frameworks to allow the heterogeneous IoT devices to communicate and leverage common software applications.

IoTivity Cloud open-source platform has been developed based on the IoTivity framework. The platform brings an opportunity to collect, analyze, and interpret a huge amount of data available in the IoT environment. The processed data is then required to be available for a provision of smart services to human beings. All these data should be acquired in real-time, stored for a long period and analyzed in a proper way. To this end, they develop a high scalability and low overhead monitoring platform. The monitoring platform will be designed in a way of collecting any type of data into a cloud database by utilizing IoTivity Cloud. This feature opens many opportunities for network operators to customize the monitoring system based on their demands in target deployment area.

Most of IoTivity framework core components are developed in C and C++, but the IoTivity Cloud is developed in Java. In the platform, servers are separated into two levels: IoTivity Region Cloud (IRC) and IoTivity Global Cloud (IGC). IRC includes regional Cloud Interface servers that accept connections from both IoT devices and IoT controllers, receive sensory data and sends RESTful messages through connected pipelines to IGC. The IGC is the global cloud that includes Message Queue, Resource Directory, and Account servers.

In paper [16], evaluates the performance of IoTivity platform. The work in the paper is our inspiration to evaluate the performance of the CloudPush framework and compare output with IoTivity platform. In the experiment, it has used same cloud server for processing and delivering push messages.

The testing results of IoTivity framework show that the CI can handle a load of 1000 IoT devices simultaneously and the maximum throughput reaches 595406.2 (msg/sec). The maximum throughput of the message queue (MQ) is 4926.8 (msg/sec) which is much smaller than the throughput of the cloud interface (CI). What can be learned from this is the overall throughput of the architecture is mainly depended on the performance of the MQ. In addition, the throughput of the MQ drops significantly when it introduced a large number of IoT controllers into the system.

2.2 Security Hazards in Mobile Push-Messaging Services

Push messaging is a type of cloud services that help developers of mobile applications deliver data to their apps running on their customers mobile devices. Such services are offered by almost all major cloud providers. Prominent examples include Google Cloud Messaging (GCM), Apple Push Notification Service and Amazon Device Messaging (ADM). Subscribers of the services are popular apps (Android Device Manager, Facebook, Skype, Netflix, Oracle, ABC news, etc.) on billions of mobile devices. Through those services, the developer can conveniently push notification messages and even commands through apps, avoiding continuous probes initiated from the app side, which are more resource-consuming. With the pervasiveness of the services and the increasingly important information exchanged through them, which includes not only private notifications but also security-critical commands.

In paper [17] found out push messaging security hazards:

- Hackers can steal sensitive messages from a target device by using push-messaging services GCM and Amazon Device Messaging (ADM) and fully take control of the device.
- Cloud in between client device and push-messaging services make the whole system security vulnerable.
- Weak cloud-side access control and extensive use of PendingIntent (Android) cause some applications and system services like PayPal, Facebook, Skype, Google+, Android Device Manager etc. vulnerable and leak out user sensitive data while communicating with those apps.

The problems were found to affect many popular apps, such as Facebook, Google Plus, Skype and PayPal/Chase apps, bringing in serious security threats to billions of Android users. Fundamentally, they come from questionable practices in developing those services, including weak server-side authentication and access control, and the insecure use of the IPC channels and PendingIntent. To mitigate those threats and help app developers protect their communication over those services, they designed and implemented a new technique that establishes an end-to-end secure channel on top of existing push-messaging services.

We also use cloud push service to send push message to client application. They have revealed significant security flaws in push-messaging services that affect billions of mobile users. Hackers can steal sensitive messages from a target device by using push-messaging services GCM and Amazon Device Messaging (ADM) and fully take control of the device. Cloud in between client device and push-messaging services make the whole system security vulnerable. They figure out that weak cloud-side access control and extensive use of PendingIntent (Android) cause some applications and system services like PayPal, Facebook, Skype, Google+, Android Device Manager etc. vulnerable and leak out user sensitive data while communicating with those apps. They solve the problem by restricted use of PendingIntent in Android application and creation of refreshed GCM token in client side. We solve our system's vulnerabilities by encrypting notification message and handling cloud data properly so that other third party's injected malicious data in cloud cannot make our system vulnerable. Checking in database layer makes our IoT system more secure. Security flows of GCM and traces of GCM flow in Android applications help to prevent malicious activities as described in paper [18].

2.3 Existing network security protocols is not enough!

Many IoT projects are being developed nowadays and this technology is very affordable to the general people. Researchers are involved in the activity of IoT development platforms and researching on solving security issues of these platforms. Many domain-specific cloud-based IoT systems relying on heterogeneous environment like hardware, network communication protocols etc. are being proposed to develop here [19–25]. In papers [26, 27], applications of the IoT technologies and cloud computing platforms are investigated and developed. They propose IoT-based cloud manufacturing service system and its architecture. We also propose a cloud-based IoT push notification messaging system in this paper. CloudPush framework is also designed for working in heterogeneous environment with multi-user support.

There are many challenges facing the implementation of IoT. IoT security is not just device security, as all elements need to be considered, including the device, cloud, mobile application, network interfaces, software, use of encryption, use of the authentication and physical security. The scale of IoT application services is large, covers different domains and involves multiple ownership entities. There is a need for a trust framework to enable users of the system to have confidence that the information and services are being exchanged in a secure environment. Moreover, IoT application security and end point security are the biggest concerns. Poorly secured IoT devices and applications make IoT a potential target of cyber attacks. Application developers or manufacturers that create IoT products are not mature from a security standpoint. However, security is a critical dimension of every IoT design. Integrating security in IoT impacts both hardware and software design from the beginning. The technologies to secure devices and connectivity are changing very quickly. It is challenging; security is not just an add-on to existing systems, but an integral part of them. The scope of security should be end-to-end to support the device from the very beginning. Because many IoT devices are small with limited processing, memory, and power capabilities and resources, most current security methods, such as authentication, encryption, access control and auditing, are too complex to run on IoT devices [28].

There are many security solutions available for a hybrid network of the Internet. Although IoT is a hybrid network of the Internet, these security solutions cannot be directly used in the IoT system. Hence new security solutions are described in these papers [29–33]. Deploying in network layer security protocol named Datagram Transport Layer Security (DTLS) protocol is very challenging in an IoT environment [34]. An IoT–Cloud collaboration system is proposed where DTLS handshake delegation needs to apply. We authenticate account id while log-in to the cloud and check unique client device id and unique IoT device id field while inserting or searching into the cloud table and encrypt notification with a key. We solve our security issues from application layer rather than network layer. There is no redundant checking of network layer side in our IoT system. As network layer provides a generic security solution and data communication is checked in every step of a system, so overall process slows down. Comparing to network layer IoT solution, application layer solution in our system is faster.

2.4 Many-to-many topology: Bluetooth low energy mesh networks

The Bluetooth mesh specification was released in July of 2017 with the goal of increasing the range of Bluetooth networks and adding support for more industrial applications that utilize BLE. With Bluetooth mesh [7], a new topology is introduced for BLE. Devices can now operate in a many-to-many topology, or what's called mesh, where devices can set up connections with multiple other devices within the network [35].

Here's an example of a mesh network in a home, fig: 2.1 that's comprised of 6 light switches and 9 light bulbs. The network utilizes the publish-subscribe method to allow nodes to send messages to each other.

Nodes may subscribe to multiple addresses. An example of this is light #3 in the above figure,

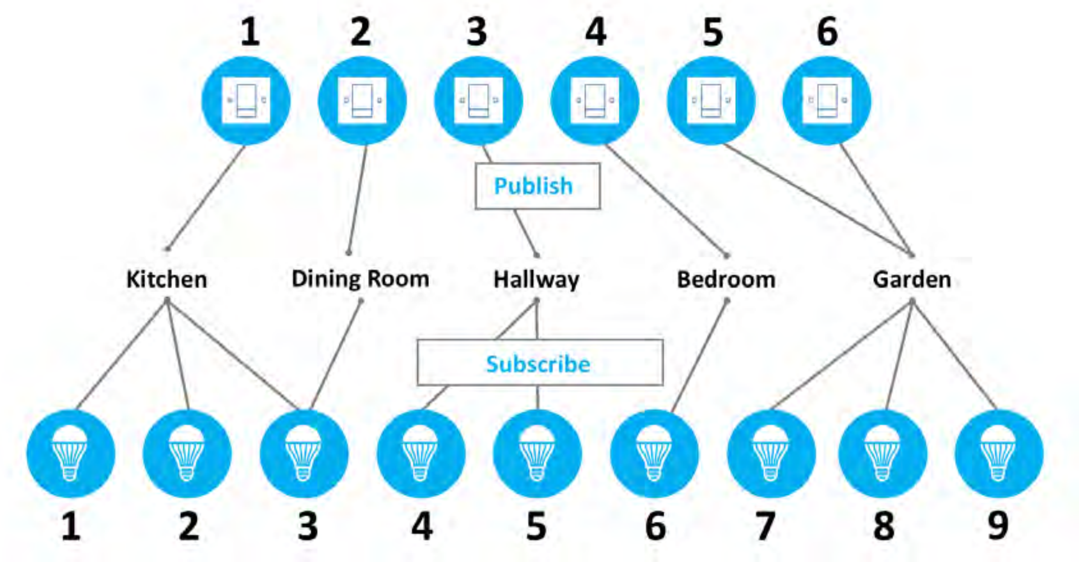


Figure 2.1: Publish-subscribe in Bluetooth mesh lighting control system

which is subscribed to both the kitchen and the dining room group addresses. Also, multiple nodes may publish to the same address, such as switches #5 and #6 in this example. These two switches control the same group of lights, located in the garden.

This is the power of our research where a switch and a light can have multiple attributes. It is more than many-to-many topology. Different users can use the same switch with different variants of the same light using our proposed multi-user supported system. For example, if user A presses the switch #1 then light #1 will give blue color. If other users press the switch #1 then light #1 will give red light. So our proposed system will work in any many-to-many topology regardless of hardware or software layer.

Chapter 3

The CloudPush Framework

The purpose of our IoT push messaging framework is to maintain a communication between IoT resources and a user (client application device) where IoT resources inform a user if an event occurs at its end through cloud servers. The IoT push messaging framework supports communication among “multiple users (client devices) or multiple accounts of a single user” with an IoT device. It consists of a client application, IoT devices and cloud server. Multiple accounts support in a single client application user device for an IoT device(s) feature is not available in the existing IoT platforms. That means an IoT device can be controlled by creating multiple accounts in a single client device (for example a mobile phone) in the CloudPush framework. The overall process of our CloudPush IoT framework is shown in Fig. 3.1.

Firstly client application performs the following tasks:

- Log-in into the cloud with account id and password.
- Registers IoT devices in cloud.
- Receive notification message sent by IoT devices through cloud.

Cloud supports notification messaging to enable a client to be notified of desired states of one or more resources (IoT devices) in an asynchronous manner. Cloud performs the following tasks:

- Completes account verification process and informs client application whether account information is valid or not.
- Maintains cloud table and performs database CRUD (create, read, update, and delete) operations.
- Passes push notification messages to cloud push service.
- Responses accordingly with the request of client applications and IoT devices.

IoT device mainly generates events (notification messages) to inform corresponding client mobile devices through cloud. Notification message parameters are:

- Unique identifier of the IoT device (UUID).
- Unique identifier of the Client.
- Timestamp.
- Key-value pairs of the message's payload.

The user can also control an IoT device by sending proper instructions like on, off, move left, move right etc. operations from a client application. IoT device operates on both local networks (wifi, wifi-direct, bluetooth etc.) and remote network (through internet).
On the other hand in IoTivity platform, cloud servers are separated into two levels:

- IoTivity Region Cloud (IRC).
- IoTivity Global Cloud (IGC).

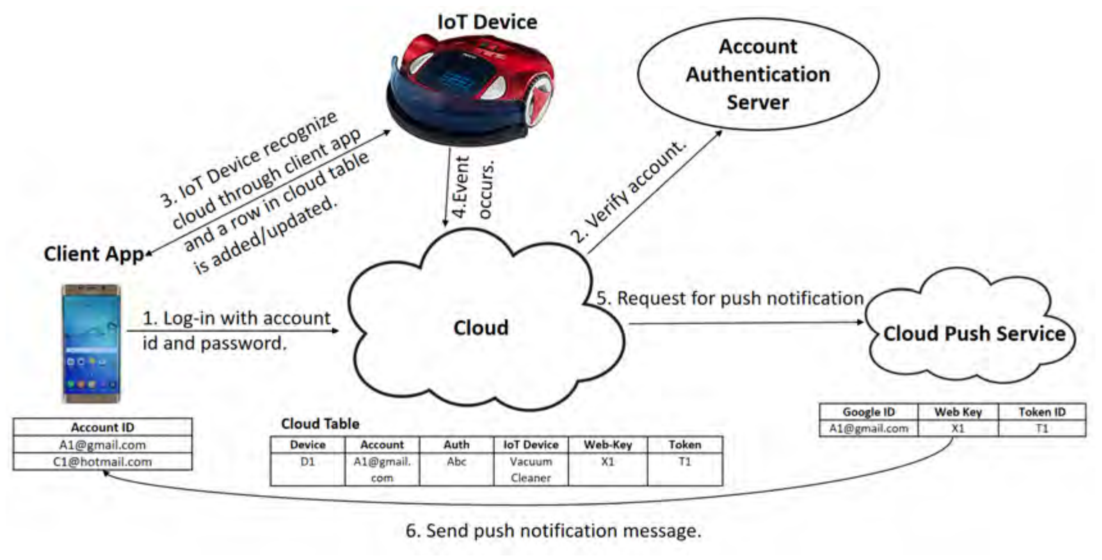
IRC includes regional Cloud Interface servers that accept connections from both IoT devices and IoT controllers, receive sensory data and sends RESTful messages through connected pipelines to IGC. The IGC is the global cloud that includes message queue, resource directory, and account servers. IoTivity client consists of IoT devices and IoT controllers. It handles the client side OAuth2.0 handshake protocol and sends keep-alive messages to cloud interface periodically. The client is also able to send resource registration and observe events sent from the cloud thrown by IoT devices.

3.1 Client application

Client app runs as a service in the background. It performs sign-in to cloud using own credentials like Facebook and Google account, does easy setup with IoT device, inserts or updates rows in Amazon cloud database, generates refreshed token and push notification message receiver and identifies message type defined in IoTvity specification.

3.1.1 Send account information to cloud for authentication

Client app's responsibility is to send account id, password and account type to Amazon cloud. Then cloud verifies account information with the proper authentication server. The client app is replied by cloud whether account login successful or not. After successful login, client app will be able to receive push notification message sent by corresponding IoT devices and perform easy setup with other IoT devices.


IoT Device
Account Authentication Server
3. IoT Device recognize cloud through client app and a row in cloud table is added/updated.
4.Event occurs.
2. Verify account.
Client App
1. Log-in with account id and password.
Cloud
5. Request for push notification
Cloud Push Service
Cloud Table
6. Send push notification message.

Account ID
A1@gmail.com
C1@hotmail.com

Device	Account	Auth	IoT Device	Web-Key	Token
D1	A1@gmail.com	Abc	Vacuum Cleaner	X1	T1

Google ID	Web Key	Token ID
A1@gmail.com	X1	T1

Figure 3.1: CloudPush Framework

3.1.2 Push notification message receiver and refreshed token generation

In client app, we implement an android project to receive a push notification. Client app generates a refresh token that will be used in FCM server part to send the push notification. A service extending FirebaseInstanceIdService is used to access FCM token. For a single client application, unique token is generated so that FCM server can send message directly to the client app. The registration token may change when client app deletes instance ID or client app is restored on a new device or user uninstalls/reinstalls client app or user clears app data in the client device. A client app can receive basic push messages up to 4KB over the Firebase Cloud Messaging APNs interface.

3.2 IoT devices

IoT devices are an outcome of combining the worlds of information technology (IT) and operational technology (OT). Many IoT devices are the result of the convergence of cloud computing, mobile computing, embedded systems, big data, low-price hardware, and other technological advances. IoT devices can provide computing functionality, data storage, and network connectivity for equipment that previously lacked them, enabling new efficiencies and technological capabilities for the equipment, such as remote access for monitoring, configuration, and troubleshooting. IoT can also add the abilities to analyze data about the physical world and use the results to better inform decision making, alter the physical environment, and anticipate future events.

IoT devices are often called “smart” devices because they have sensors and complex data analysis programs (analytics). IoT devices collect data using sensors and offer services to the user based on the analyses of the data and according to user-defined parameters. For example, a smart refrigerator uses sensors (e.g., cameras) to inventory stored items and can alert the user when items run low based on image recognition analyses. Sophisticated IoT devices can "learn" by recognizing patterns in user preferences and historical use data. An IoT device can become "smarter" as its program adjusts to improve its prediction capability so as to enhance user experiences or utility.

IoT devices are connected to the internet: directly; through another IoT device; or both. Network connections are used for sharing information and interacting with users. The IoT creates linkages and connections between physical devices by incorporating software applications. IoT devices can enable users to access information or control devices from anywhere using a variety of internet-connected devices. For example, a smart doorbell and lock may allow a user to see and interact with the person at the door and unlock the door from anywhere using a smartphone.

IoT devices include:

- Automated devices which remotely or automatically adjust lighting or HVAC
- Security systems, such as security alarms or Wi-Fi cameras, including video monitors used in nursery and daycare settings
- Medical devices, such as wireless heart monitors or insulin dispensers
- Thermostats
- Wearables, such as fitness devices
- Lighting modules which activate or deactivate lights
- Smart appliances, such as smart refrigerators and TVs
- Office equipment, such as printers
- Entertainment devices to control music or television from a mobile device
- Fuel monitoring systems.

An IoT device that needs to be configured to join user's device network. It typically has limited CPU, memory, and power resources, so- called "constrained devices". It is often used as sensors/actuators, smart objects, or smart devices. IoT technologies allow everyday objects including small devices in sensor networks to be capable of connecting to the Internet. IoT devices will request cloud to send its event to client app using amazon cloud through fcm. Every IoT device has universal uid that is a unique identity for an IoT device.

In this CloudPush framework, IoT devices will generate events that are defined in IoTivity spec. so that it works in a heterogeneous environment. All IoT device's generated events are recognized in IoT ecosystem. Suppose, an IoT device, Robot Vacuum Cleaner(RVC) is cleaning a room. After cleaning is completed, RVC sends a push notification to client application so that user gets notified. RVC has other event types too like its battery charge level is too low, inside of its desk full of dust etc. So all events of RVC must be recognized by the client application to propagate exact information to the user.

3.2.1 IoT event specifications

A standard notification message format is needed to be defined because there are varieties of IoT devices exist in an IoT ecosystem. As different IoT devices behave differently, Open Connectivity Foundation (OCF) developed standard event specifications [36] and other technology companies will follow these OCF standards. Suppose, Fridge, an IoT device's door is opened and it needs to notify client application (user). In this case, the fridge will need a standard messaging format so that client application understands the message and render properly in the display. On the other hand, IoT device also needs to understand client's requested message to execute its operation. Fridge device has door open or close operation but vacuum cleaner has turn left right operation. So instruction set will vary among devices to devices. We use JSON data format while sending request between a client application and IoT device via the cloud. An example is given below:

```
URI:/notification/message
{
  "rt": "x.notification.message",
  "clientid": "cid",
  "providerid": "uuid",
  "cloudid": "aid",
  "from": "Fridge",
  "to:" "Client",
  "messageData": [
    {
      "messageid": 1234,
      "type": "Home Appliance",
      "Source": "Fridge",
      "title": "Door opened",
      "contenttext": "Front door opened",
      "time": "2017-10-13 11:22",
      "ttl": 30,
      "type": 1 (EVENT)
    }
  ]}
```

3.3 Easy setup with IoT device

When an IoT device is shipped to the end consumers, it's required to configured to the system.A consumer would expect the following logical steps to configure the IoT device:

- Install client application in the mobile device.
- Power on the IoT device.
- Client application recognizes the IoT device and perform an easy setup.

3.3.1 Challenges and Solutions

IoT device can't connect to Wi-Fi network at home

In order to perform easy setup, an IoT device needs to connect to the internet. An IoT device generally gets access to a Wi-Fi network through the client application.The first step in the setup is logically involved providing the home Wi-Fi network and the password. The IoT device is shipped to host an hotspot with a known service set identifier(SSID) name. The client app connects to the IoT device and informs it about the wifi network and cloud information. Once the network settings are available, the IoT device can join the home Wi-Fi network. Then it can communicate with the cloud and complete the easy setup process. It can fall back to be an hot-spot if the credentials are invalid or has other connectivity issues.

IoT device discovery

Normally the client application needs to know the IP address to establish the communication with the IoT device. Conversely, the IoT device can't connect to the client app because it does not know the IP address of it or the phone or client device IP address could have changed. In this case, the solution is the IoT device joins the home Wi-Fi network registered itself to a registry server using a well defined HTTP POST request with the following Parameters:

- MAC address.
- IP address.

The Client can query to get the IP address using HTTP GET request that returns:

- IP address.
- TTL (Time to Live)

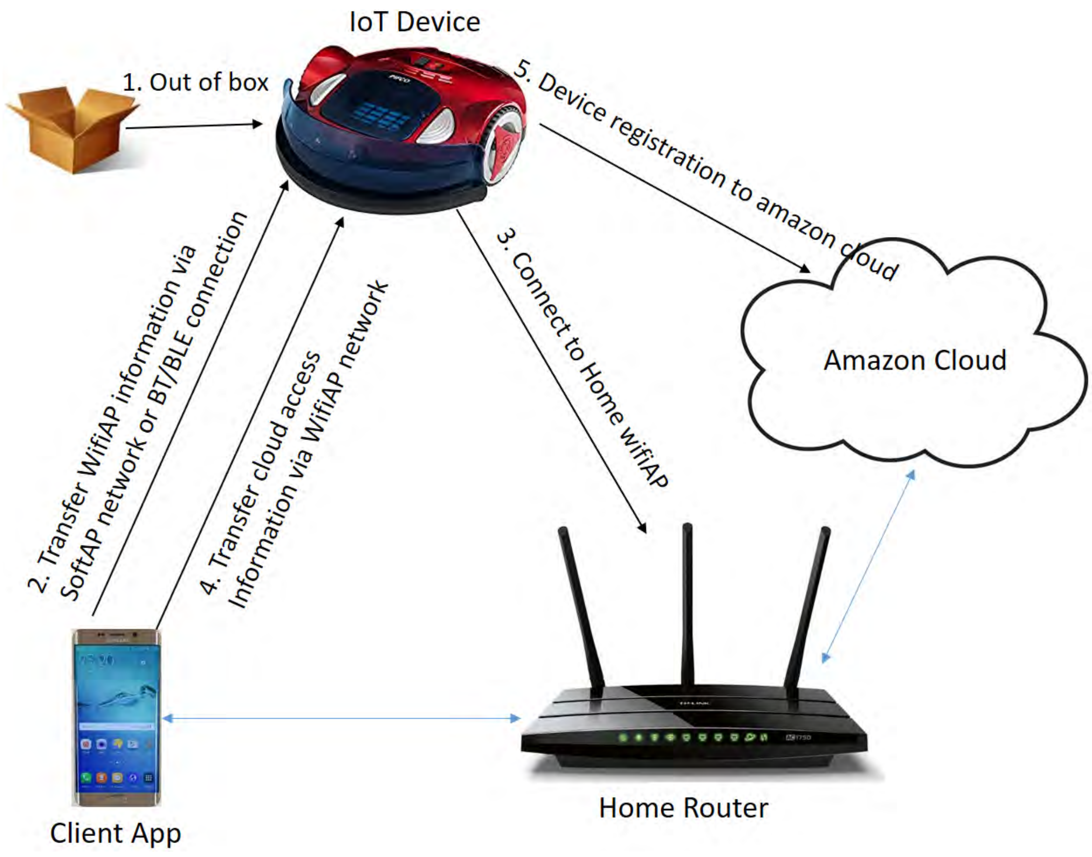

IoT Device
1. Out of box
2. Transfer WifiAP information via SoftAP network or BT/BLE connection
3. Connect to Home wifiAP
4. Transfer cloud access Information via WifiAP network
5. Device registration to amazon cloud
Amazon Cloud
Client App
Home Router

Figure 3.2: Easy setup work flow

The IP address of the IoT device itself is not a publicly visible IP address but private address that is only valid in the local Wi-Fi network. The session mapping of public IP address to the private address is done automatically by the local router using the NAT protocol. The client app manages its own MAC address to IP address table and uses TTL values to intelligently query to get the IP address. Figure 3.2, client app is named as Mediator, IoT device as Enrollee and home AP as Enroller in an IoT system.

After completing successful account authentication, easy setup process starts for the very first time between client application and unboxed IoT device. Client app connects to the unboxed IoT device using a Soft AP network or BT/BLE connection. On the other hand, client app is also connected to the Home router or Home AP that provides public internet. Now client app transfers Home router or Home AP and other information such as Service Set Identifier(SSID), password, security type of Home AP to the unboxed IoT device. Unboxed IoT device connects to Home AP using this information. Client app also transfers cloud access information to the IoT device via Home AP network. Then unboxed IoT device registers to our Amazon cloud and lets client app know that cloud registration is successful. A new entry is added to the cloud table containing IoT device id, client device id, account id, server key, FCM refreshed token etc. after easy setup phase.

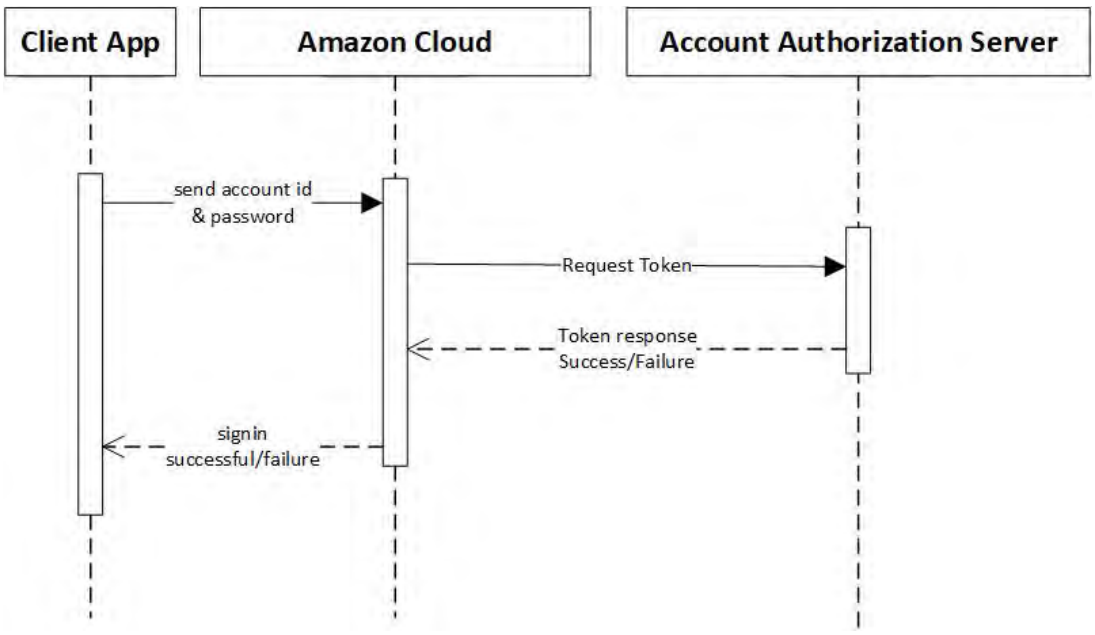

Client App
Amazon Cloud
Account Authorization Server
send account id & password
Request Token
Token response Success/Failure
sign in successful/failure

Figure 3.3: Account authorization

3.4 Private cloud: Amazon Web Services

3.4.1 Account authorization

At first, client app sends account id, password and account type to Amazon cloud. Then Amazon cloud verifies the account information provided by client app with proper account authorization server. Amazon cloud verifies whether provided account information is valid or not. Account authorization server can be Google, Yahoo, Samsung servers etc. Amazon cloud implements only the business layer of account authorization. It returns to client success if provided account is valid or failure otherwise. Account authorization sequence flow is shown in Figure 3.3.

After giving the permission to retrieve an access token on behalf of the end user, Amazon cloud by Authorization server, cloud request for access token. Access token is a long string of characters that serves as a credential used to access protected resources. Account registration is successful if authorization server response with a token. This process runs every time, client app login into amazon cloud. This will ensure cloud security so that no 3rd party app or bot can request to the cloud to hack the system. Only valid app id can request to the cloud.

3.4.2 Push notification message sending logic

We have used Firebase Cloud Messaging (FCM) in our CloudPush framework. FCM is an extremely powerful and simple to use product by Google for sending messages and notifications. It is part of a huge growing ecosystem of integrated tools that is called Firebase. It offers a powerful console, a ready-made integration with a host of other tools such as A/B Testing and

Predictions. It does not offer specific features for the management of users who share the same device and log in and log out of it and for those who at the same time want to receive their own notifications and decide what types of content to receive through them. A typical implementation of Firebase Cloud Messaging system will include an app server that interacts with FCM either using HTTP or XMPP protocol and a client app. Messages and notifications can be sent to the client app using the app server or the Firebase notifications console. It works on the principle of down streaming messages from FCM servers to client's app and upstream messages from client apps to FCM servers. In our framework, IoT device informs cloud that an event has occurred. After that cloud server sends the push message to the corresponding client application devices who register the FCM callback.

3.4.3 Amazon cloud DynamoDB operations

Amazon DynamoDB seamlessly scales a single table over hundreds of servers. It has no limit to the amount of data user can store in its table [37–39]. Our IoT system performs insert operation after easy setup phase and scans cloud table when data communication between client applications and IoT devices are needed via the cloud.

We have used Amazon cloud as private cloud. Amazon Web Service(AWS) is a cloud computing platform that enables user to access on demand computing services like database storage, virtual cloud server, etc. And AWS Lambda lets user run code without provisioning or managing servers. There are many services that AWS provides. We mainly used two services of AWS to implement our cloud framework. We upload some logic(code) into cloud and AWS Lambda takes care of everything required to run and scale the code with high availability. Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets user offload the administrative burdens of operating and scaling a distributed database, so that user doesn't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. We have performed amazon dynamoDB for create, update, read, write, scan/query etc. into the table.

After finishing easy setup procedure, client app will insert a row in amazon cloud table. There is no limit to the amount of data user can store in an Amazon DynamoDB table, and the service automatically allocates more storage, as user store more data using the Amazon DynamoDB write APIs. Amazon DynamoDB scales horizontally and seamlessly scales a single table over hundreds of servers. Scaling of dynamodb depends on two dimensions, throughput and size of the data. Any number of items can be added to a table where maximum item size is 400KB. So scaling is achieved through partitioning. Throughput is provisioned at the table level. Write capacity units (WCUs) are measured in 1 KB per second and read capacity units (RCUs) are measured in 4 KB per second. RCUs measure strictly consistent reads and eventually consistent reads cost 1/2 of consistent reads. Read and write throughput limits are independent [38].

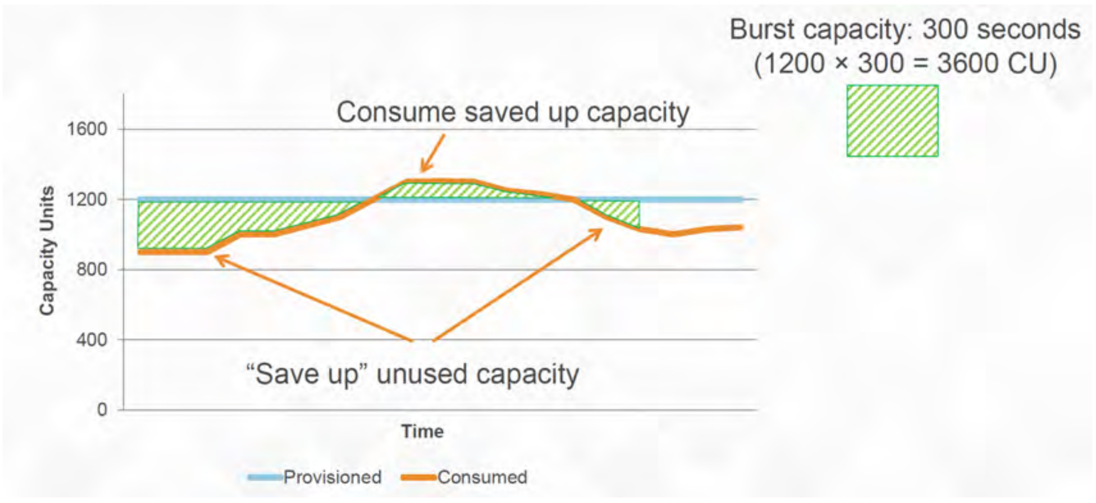


Figure 3.4: Built-in burst capacity

As IoT devices are increasing rapidly, we have used amazon dynamoDB that will support unlimited number of rows. For determine performance issue properly, we perform partitioning math below.

Table 3.1: Partitioning math

Number of partitions	
By Capacity	(Total RCU/3000)+(Total WCU/1000)
By Size	Total Size/10 GB
Total Partitions	CEILING(MAX(Capacity, Size))

Table 3.2: Partitioning Example:
Total size = 8GB, RCUs=5000, WCUs=500

Number of partitions	
By Capacity	(5000/3000)+(500/1000)=2.17
By Size	8/10 = 0.8
Total Partitions	CEILING(MAX(2.17, 0.8))=3

RCUs and WCUs are uniformly spread across partitions.

RCUs per partition = 5000/3 = 1666.67

WCUs per partition = 500/3 = 166.67

Data/partition = 10/3 = 3.33 GB

According to DynamoDB developer guide, "To get the most out of DynamoDB throughput, create tables where the hash key element has a large number of distinct values, and values are requested fairly uniformly, as randomly as possible." DynamoDB saves 300 seconds of unused capacity per partition to handle bursts. This is used when a partition runs out of provisioned throughput due to bursts and provided excess capacity that is available at the node.

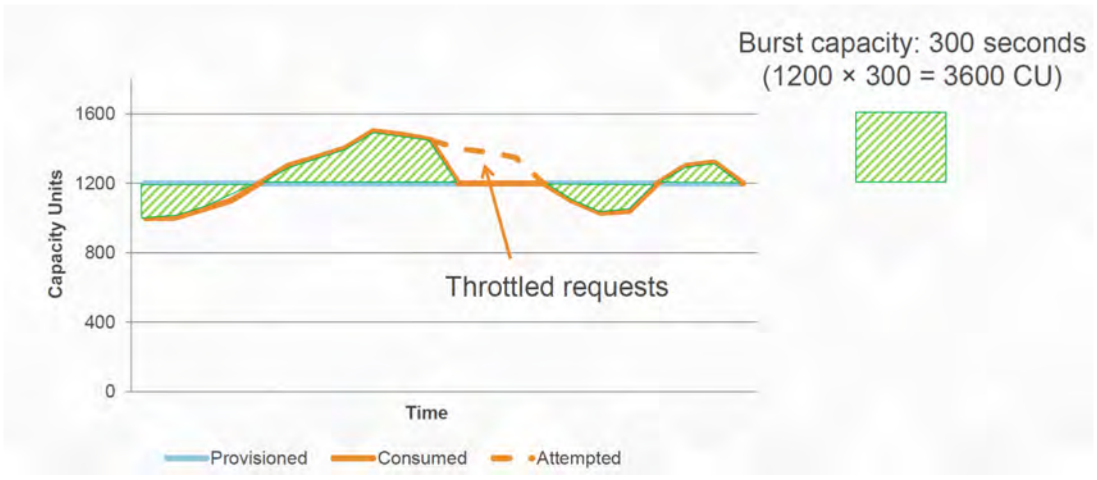


Figure 3.5: Insufficient burst capacity

If sustained throughput goes beyond provisioned throughput per partition caused throttling, Fig: [3.4 , 3.5].

Suppose a table is created with 5000 RCUs, 500 WCUs

RCUs per partition = 1666.67

WCUs per partition = 166.67

If sustained throughput is greater than (1666 RCUs or 166 WCUs) per key or partition, DynamoDB may throttle requests. Solution of this problem is to increase provisioned throughput.

3.5 Algorithms

CloudPush framework initialization algorithm is described in 1. Initially client application logs-in to the cloud with a token given by account authentication server. It helps IoT device by providing information about cloud and a new row is inserted in the cloud table for the first time when client application, IoT device and cloud are connected with each other. After that client registers cloud push service to receive any events or state change from the IoT device.

In algorithm 2, when an IoT device state is changed or event is triggered, cloud gets notified and scans cloud table. It retrieves a list of events associated with the IoT device id. Then cloud push service scans for corresponding client list to send notifications to clients. After that clients receive and process the push notification messages.

Algorithm 1 CloudPush Framework Initialization

```
if $token = null$ then
    $token \leftarrow Cloud.GetToken(AccountID, Password)$
end if
Client.LoginCloud($token$)
if IoT Device is registered in cloud then
    Connect($IoTDevice$, $Cloud$)
else
    Client.Send($IoTDevice$, $CloudInformation$)
    Connect($IoTDevice$, $Cloud$)
    Cloud.AddToCloudTable($IoTDevice$, $ClientInformation$)
    Cloud.RegisterPushService($CloudPushService$)
end if
```

Algorithm 2 Event Triggered By IoT Device

```
Cloud:
while $eventTriggered$ in Cloud do
    $eventList \leftarrow Cloud.search(CloudTable, IoTDeviceID)$
    Cloud.Send($CloudPushService$, $eventList$)
    $ClientList \leftarrow CloudPushService.Search(eventList)$
    while $ClientList$ is not empty do
        CloudPushService.Send($Client$, $clientEventList$)
    end while
end while

Client Application:
while $waitingForPushMessage$ do
    $message \leftarrow CloudPushService.getPushMessage()$
    Client.Process($message$)
end while
```

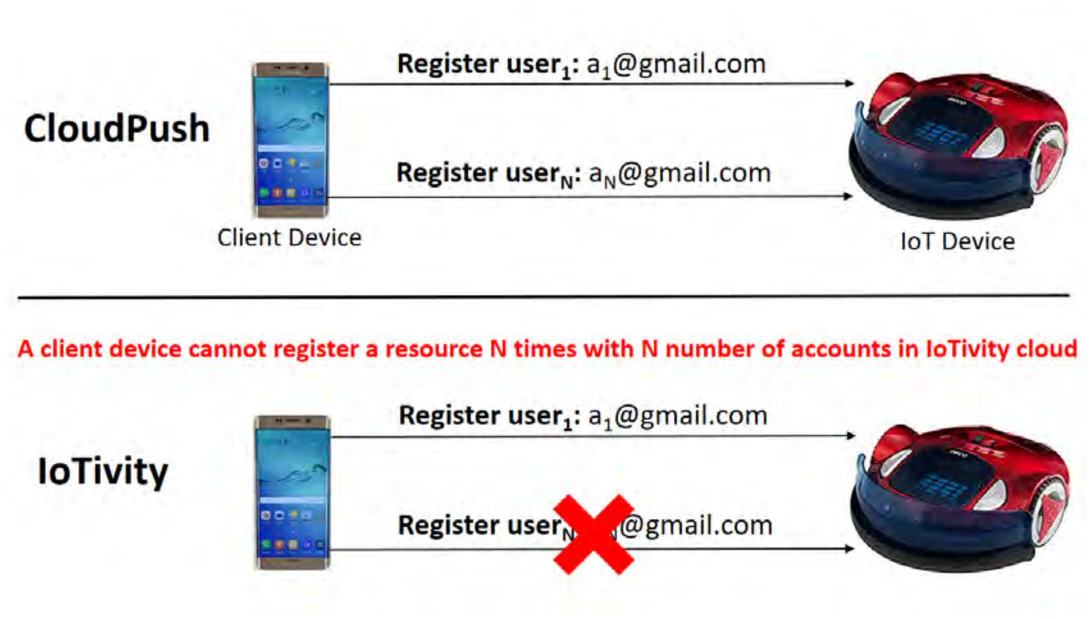

CloudPush
Register user$_1$: a$_1$@gmail.com
Register user$_N$: a$_N$@gmail.com
Client Device
IoT Device
A client device cannot register a resource N times with N number of accounts in IoTivity cloud
IoTivity
Register user$_1$: a$_1$@gmail.com
Register user$_N$: a$_N$@gmail.com

Figure 3.6: Diff CloudPush Iotivity

3.6 Difference between CloudPush And IoTivity notification framework

Difference between the CloudPush And IoTivity notification framework is shown in Fig: 3.6. It clearly depicts that the CloudPush framework supports communication among ”multiple users (client devices) or multiple accounts of a single user” with an IoT device where IoTivity framework doesn’t.

Chapter 4

Implementation

In our CloudPush framwork, client application runs as a service. It registers firebase messaging service to receive remote messages sent by IoT devices. Client app service also updates refreshed token by extending FirebaseInstanceIdService class. This token is used by FCM server to identify the client app device for sending push messages. Client app now sends account id and password to amazon cloud to verify account id. Amazon cloud verifies account through account authentication server. After completed verification account, amazon cloud sends success notification to client app. Now client app is ready for easy setup. Client app starts easy setup process with IoT device. After successful wifi-provisioning and cloud-provisioning, IoT device is successfully registered into the cloud and a new row is inserted in the dynamodb table that contains client device id, account id, authentication code, IoT device id, web-key and token columns. Now the setup is done. If IoT device generates an event according to iotivity specification, it notifies amazon cloud that an event is generated from the IoT device. Inside dynamodb table, specific IoT device related entries are searched. As FCM server logic is implemented inside amazon cloud, then it requests for push notification to FCM server. FCM server then sends the push notification to client app. If N number of rows of the IoT devices are found, then N times FCM notifications are sent to client app through firebase cloud messaging service.

4.1 Client application

Client app runs as a service in the background. It performs sign-in to cloud using own credentials like Facebook and Google account, does easy setup with IoT device, inserts or updates rows in Amazon cloud database, generates refreshed token and push notification message receiver and identifies message type defined in IoTvity specification.

It registers Firebase Cloud Messaging (FCM) client part to receive push notification from IoT devices. Then client app sign-up to amazon cloud using their own credential like Facebook or Google Account. Amazon cloud will verify through proper authentication servers. For example, client app sign-up google account using OAuth 2.0 where amazon cloud will verify token to make sure valid sign-in. After successful cloud sign up process, client app will start easy setup process with corresponding IoT device. When easy setup process is completed then client app will add a row inside amazon cloud table with deviceID, Account, Authentication code, IoT device ID, FCM Web key and FCM token. After IoT devices are connected to cloud through internet and ready for receiving command. Even if user is not at home, he can control things using 3G network. If things working as Hub, it also delegates local device to cloud.

4.1.1 Send account information to cloud for authentication

Client app's responsibility is to send account id, password and account type to Amazon cloud. Then cloud verifies account information with the proper authentication server. The client app is replied by cloud whether account login successful or not. After successful login, client app will be able to receive push notification message sent by corresponding IoT devices and perform easy setup with other IoT devices.

4.1.2 Push notification message receiver and refreshed token generation

Logic for refreshed token and push message receiving APIs is shown in algorithm 3

Algorithm 3 Push notification handling and refresh token access

```
Push notification receive:
Receive $RemoteMessage$ from the push server
if $RemoteMessagesize > 0$ then
    Process push notification message by Client device
end if
Refresh token:
if $Refreshtokenisexpired$ then
    Request $MyFirebaseInstanceIDService$ instance to get the refresh token from push
    server.
    $refreshedToken \leftarrow FirebaseInstanceId.getToken()$
end if
```

In client app, we implement an android project to receive a push notification. Push notification message attributes are converted into JSON string. Then the JSON string is passed and processed in the communication channel is shown in algorithm 4.

Client app generates a refresh token that will be used in FCM server part to send the push notification. A service extending FirebaseInstanceIdService is used to access FCM token. For a single client application, unique token is generated so that FCM server can send message directly to the client app. The registration token may change when client app deletes instance ID or client app is restored on a new device or user uninstalls/reinstalls client app or user clears app data in

Algorithm 4 JSON conversion from push message

```
Instantiate parent JSON object
Create and add every notification attribute into its own json objects
Add notification object to parent
Add request attributes to parent
if $Multicast$ Collection is not empty then
    Create and add all targets to the JSON array
    Add targets to parent.
end if
```

the client device. A client app can receive basic push messages up to 4KB over the Firebase Cloud Messaging APNs interface.

4.2 IoT devices

IoT devices will generate events that are defined in IoTivity spec. so that it works in a heterogeneous environment. All IoT device's generated events are recognized in IoT ecosystem. Suppose, an IoT device, Robot Vacuum Cleaner(RVC) is cleaning a room. After cleaning is completed, RVC sends a push notification to client application so that user gets notified. RVC has other event types too like its battery charge level is too low, inside of its desk full of dust etc. So all events of RVC must be recognized by the client application to propagate exact information to the user.

4.2.1 IoT event specifications

A standard notification message format is needed to be defined because there are varieties of IoT devices exist in an IoT ecosystem. As different IoT devices behave differently, Open Connectivity Foundation (OCF) developed standard event specifications [36] and other technology companies will follow these OCF standards. Suppose, Fridge, an IoT device's door is opened and it needs to notify client application (user). In this case, the fridge will need a standard messaging format so that client application understands the message and render properly in the display. On the other hand, IoT device also needs to understand client's requested message to execute its operation. Fridge device has door open or close operation but vacuum cleaner has turn left right operation. So instruction set will vary among devices to devices. We use JSON data format while sending request between a client application and IoT device via the cloud. An example is given below:

```
URI:/notification/message
{
  "rt": "x.notification.message",
  "clientid": "cid",
  "providerid": "uuid",
  "cloudid": "aid",
  "from": "Fridge",
  "to:" "Client",
  "messageData": [
    {
      "messageid": 1234,
      "type": "Home␣Appliance",
      "Source": "Fridge",
      "title": "Door␣opened",
      "contenttext": "Front␣door␣opened",
      "time": "2017-10-13␣11:22",
      "ttl": 30,
      "type": 1 (EVENT)
    }
  ]
}
```

4.2.2 Virtual IoT Device

We implement virtual IoT device to generate an event that is sent to cloud and the cloud requests the message to the corresponding client application device through FCM server. Virtual IoT device is nothing but an Android application that trigger the lambda function of the Amazon cloud if an event is occurred. Then Amazon cloud has the communication channel with FCM. When Amazon cloud communicates with corresponding client application device id with FCM then it sends the push message to the client device. Virtual IoT device event sending logic is shown in algorithm 5.

Algorithm 5 Virtual IoT Device

```
***Event preparation:***
$CognitoCachingCredentialsProvider$ object setup:
get the context for the application
Identity Pool ID of the Amazon server
Select Region (US-EAST-1 or EU-WEST-1)
$LambdaInvokerFactory$ object instantiate
Invoke $Lambdafunction$ with the corresponding event into the server.
```

4.3 Easy setup with IoT device

After completing successful account authentication, easy setup process starts for the very first time between client application and unboxed IoT device. Client app connects to the unboxed IoT device using a Soft AP network or BT/BLE connection. On the other hand, client app is also connected to the Home router or Home AP that provides public internet. Now client app transfers Home router or Home AP and other information such as Service Set Identifier(SSID), password, security type of Home AP to the unboxed IoT device. Unboxed IoT device connects to Home AP using this information. Client app also transfers cloud access information to the IoT device via Home AP network. Then unboxed IoT device registers to our Amazon cloud and lets client app know that cloud registration is successful. A new entry is added to the cloud table containing IoT device id, client device id, account id, server key, FCM refreshed token etc. after easy setup phase.

4.3.1 Local communication between the client app and IoT device

In an easy setup process, the IoT device first connects to the local network with the client device. Client-server communication protocol is established in between them. As both the client application and the IoT device perform communication by sending and receiving response, both server and client socket mechanism needs to apply to them. In our case, virtual IoT device and client app both implement the below described codes:

Retrieve local network IP list of the connected devices

The IoT device can't connect to the client app because it does not know the IP address. Client app performs a searching operation with it's connected devices and retrieve the IP address of the IoT device. IP address retrieval algorithm 6 is shown below.

Algorithm 6 Retrieve local network IP list from ARP table

```
Prepare a data structure to hold $IPaddress$, $H/Waddress$ and $Deviceinformation$
Read file buffer from $/proc/net/arp$
while read each line of file until $EOF$ do
    Split the line string with $" + "$
    if $splitted = nullANDsplitted.length >= 4$ then
        $IPaddress \leftarrow splitted[0]$
        $H/Waddress \leftarrow splitted[1]$
        $Deviceinformation \leftarrow splitted[3]$
    end if
end while
```

4.3.2 IoT device public network communication with cloud server

An IoT device needs to connect to the internet to register its entry into the server. The client app connects to the IoT device locally and informs it about the wifi network and cloud information. Once the network settings are available, the IoT device can join the home Wi-Fi network. Then it can communicate with the cloud and complete the easy setup process.

4.4 Private cloud: Amazon Web Services

4.4.1 Account authorization

OAuth 2.0 client code is written in Java in the CloudPush framework. The Client app sends account id, password and account type to Amazon cloud. Then Amazon cloud verifies the account information provided by client app with proper account authorization server. Amazon cloud verifies whether provided account information is valid or not. Account authorization server can be Google, Yahoo, Samsung servers etc. Amazon cloud implements only the business layer of account authorization. It returns to client success if provided account is valid or failure otherwise.

Amazon server request the authentication server for access token. Access token is a long string of characters that serves as a credential used to access protected resources. Account registration is successful if authorization server response with a token. This process runs every time, client app login into amazon cloud. Access token retrieval process is shown in algorithm 7.

Algorithm 7 Access Token Retrieval

```
Initialize $HttpPost$ object
$clientId \leftarrow OauthDetails.getClientId()$
$clientSecret \leftarrow OauthDetails.getClientSecret()$
$scope \leftarrow OauthDetails.getClientSecret()$
Prepare the payload of the http post request from OauthDetails
**try**
    Execute http post request
    $response \leftarrow DefaultHttpClient.execute()$
    $code \leftarrow response.getStatusLine().getStatusCode()$
**catch**$(ClientProtocolException)$
    **throw** new RuntimeException ex
**finally**
    $accessToken \leftarrow response.getAccessToken()$
```

4.4.2 Amazon cloud DynamoDB operations

Amazon cloud side DynamoDB operations is shown in algorithm 8.

Algorithm 8 Amazon cloud DynamoDB operations

```
Get $AmazonDynamoDBClient$ object from $DynamoDBController$
Initialize $DynamoDBMapper$ object
try
    Initialize $UserPreference$ object
    $userPreference.setDeviceID \leftarrow deviceID$
    $userPreference.setAccountID \leftarrow accountID$
    $userPreference.setIoTDevID \leftarrow ioTDevID$
    $userPreference.setToken \leftarrow token$
    $userPreference.setServerkey \leftarrow serverkey$
    Assign $UserPreference$ object to the $DynamoDBMapper$ object
    Insert or update row in cloud table
    Delete row in cloud table
    Scan relevant data in cloud table
catch($AmazonServiceException$)
    Print error message
```

Amazon DynamoDB seamlessly scales a single table over hundreds of servers. It has no limit to the amount of data user can store in its table [37–39]. Our IoT system performs insert operation after easy setup phase and scans cloud table when data communication between client applications and IoT devices are needed via the cloud.

4.4.3 Lambda Function in Amazon Cloud Server

We have used AWS Lambda function that works on demand where FCM server logic is implemented in our system. A Lambda instance executes within milliseconds of an event. When an IoT device generates an event, it immediately invokes lambda function to let Amazon cloud know. Then Amazon cloud searches corresponding IoT device id in the cloud table. Following algorithm 9 is showing how lambda function has been implemented and triggering http post request to FCM server:

4.5 Work Flow

4.5.1 Work flow of client application

Client app runs as a background service so that it can receive push notification message sent by IoT devices through Amazon cloud and FCM. Work flow of a client application is shown in Figure 4.1. Client application asks the user for easy setup with an IoT device. If user wants easy setup with an IoT device, then client application asks user for valid account id (Gmail, Yahoo etc.) and password for verifying account with the corresponding authentication server. After successful account verification by cloud, easy setup process starts. A new row is added to

Algorithm 9 Activate push notification into the Amazon cloud

```
Instantiate Lambda function into the Amazon cloud server
try
    $userPreference.setAccountID \leftarrow "https://fcm.googleapis.com/fcm/send"$
    Setup connection with $URL$ object for $httpPOST$ request
    Set $httpPOST$ headers
    Set request property with $FIREBASE - SERVER - KEY$
    Set request property with $Content - Type$
    Send $httpPOST$ body
    Close output stream
    Trigger $httpPOST$ request to the FCM server
catch($Exception$)
    PrintStackTrace
```

the DynamoDB table existed in Amazon cloud if corresponding IoT device row does not exist otherwise row is updated. Easy setup is needed only once between a particular IoT device and an account ID from client app side. In most of the cases, the only login is required to authorize account id by Amazon cloud. As client device id is unique id, after login Amazon cloud checks whether device id's row exists. If corresponding client device's row doesn't exist, the error code is returned from the cloud. As client app starts "FirebaseInstanceIdService", by this time FCM token is available. FCM token is required to identify client device to send push messages by FCM server. Now client app sends unique device id, server key, token to update a row of cloud table.

4.5.2 IoT device sends event to client app through Amazon cloud and FCM

If an event occurs in IoT device, then it sends notification message to Amazon cloud with the specific event code. For example, a vacuum cleaner is an IoT device. After completion of its work, it sends event code to Amazon cloud that its work is done. Then Amazon cloud queries in DynamoDB table and finds out IoT device relevant rows by searching unique universal IoT device id in the table. After finding corresponding rows, Amazon cloud sends the messages to FCM server. FCM server then sends push messages to corresponding client applications with the help of refreshed tokens. Client application registers "FirebaseMessagingService" to receive push notification messages from IoT devices. After receiving the message by a client application, JSON parser parses the message and identify event type that defines in IoT event specification and then client app notifies user by rendering message and event type into the user interface or notification panel. Work flow of an IoT device is shown in Figure 4.2.

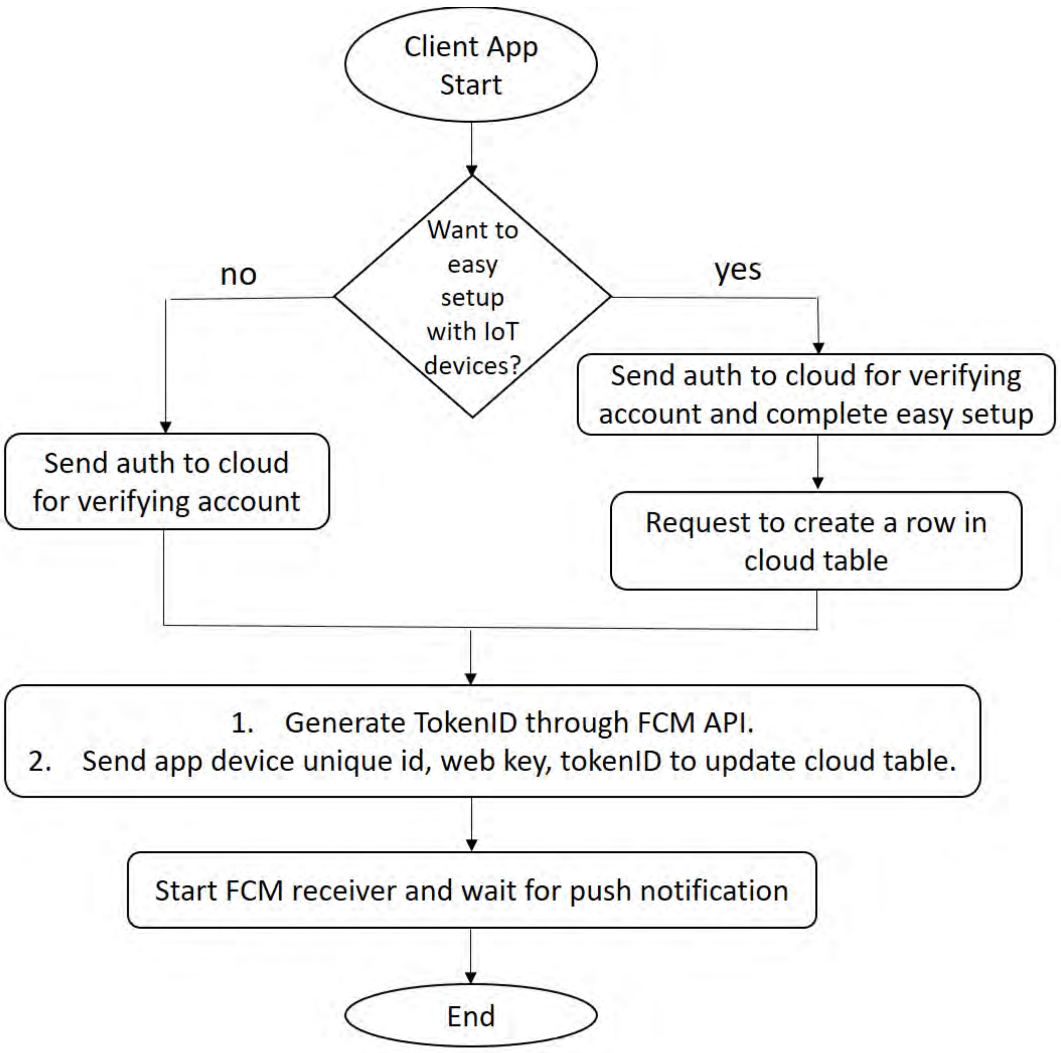

Client App Start
Want to easy setup with IoT devices?
no
yes
Send auth to cloud for verifying account
Send auth to cloud for verifying account and complete easy setup
Request to create a row in cloud table
1. Generate TokenID through FCM API.
2. Send app device unique id, web key, tokenID to update cloud table.
Start FCM receiver and wait for push notification
End

Figure 4.1: Workflow of Client Application

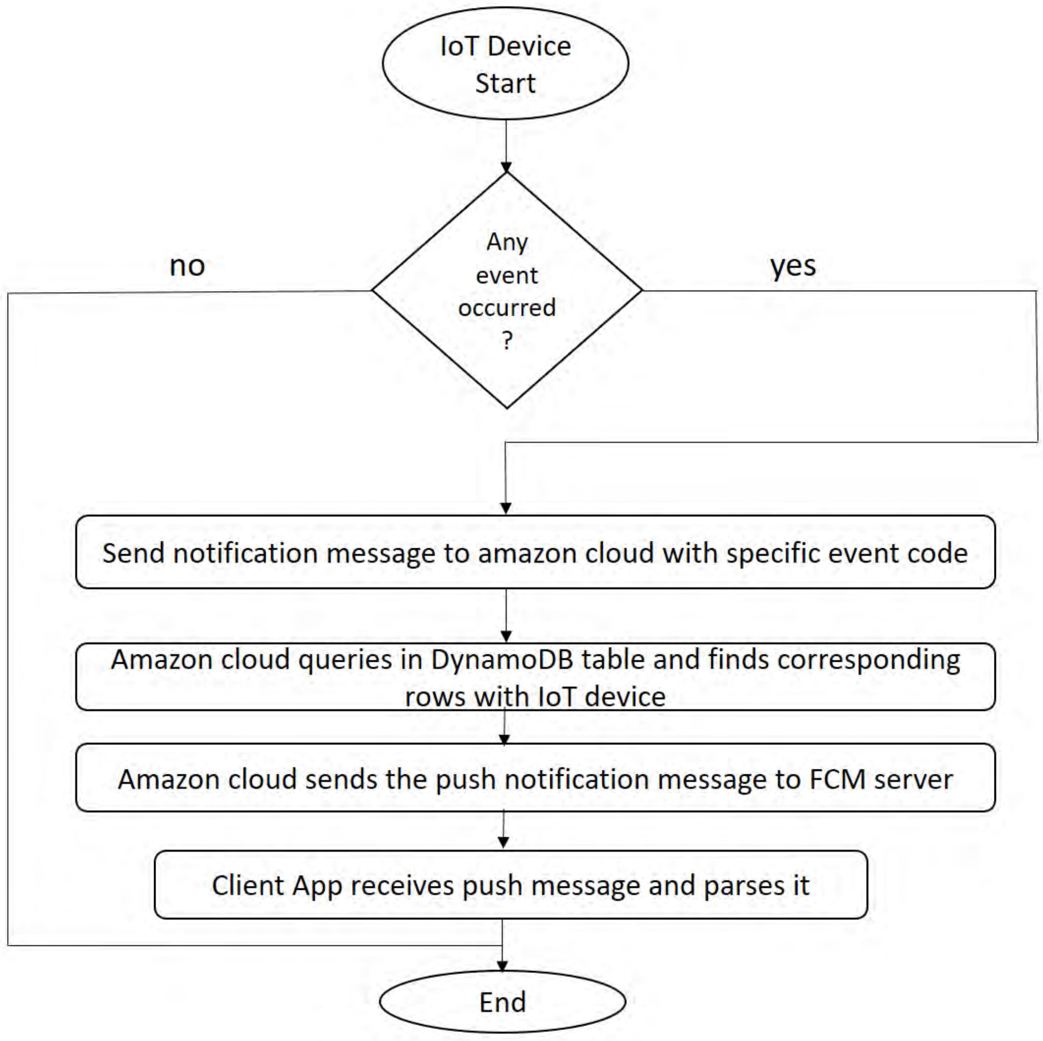

IoT Device Start
Any event occurred ?
no
yes
Send notification message to amazon cloud with specific event code
Amazon cloud queries in DynamoDB table and finds corresponding rows with IoT device
Amazon cloud sends the push notification message to FCM server
Client App receives push message and parses it
End

Figure 4.2: Work flow among IoT device, Amazon cloud and client application.

Chapter 5

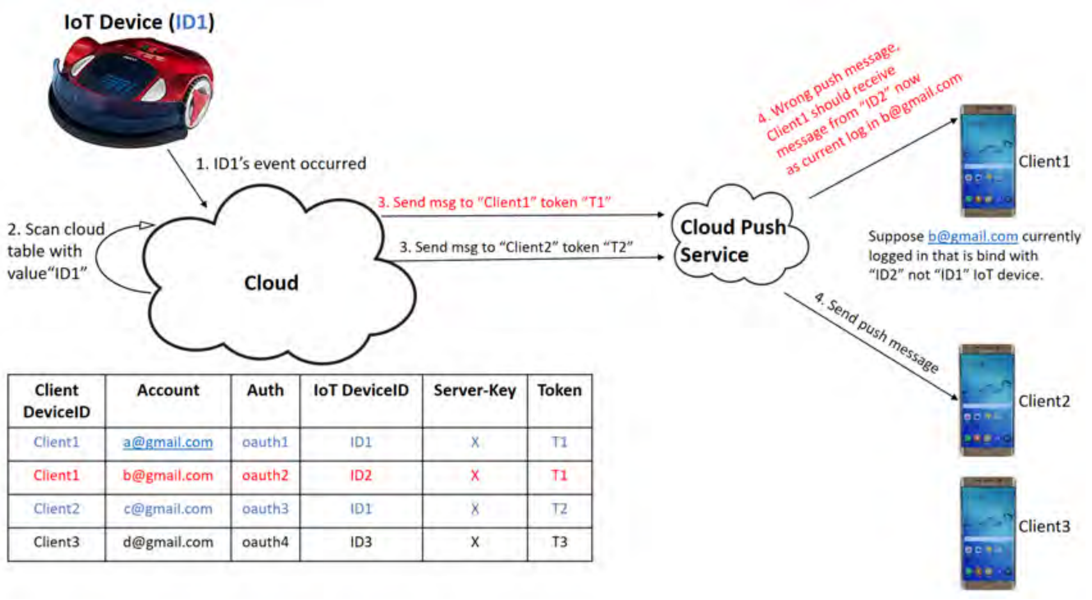
Vulnerabilities of the Proposed Framework

While supporting multi-user feature in the deployment of the proposed framework, we figure out some vulnerabilities that can leak information to outsiders. Client application receives unintended and unauthorized notification messages through cloud from IoT device. As a result, user sensitive data can be exposed by others. Below vulnerabilities of the framework are discussed:

5.1 Hack IoT device information through phishing attack

Phishing attacks involve tricking a victim into taking some action that benefits the attacker. Attacker can track the IoT device credentials by registering the IoT device in the system along with the real user. As, it is possible to receive unauthorized message by the CloudPush system in this case, hacker can perform phishing attack. Details of phishing attack in terms of CloudPush framework are described below with examples.

As CloudPush framework supports multi-user feature, a single client application can use multiple accounts in its device. For a client, only one account id is active at a time. Client application should receive only the notification from IoT devices that are registered with this current account id. Other registered accounts than current should not get notification message triggered by IoT devices. The issue is, client app's current log-in account id is not related to IoT devices but client app can still receive messages from those IoT devices. Unauthorized message received by client app is shown in Fig. 5.1. Client device "Client1" has two entries in the cloud table. "Client1" is registered with IoT Device "ID1" and "ID2". "ID1" is registered with account "a@gmail.com" and "ID2" is registered with account "b@gmail.com" by "Client1". Suppose, "Client1" is currently logged in with account "b@gmail.com". Now "Client1" should not receive any push messages from IoT device "ID1" as it is registered with different account. As shown in Fig. 5.1, IoT device "ID1" generates an event and "Client1" is currently logged in with account


IoT Device (ID1)
1. ID1's event occurred
2. Scan cloud table with value"ID1"
Cloud
3. Send msg to "Client1" token "T1"
3. Send msg to "Client2" token "T2"
Cloud Push Service
4. Wrong push message, Client1 should receive message from "ID2" now as current log in b@gmail.com
4. Send push message
Client1
Suppose b@gmail.com currently logged in that is bind with "ID2" not "ID1" IoT device.
Client2
Client3

Client DeviceID	Account	Auth	IoT DeviceID	Server-Key	Token
Client1	a@gmail.com	oauth1	ID1	X	T1
Client1	b@gmail.com	oauth2	ID2	X	T1
Client2	c@gmail.com	oauth3	ID1	X	T2
Client3	d@gmail.com	oauth4	ID3	X	T3

Figure 5.1: Unauthorized message received by Client app

"b@gmail.com". Cloud scans cloud table with value "ID1" and sends an unauthorized push message to "Client1" through cloud push service. In this case, "Client1" should not receive any messages from "ID1" as currently logged in account is "b@gmail.com".

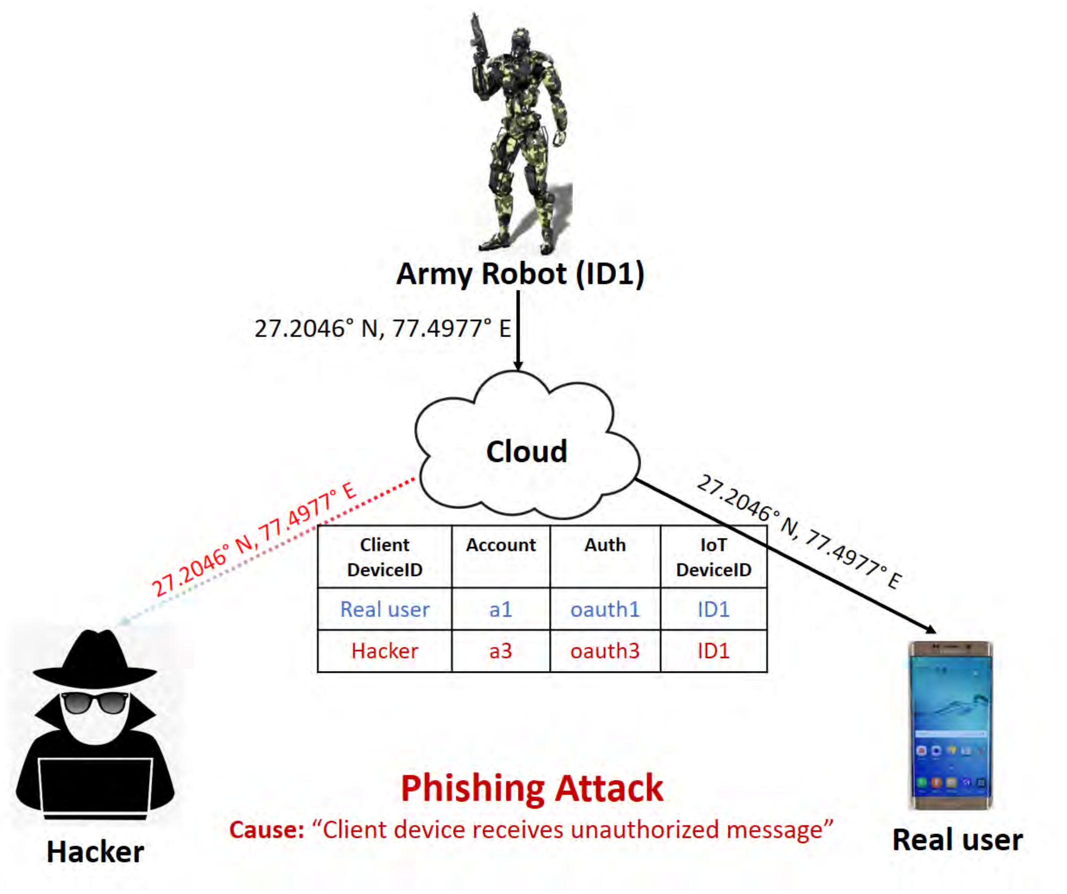
In the same way, hacker can register his account "a3" with IoT device "ID1", an army robot. Above described way, hacker can trace the location information of the army robot as shown in Fig. 5.2. When the robot sends its location information to the real user, hacker is also getting the same information due to the security flow of the system.

5.2 Denial of service attack (DoS attack) by triggering unlimited push notification messages

A denial-of-service (DoS) attack occurs when users are unable to access information systems, IoT devices, or other network resources due to the actions of a malicious cyber threat actor. The possibility of DoS attack in the CloudPush framework is described below.

When a single client application device with multiple accounts register with a single IoT device then corresponding multiple rows are inserted in cloud table. For multiple accounts (multiple rows in cloud table) with a single IoT device, a client application may get multiple messages wrongly. Two scenarios are discussed below. For the correct scenario, client devices receive push messages properly but for an incorrect scenario, a client receives multiple messages for a single IoT device's event instead of a single message.

Correct scenario: Multiple client devices with multiple accounts: We show an example in Fig. 5.3 where multiple client devices are registered with a single IoT device. Suppose, IoT



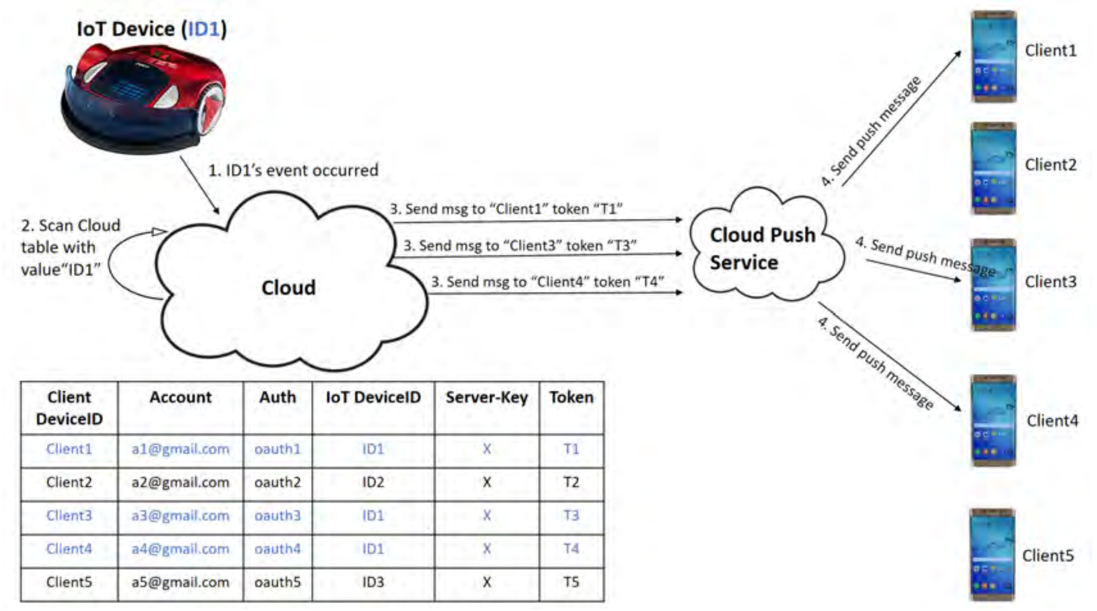
Client DeviceID	Account	Auth	IoT DeviceID
Real user	a1	oauth1	ID1
Hacker	a3	oauth3	ID1

Figure 5.2: Phishing attack by the hacker

device, ID1 generates an event. Cloud starts searching cloud table's corresponding rows with IoT DeviceID. Three entries of cloud table have been found that match with "ID1". As cloud push service is implemented in the cloud, it pushes messages for "Client1" with token "T1", "Client3" with token "T3" and "Client4" with token "T4". For example, vacuum cleaner completes its task and wants to notify client applications that are registered with it. Three client devices are registered with the vacuum cleaner. So all the three client devices get notified through cloud.

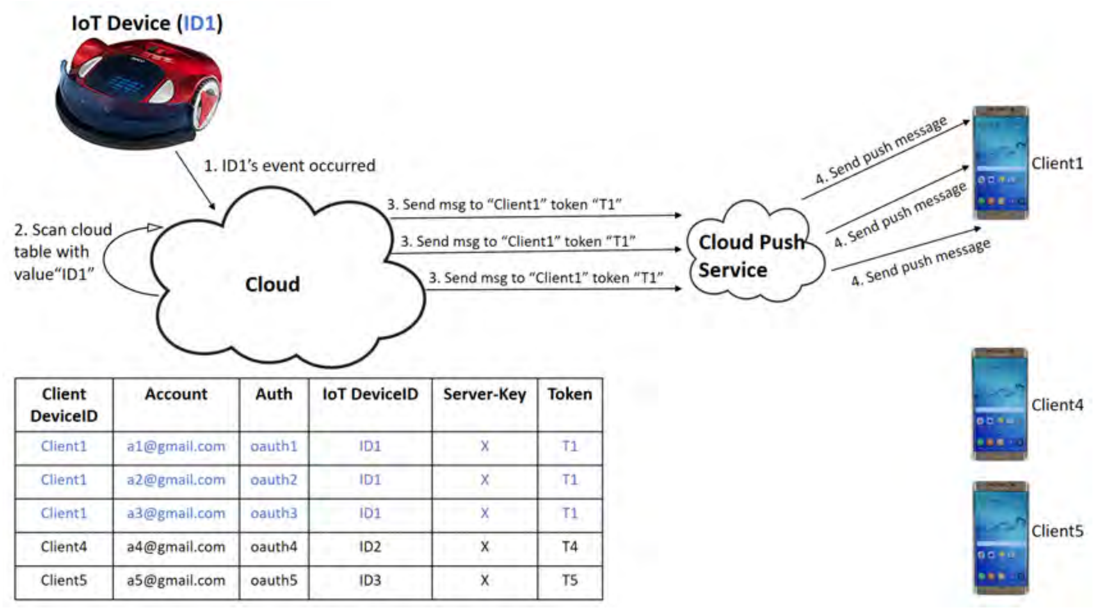
Incorrect scenario: Single client device with multiple accounts: In Fig. 5.4, it is shown that single client device receives multiple notification messages from an IoT device for a single generated event. Suppose, a single device supports multiple accounts for registering with a single IoT device. "Client1" device registers three different accounts with a single IoT device "ID1". As same client device, refreshed token value is same "T1". If IoT device "ID1" generates an event, Cloud table inside cloud server finds three rows with different account id but same client device id and token value. Cloud sends three messages to cloud push service with token value "T1". Then cloud push service delivers three push messages to "Client1". For a single IoT device event, client app is receiving three push messages through cloud. This is an issue in our proposed IoT push notification framework.

In Fig: 5.5, hacker registers many accounts with the same IoT device and user device by



Client DeviceID	Account	Auth	IoT DeviceID	Server-Key	Token
Client1	a1@gmail.com	oauth1	ID1	X	T1
Client2	a2@gmail.com	oauth2	ID2	X	T2
Client3	a3@gmail.com	oauth3	ID1	X	T3
Client4	a4@gmail.com	oauth4	ID1	X	T4
Client5	a5@gmail.com	oauth5	ID3	X	T5

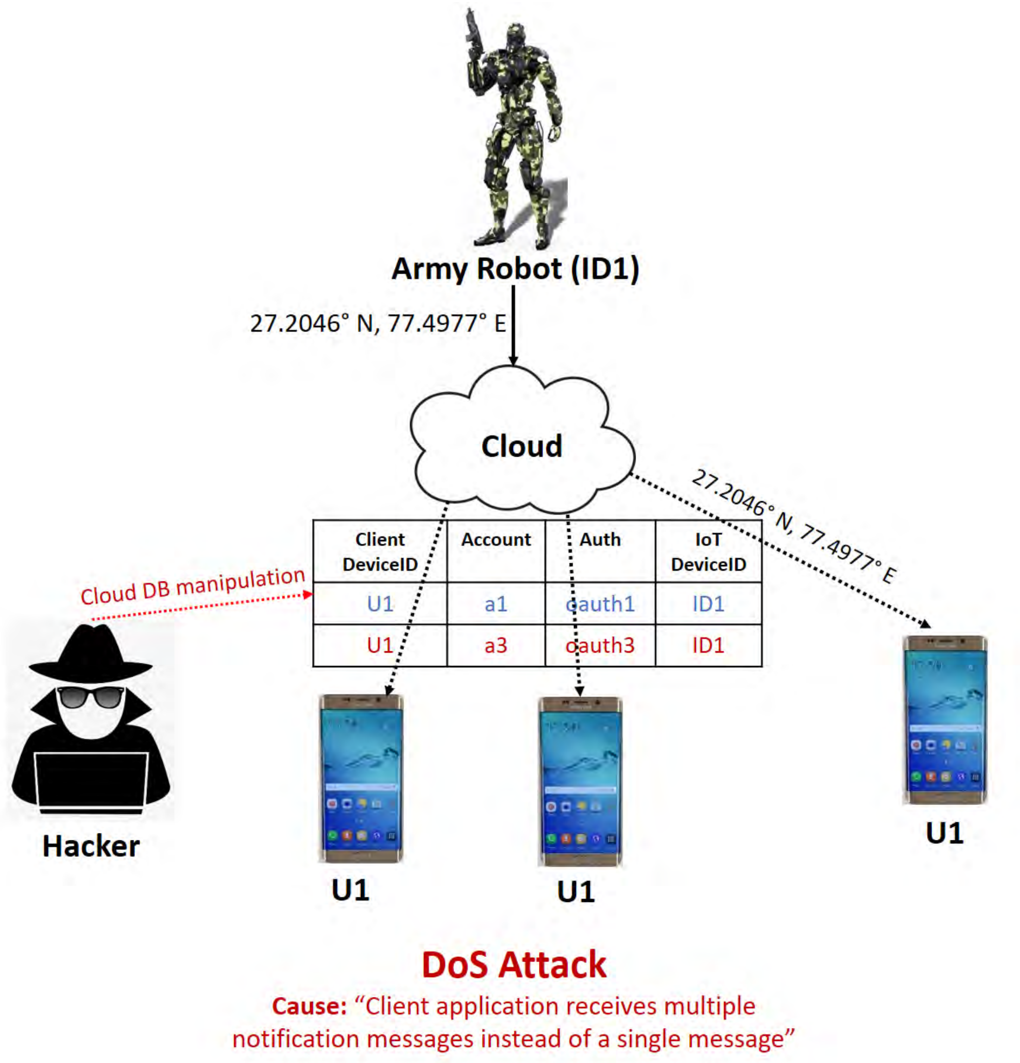
Figure 5.3: Multiple devices with multiple accounts in CloudPush Framework



Client DeviceID	Account	Auth	IoT DeviceID	Server-Key	Token
Client1	a1@gmail.com	oauth1	ID1	X	T1
Client1	a2@gmail.com	oauth2	ID1	X	T1
Client1	a3@gmail.com	oauth3	ID1	X	T1
Client4	a4@gmail.com	oauth4	ID2	X	T4
Client5	a5@gmail.com	oauth5	ID3	X	T5

Figure 5.4: Single device with multiple accounts issue in CloudPush Framework

launching a bot or manipulating cloud database. In this scenario, we consider that the hacker somehow manage to get the user device "U1" credentials. When the army robot triggers an event, communication channels can be overloaded due to huge number of push message requests send through the server to the client device and the system can be unresponsive as well as army robot can be unavailable or inaccessible by the CloudPush system.

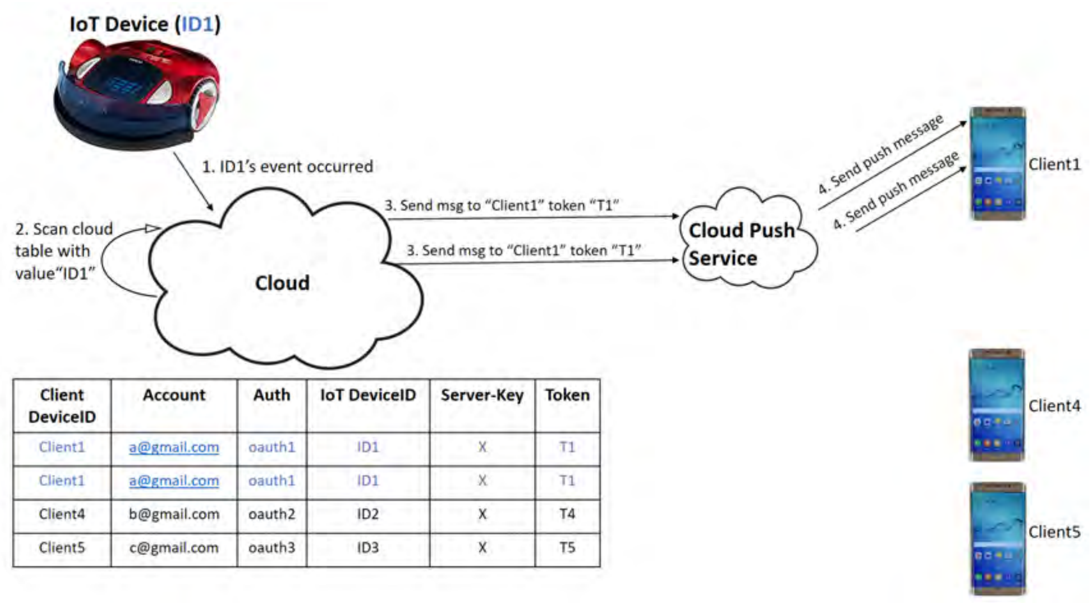


Client DeviceID	Account	Auth	IoT DeviceID
U1	a1	oauth1	ID1
U1	a3	oauth3	ID1

Figure 5.5: DoS attack by the hacker

5.3 Hacker controls the IoT Device: Man in the Middle (MITM) attack

Hacker pretends to be as a real user by manipulating cloud database or getting real user's credentials and controls the IoT device. It is possible when the same client device ID exists in multiple rows in the cloud table. If Client app and same IoT device register for multiple times in cloud somehow, then multiple cloud table's rows can be inserted. As shown in Fig. 5.6 "Client1" registers with an IoT device "ID1" in cloud twice. That's why two rows are inserted in cloud table with account id "a@gmail.com" and token "T1". Suppose, "ID1" generates an event. Cloud scans cloud table with value "ID1" and sends two event messages with token "T1" to cloud push service. Then cloud push service sends two push messages to "Client1" as token "T1" is associated with "Client1" device.

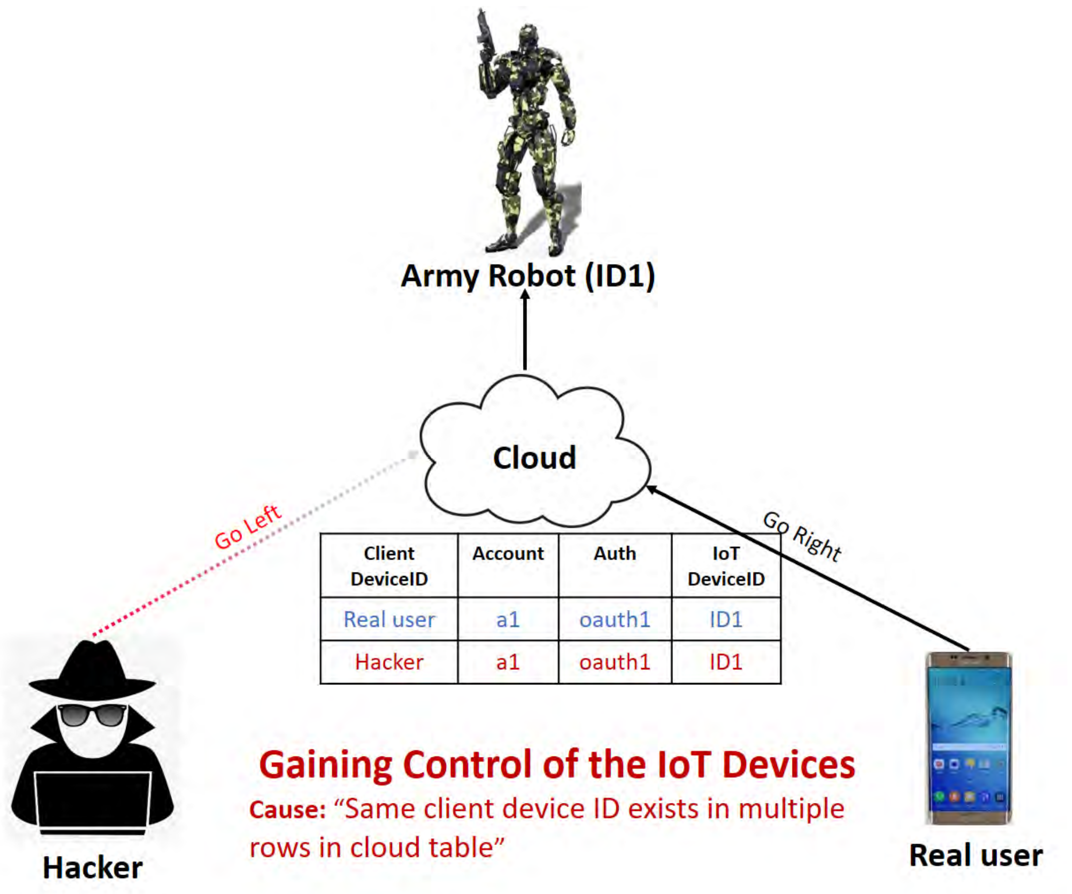


Client DeviceID	Account	Auth	IoT DeviceID	Server-Key	Token
Client1	a@gmail.com	oauth1	ID1	X	T1
Client1	a@gmail.com	oauth1	ID1	X	T1
Client4	b@gmail.com	oauth2	ID2	X	T4
Client5	c@gmail.com	oauth3	ID3	X	T5

Figure 5.6: Single event generated by IoT device and client device receives multiple messages

As per above scenario, hacker takes the control of the real user device or manipulating cloud database. As a result, same client device id exists in multiple rows in the cloud table. Now hacker is able to control the army robot as per his intention.

In Fig: 5.7 and 5.8, army robot believes that it is communicating with the real user. Attacker makes independent connections with the army robot and relays messages between them to make it believe they are talking to each other, when in fact the entire conversation is controlled by the attacker. This enables an attacker to intercept information and data from the CloudPush framework.



Client DeviceID	Account	Auth	IoT DeviceID
Real user	a1	oauth1	ID1
Hacker	a1	oauth1	ID1

Gaining Control of the IoT Devices

Cause: “Same client device ID exists in multiple rows in cloud table”

Figure 5.7: IoT device is controlled by the hacker

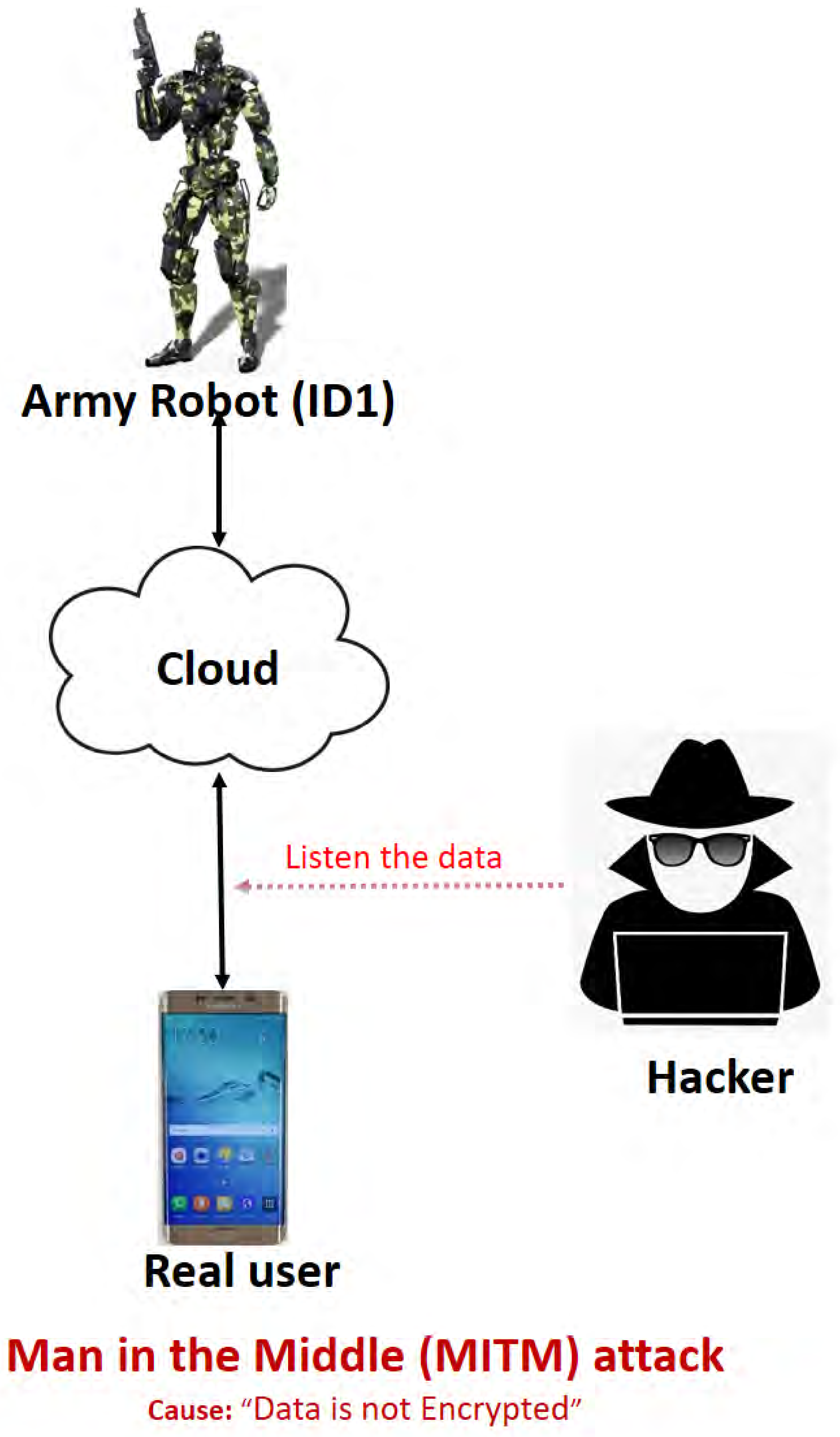

Army Robot (ID1)
Cloud
Listen the data
Hacker
Real user
Man in the Middle (MITM) attack
Cause: "Data is not Encrypted"

Figure 5.8: Man in the Middle (MITM) attack

Chapter 6

Solutions

We solve CloudPush framework vulnerabilities by encrypting notification message data and implementing in cloud database layer by checking with currently active client account id.

6.1 Push notification message data

6.1.1 Key generation

Cloud encrypts notification message data using a key generated using the algorithm 10 and shares the key with corresponding client application. When client application receives the encrypted message from cloud, it decrypts the message using the key. As the key is generated based on account id and client device id, it is not possible for other devices to extract the unauthorized messages. This will help to prevent MITM attack of the CloudPush framework.

Algorithm 10 Key generation in cloud

```
  input: $AccountID, ClientDeviceID$
  $prime1 \leftarrow GetPrimeNumber(AccountID)$
  $prime2 \leftarrow GetPrimeNumber(ClientDeviceID)$
  $key \leftarrow GenerateKey(prime1, prime2)$
  return Key
```

6.1.2 Key encryption

We use password-based encryption (PBE) [40] ciphers to encrypt the key and data that require an initialization vector (IV) can obtain it from the key, if it's suitably constructed, or from an explicitly-passed IV. When passing a PBE key that doesn't contain an IV and no explicit IV, the PBE ciphers currently assume an IV of zero. When using PBE ciphers, it is required to pass an explicit IV, as shown in the following algorithm 11:

Algorithm 11 Key encryption

```
input: $SecretKey, IV$
$Cipher \leftarrow Get\_instance\_of\_PBE\_cipher\_with\_SHA - 256\_hashing")$
$IV \leftarrow Generate\_random\_bytes$
$Cipher \leftarrow Initialize\_the\_cipher$
return Encrypted Key
```

Table 6.1: Prevent receiving multiple messages by adding a column named "Current Account" in cloud table

Client DeviceID	Account	Auth	IoT DeviceID	Server-Key	Token	Current Account
Client1	A1@gmail.com	Abc	ID1	X1	T1	N
Client1	B1@gmail.com	Xyz	ID1	X1	T1	Y
Client1	C1@gmail.com	Pqr	ID1	X1	T1	N

6.2 Active client account id verification in cloud

We prevent unnecessary and unauthorized notification message sending to client side by keeping track of active client account id in cloud. We keep another column into the cloud table that maintains which one is currently log-in account. A user can choose the current log-in account from a client application and request to update corresponding row in cloud table or last log-in account is the current account. When an event is generated by IoT device and the event is sent to cloud, "Current Account" column in the cloud table is checked with corresponding client device ID. Only current account "Y" related to client device get notification message if relevant IoT device generates an event. In Table 6.1, client device "Client1" registers with IoT device "ID1" and three accounts. Suppose "B1@gmail.com" is the current log-in account. "ID1" generates an event and sends its event to cloud. Cloud finds "ID1"s corresponding row by checking "Current Account" column "Y" and sends one notification message to client application "Client1". So the problem of receiving multiple messages is solved here.

In Table 6.2, client device "Client1" registers with two different IoT devices "ID1" and "ID2" with two different accounts "A1@gmail.com" and "B1@gmail.com". "B1@gmail.com" account is currently logged in and bounded with "ID2". If "ID1" generates an event, there is no current login account exist in the cloud database. Hence no notification message is sent to "Client1". Thus "Current Account" column solves unauthorized message receiving problem too.

Table 6.2: Prevent receiving unauthorized message by adding a column named "Current Account" in cloud table

Client DeviceID	Account	Auth	IoT DeviceID	Server-Key	Token	Current Account
Client1	A1@gmail.com	Abc	ID1	X1	T1	N
Client1	B1@gmail.com	Xyz	ID2	X1	T1	Y
Client1	C1@gmail.com	Pqr	ID1	X1	T1	N

6.3 Maintain unique client device ID

To prevent creating a similar row for multiple times, we use unique client ID instead of random device ID. This will ensure that same client device with same account id and same IoT device cannot be registered more than once.

Every device has a unique id. In case of Android device, "TelephonyManager" class [41] has an API "getDeviceId()" that provide unique device id. To keep unique client device ID for different accounts, the following rule is used:

$ClientDeviceID \leftarrow Device.MAC$

Same client application device with multiple accounts situation in cloud table is shown in Table 6.3.

Table 6.3: Maintain Client device id uniqueness by adding client device id with account id.

Client DeviceID	Account	Auth	IoT DeviceID	Server-Key	Token	Current Account
Client1 + A1	A1@gmail.com	Abc	ID1	X1	T1	N
Client1 + B1	B1@gmail.com	Xyz	ID1	X1	T1	Y
Client1 + C1	C1@gmail.com	Pqr	ID1	X1	T1	N

6.4 Remove history records from cloud database

Another solution to prevent receiving multiple or unauthorized messages is that remove all other records related to client app device except currently login account row just added in the cloud table. This will make sure only one entry exists in cloud table related to the client application device and the IoT device. Every time, client device logs into cloud, it creates a new row and removes other rows related to client device id if it exists. This approach causes loss of previous data as each time signing into the cloud, a new row is added and old rows are removed with client device id. Moreover every time row insertion and deletion is costly and time-consuming operation.

6.5 Optimized searching algorithm in cloud

We apply optimized runtime searching algorithm in cloud that find any component from cloud database in $O(search\_key\_length)$ time complexity. In Cloudpush framework, search items can be (1) Client DeviceID (2) Account ID (3) IoT DeviceID. These three items are indexed in a search tree. A search tree structure is formed in Fig: 6.1 based on the cloud table 6.5 using these three search items. Leaf of the tree contains a object associated with the cloud table row. When an item is searched from the tree, it can easily retrieve the corresponding row of the cloud table. It saves a significant amount of time while querying and improves throughput of CloudPush

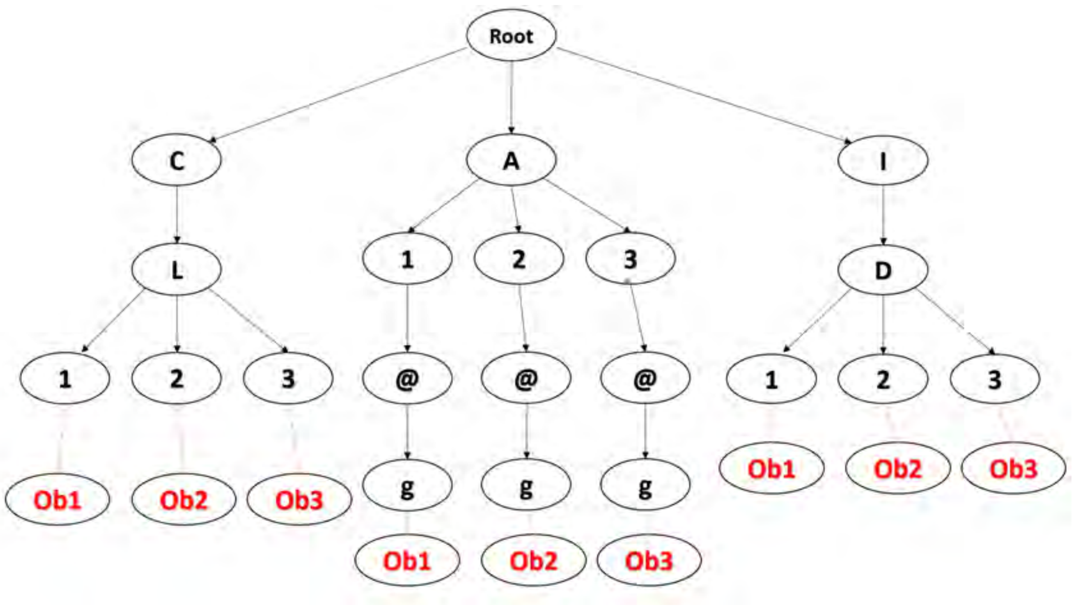


Figure 6.1: Cloud items index tree for searching cloud records efficiently

framework.
Let w be the total amount of characters in the search items. It simply represents the max amount of unique characters, N in the customized tree, that is space boundary.

- Search key length = L
- Total number of unique characters in the search items = N
- Total number of row in the table = K
- Total number of search items = I

Search algorithm efficiency is shown in table 6.4 in big O notation.

Table 6.4: Efficiency of the algorithm.

#	Notation
Time complexity	$O(L)$
Space complexity	$O(N + K * I)$

Table 6.5: Cloud data table for figure: 6.1.

Object	Client DeviceID	Account	Auth	IoT DeviceID	Server-Key	Token	Current Account
Ob1	CL1	A1@g	Abc	ID1	X1	T1	N
Ob2	CL2	A2@g	Xyz	ID2	X1	T1	Y
Ob3	CL3	A3@g	Pqr	ID3	X1	T1	N

Chapter 7

Experimentation

7.1 Performance comparison between CloudPush framework and IoTivity framework

7.1.1 Performance of IoTivity Framework

Testbed Setting, Testing Scenarios and Results

In this paper [16], testbed consists of four physical nodes as shown in Fig. 7.1. IoT device node is aMid-2010 MacBook Pro with 4GB RAM and an Intel Core 2 duo CPU 2.4 GHz. IoT controller node is hosted in a laptop computer with 8GB RAM and an Intel Core i5 duo CPU 2.3 GHz. The CI is hosted in a desktop computer with 3 GB RAM and an Intel Core i5-2500 CPU 3.3 GHz. The IGC with three components of MQ, Account, and RD servers is set up in a single desktop computer. The computer has 16 GB RAM and Intel Core i7 CPU 3.60 GHz. All the computers run the Ubuntu 16.04 LTS operating system, and support IoTivity framework 1.2.0. The IoTivity Cloud is deployed on the testbed similar to its architecture. They have added Java hooking scripts into source code of the CI and the MQ to measure the performance of these servers. The size of requests/responses are set in the simplest case where a message contains only one character. The number of devices are varied to identify maximum capacity of the system.

Figure 7.2 illustrates similar trends of throughputs with different numbers of IoT devices. It gradually increases and reaches the maximum value before starting decreasing as the number of requests continues to increase. The increase of throughput can be justified by the fact that a greater number of requests is introduced into the system and the system uses more CPU utilization for handle those requests. On the other hand, the throughput decreases when the number of requests is large enough and impacts the processing capacity of the system. The throughput increases from 252269.2 (msg/sec) for one IoT device setup to 465519.2 (msg/sec) for 10 IoT devices setup, to 562967.7 (msg/sec) for 100 IoT devices setup and to 595406.2 (msg/sec) for 500 IoT devices setup. In other words, the throughput increases at 343137 (msg/sec) which

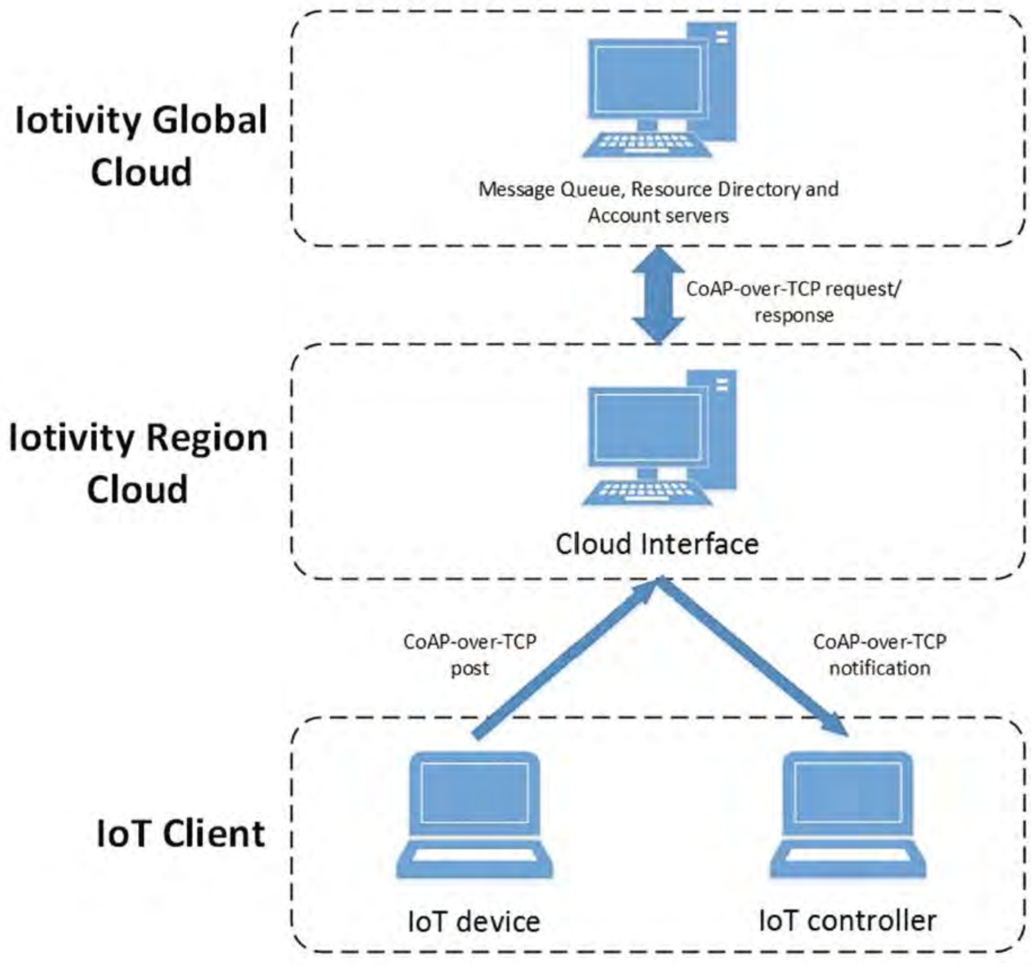

Iotivity Global Cloud
Message Queue, Resource Directory and Account servers
CoAP-over-TCP request/ response
Iotivity Region Cloud
Cloud Interface
CoAP-over-TCP post
CoAP-over-TCP notification
IoT Client
IoT device
IoT controller

Figure 7.1: Testbed model.

is almost a 57.63% increase. However, the maximum throughput lightly decreases to 583455.1 (msg/sec) with the 1000 IoT devices setup. The maximum throughput of the CI reaches 595406.2 (msg/sec) for 500 IoT devices setup in the scope of our testbed configuration.

As shown in Fig.7.3, throughput of the MQ has a similar trend to that of the CI. However, throughputs for setups with different numbers of IoT devices converge to approximate 3640 (msg/sec) when the number of requests passes 4000 (msg/sec). The maximum throughput increases from 4266.4 (msg/sec) for one IoT device setup to 4926.8 (msg/sec) for 10 IoT devices setup before going down to about 4772.2 (msg/sec) with 100 IoT devices setup and 500 IoT devices setup, and to 4610.5 (msg/sec) with 1000 IoT devices setup. The maximum throughput of the MQ reaches 4926.8 (msg/sec) which is much smaller than the throughput of the CI. This can be justified by the fact that the CI is tasked to only forward requests of IoT devices to the MQ while the MQ has to detaches the requests and then constructs the Kafka messages before pushing them into the Kafka message queue. In case the number of generated requests is less than the number of IoT devices, there will be some IoT devices do not generate requests but

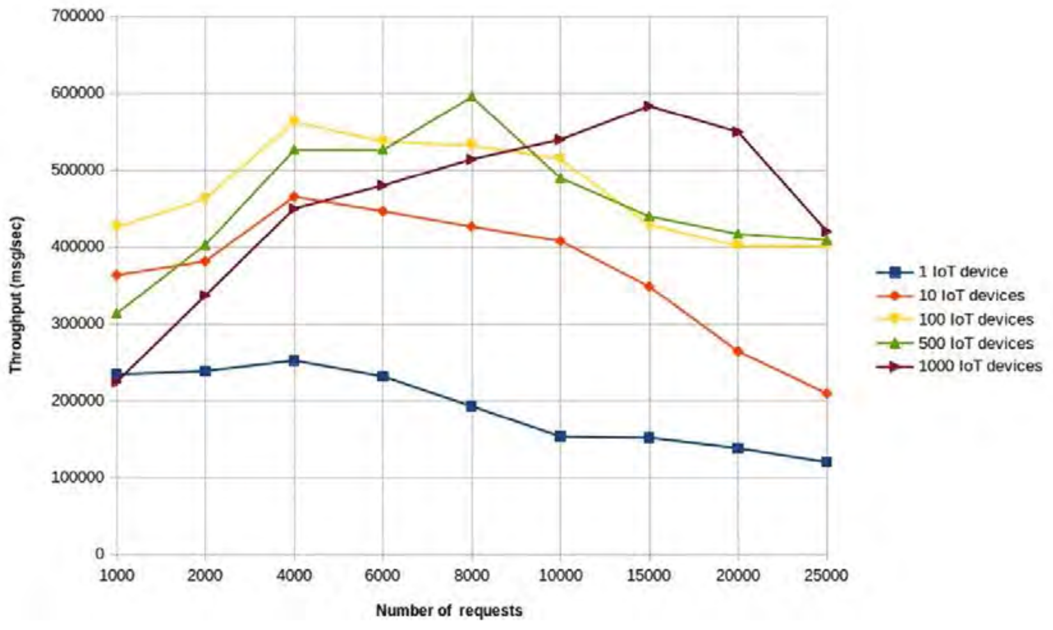


Figure 7.2: Throughput of CI server.

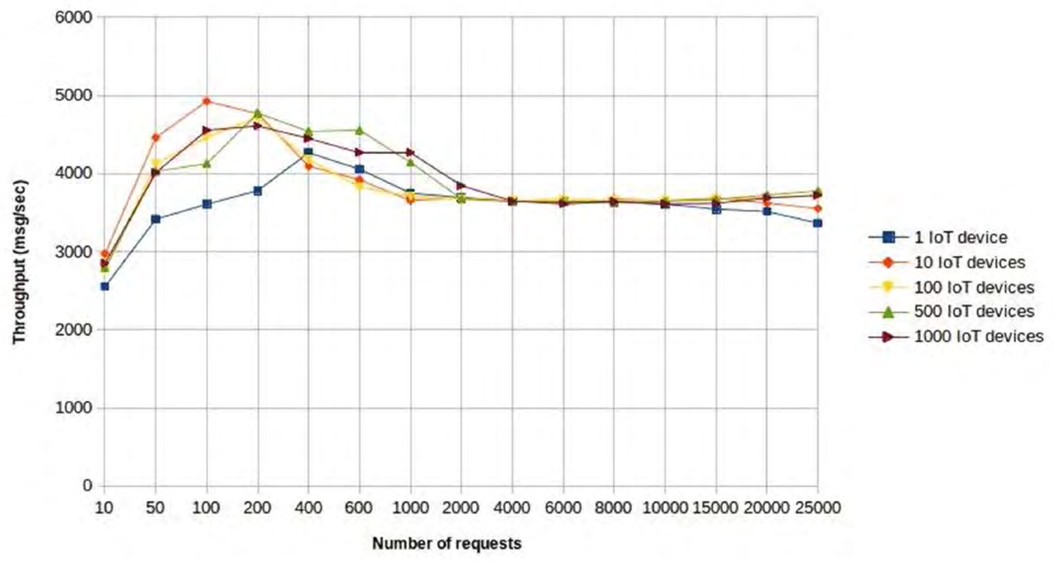


Figure 7.3: Throughput of the MQ (no IoT controllers).

we still create these IoT devices (C++ threads) and make them connect to the CI to experience various testing cases.

The next scenario is the generalized case of the above described scenario in terms of the functionality. What is added into this scenario is the different number IoT controllers which are 1, 10, 50, 100, 200 for different tests. Unlike the simulation method of IoT devices, the different numbers of IoT controllers are introduced by using independent processes instead of using threads. A process is a compiled C++ program which is implemented by using IoTivity

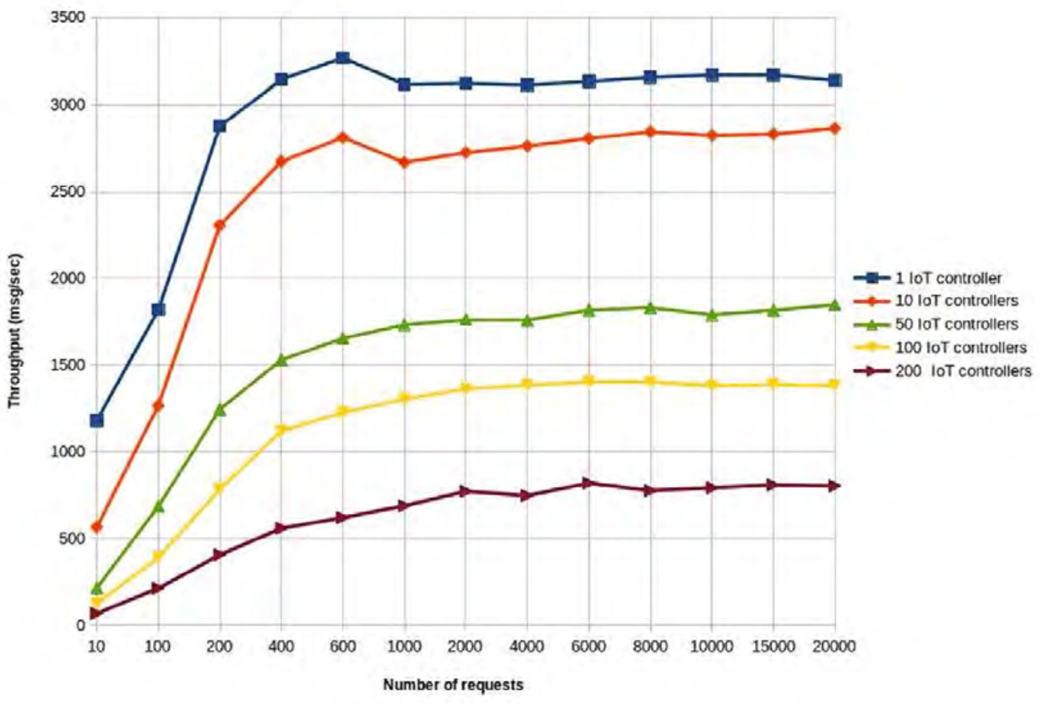


Figure 7.4: Throughput of the MQ IoT controllers

framework APIs. All IoT controllers subscribe to the same topic of the MQ; therefore, they receive the same notification when a post request is published to the MQ broker. The purpose of using different processes is that we want to duplicate the notification corresponding to the number of subscribers. This cannot be done with a single process having multiple threads to our knowledge about IoTivity framework APIs. Testing result from the first scenario shown that throughput of the CI is much higher than throughput of the MQ so the bottleneck of the throughput of the overall architecture is at the MQ. For this reason, we only focus on measuring the throughput of the MQ in this scenario. Testing results of the second scenario are shown in the Fig. 7.4.

The testing results of IoTivity framework show that the CI can handle a load of 1000 IoT devices simultaneously and the maximum throughput reaches 595406.2 (msg/sec). The maximum throughput of the MQ is 4926.8 (msg/sec) which is much smaller than the throughput of the CI. What can be learned from this is the overall throughput of the architecture is mainly depended on the performance of the MQ. In addition, the throughput of the MQ drops significantly when we introduce a large number of IoT controllers into the system.

7.1.2 Performance of CloudPush Framework

There are two variants:

1. Number of events generated by IoT devices.

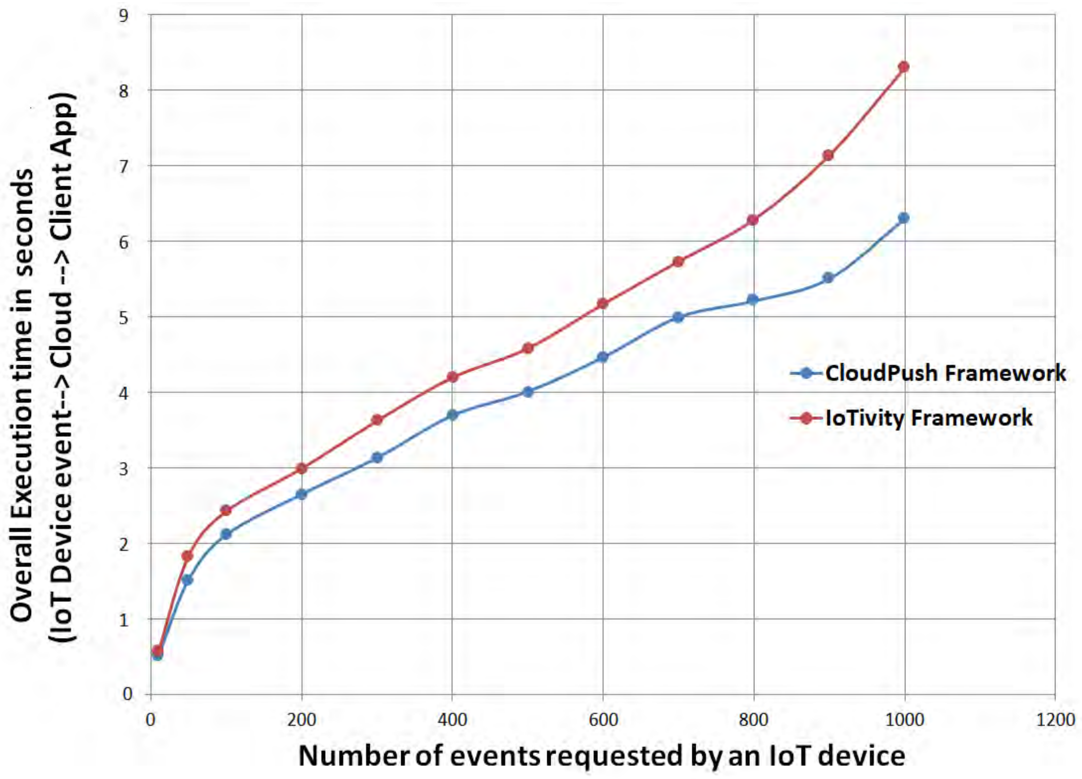


Figure 7.5: Aggregated throughput comparison between CloudPush framework and IoTivity framework by a varying number of events generated by an IoT device [variant: Events].

2. Number of client application devices received messages, are considered while comparing aggregated throughput between CloudPush framework and IoTivity framework.

We observe that overall execution time of an IoT device generated events in CloudPush framework is 12-15% better than IoTivity framework shown in Fig. 7.5. In this experiment, one IoT device generates all the events and a client application receives all of the events. Overall execution time increases if number of events generation by the IoT device increases for both frameworks.

If we vary both number of events generated by IoT devices and number of notification message receivers (client application devices) then performance of CloudPush framework increases to 12-25% shown in Fig. 7.6. Events are distributed among client applications in this experiment. Each IoT device generates equal number of events and these events are received by equal number of client applications shown in the table 7.1. CloudPush framework performs better because we use separate cloud server and cloud push service layer that are scaled up depending on number of events requested by IoT devices and number of client application devices involved with the events during execution time and run optimized searching algorithm in cloud server.

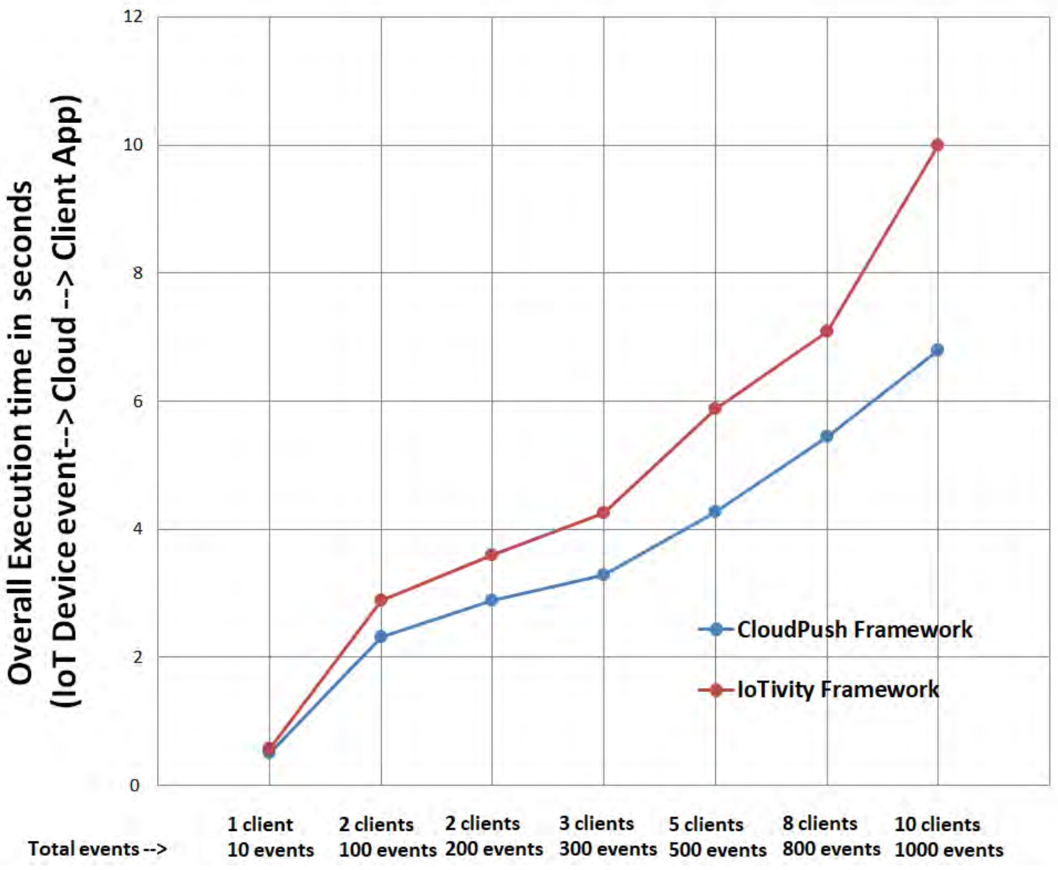


Figure 7.6: Aggregated throughput comparison between CloudPush framework and IoTivity framework by a varying number of events generated by IoT devices and message receivers [variants: Events, client application devices].

Table 7.1: Distributed events among receivers (Client devices).

No. of IoT device	Events(Per IoT device)	Total events	No. of Client (Receiver)
1	10	10	1
2	50	100	2
2	100	200	2
3	100	300	3
5	100	500	5
8	100	800	8
10	100	1000	10

Let, execution time to perform a full cycle of an event operation by CloudPush framework is T_O and by IoT framework is T_I then the equation for performance measurement for CloudPush framework is,

$$P = \frac{T_I - T_O}{T_O} \times 100\% \tag{7.1}$$

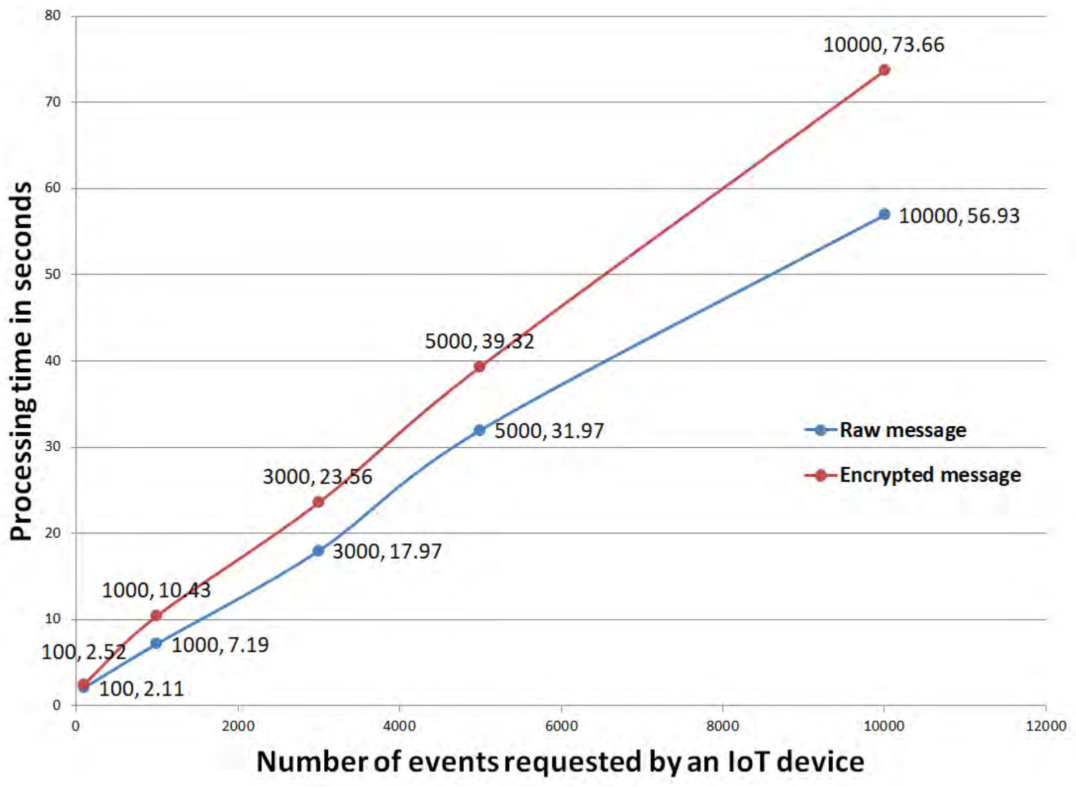


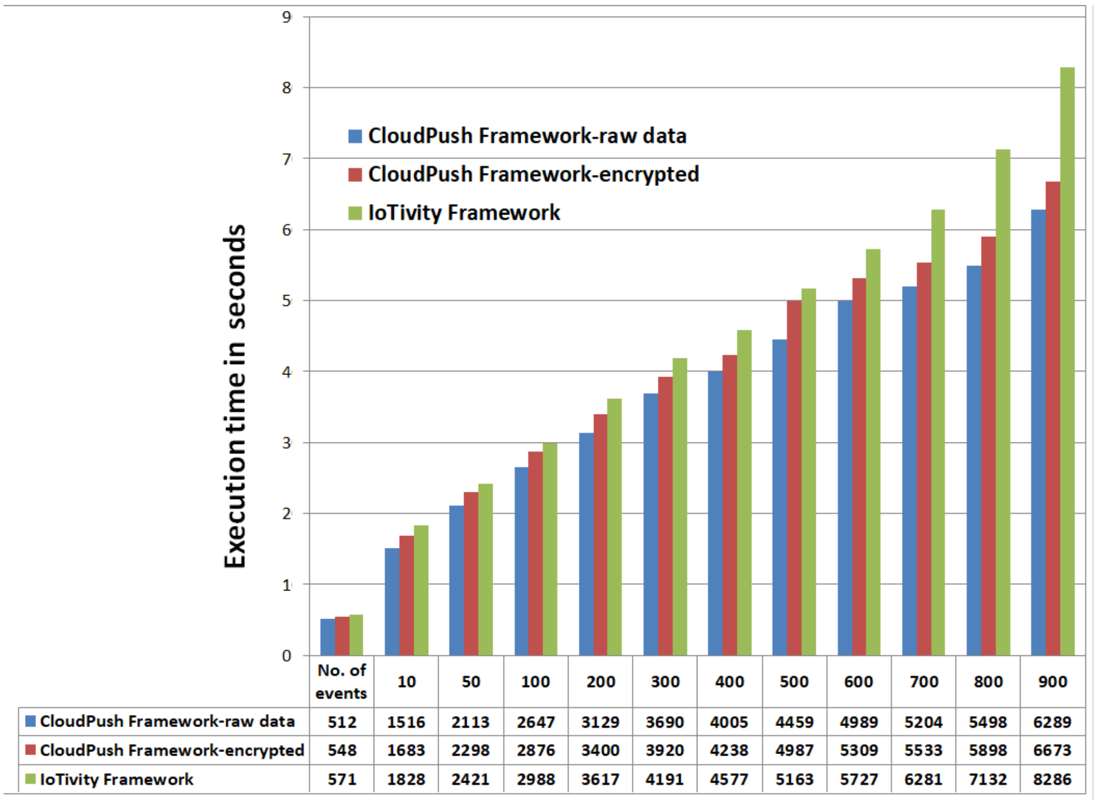
Figure 7.7: CloudPush framework performance comparison after applying encryption in notification message.

7.2 Performance comparison after applying notification message encryption

After applying notification message encryption solution in CloudPush framework, it is observed that system performance is reduced to 3-5% compared to without encrypted notification message in CloudPush framework shown in Fig. 7.7. Notification message is encrypted for a secured multi-user support but it makes existing CloudPush framework with raw message 3-5% slower but performs 9-12% better than IoTivity framework for a single client. If events are distributed among multiple clients than performance improves about 9-23% than the IoTivity framework. Performance comparison between IoTivity framework and CloudPush framework for encrypted and raw notification message solution is shown in table 7.2. As cloud server and client application need to process notification message data, CloudPush framework gets little slower. Cloud server converts a raw message to an encrypted message and sends to a client. A client device decodes encrypted the notification message with a key shared earlier by the cloud.

Table 7.2: Performance improvement in CloudPush framework compared with IoTivity framework

Variants	Improvement without message encryption	Improvement with message encryption
Events	12-15%	9-12%
Events + Client applications	12-25%	9-23%



	No. of events	10	50	100	200	300	400	500	600	700	800	900
🟦 CloudPush Framework-raw data	512	1516	2113	2647	3129	3690	4005	4459	4989	5204	5498	6289
🟥 CloudPush Framework-encrypted	548	1683	2298	2876	3400	3920	4238	4987	5309	5533	5898	6673
🟩 IoTivity Framework	571	1828	2421	2988	3617	4191	4577	5163	5727	6281	7132	8286

Figure 7.8: Aggregated events processing time comparison among CloudPush system encrypted and not encrypted notifications and IoTivity framework.

Overall aggregated events processing time comparison among CloudPush system encrypted and not encrypted notifications and IoTivity framework is shown in Fig. 7.8. It is shown that without message encryption, CloudPush throughput is better than encrypted message in CloudPush and IoTivity framework. Even applying message encryption in CloudPush framework, throughput is better than IoTivity framework.

Chapter 8

Conclusions

Concerns have been raised that new IoT platforms are being developed rapidly and the profound security challenges involved in these IoT environments are not taken into appropriate consideration. This thesis introduces a push messaging framework named CloudPush for IoT environment with multi-user support and discusses on the vulnerabilities of the framework and their countermeasures. The aggregated throughput of CloudPush framework is 12-15% better than IoTivity framework for a single client application working as a notification message receiver. This framework works even better 12-25% if events are distributed among different number of client application devices. Cloud server and cloud push service layer are separated that can scale up the system depending on number of events requested by IoT devices and number of client application devices involved with the events. Moreover, the cloud runs optimized searching algorithm on cloud database records. Notification message is encrypted and tracking current client account id in cloud for a secured multi-user support but it makes existing CloudPush framework with raw message 3-5% slower but performs 9-12% better than IoTivity framework for a single client. If events are distributed among multiple clients than performance improves about 9-23%. Finally, the CloudPush framework described in this thesis is significant in the field of IoT ecosystem as it supports secured multi-user feature. We have published our research work here [Published1].

8.1 Future Work

We will solve the multi-agent collaboration system with multi-user support in an energy efficient way. A multi-agent collaboration system comprises of various interacting agents that provide energy efficient solution in the system. We will focus on task allocation mechanism to IoT agents where each agent completes the task for a region in an energy efficient way on a 2D surface. Task regions in the system will be distributed among multiple agents considering their capabilities, capacities and distances from the assigned region determined by a cloud server or a local client machine. A task distribution algorithm will be proposed that distributes task region among multiple IoT agents in polynomial time. After that, each agent will perform individually to complete its task which is similar to the Traveling Salesman problem. Finding the optimal solution of the problem is NP hard. We will provide a heuristic solution for the problem.

List of publications

[Published1] M. S. A. Mozumder and M. A. Adnan, “Cloudpush: Smart delivery of push notification to secure multi-user support for iot devices,” in *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 11–19, IEEE, 2020.

References

[1] "Iotivity, linux foundation collaborative projects." [Online]. Available: https://www.iotivity.org

[2] "Iotivity open source project announces preview release," 2015. [Online]. Available: https://www.linuxfoundation.org/press-release/2015/01/iotivity-open-source-project-announces-preview-release/

[3] "Google cloud platform: Google cloud computing, hosting services & apis." [Online]. Available: https://cloud.google.com/

[4] "Amazon iot platform." [Online]. Available: https://aws.amazon.com/iot/

[5] "Azure iot solution accelerators." [Online]. Available: https://azure.microsoft.com/en-us/features/iot-accelerators/

[6] "Watson iot platform." [Online]. Available: https://www.ibm.com/internet-of-things/solutions/iot-platform/watson-iot-platform

[7] S. M. Darroudi, C. Gomez, and J. Crowcroft, "Bluetooth low energy mesh networks: A standards perspective," *IEEE Communications Magazine*, vol. 58, no. 4, pp. 95–101, 2020.

[8] Á. Hernández-Solana, A. Valdovino Bardaji, D. Perez-Diaz-De-Cerio, M. Garcia-Lozano, and J. L. Valenzuela, "Bluetooth mesh analysis, issues, and challenges," Tech. Rep., 2020.

[9] C. Bormann, A. P. Castellani, and Z. Shelby, "Coap: An application protocol for billions of tiny internet nodes," *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, 2012.

[10] M. Kovatsch, S. Mayer, and B. Ostermaier, "Moving application logic from the firmware to the cloud: Towards the thin server architecture for the internet of things," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*. IEEE, 2012, pp. 751–756.

[11] "Firebase cloud messaging client app on android." [Online]. Available: https://firebase.google.com/docs/cloud-messaging/android/client

[12] "Firebase cloud messaging server on android." [Online]. Available: https://firebase.google.com/docs/cloud-messaging/server

[13] "Iotivity github." [Online]. Available: https://github.com/iotivity/iotivity

[14] "Iotivity notification framework source code." [Online]. Available: https://github.com/iotivity/iotivity/tree/master/service/notification

[15] Y. S. Yilmaz, B. I. Aydin, and M. Demirbas, "Google cloud messaging (gcm): An evaluation," in *Global Communications Conference (GLOBECOM), 2014 IEEE*. IEEE, 2014, pp. 2807–2812.

[16] T.-B. Dang, M.-H. Tran, D.-T. Le, and H. Choo, "On evaluating iotivity cloud platform," in *International Conference on Computational Science and Its Applications*. Springer, 2017, pp. 137–147.

[17] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han, "Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 978–989.

[18] M. Ahmadi, B. Biggio, S. Arzt, D. Ariu, and G. Giacinto, "Detecting misuse of google cloud messaging in android badware," in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2016, pp. 103–112.

[19] S. W. Kum, J. Moon, T. Lim, and J. I. Park, "A novel design of iot cloud delegate framework to harmonize cloud-scale iot services," in *Consumer Electronics (ICCE), 2015 IEEE International Conference on*. IEEE, 2015, pp. 247–248.

[20] F. Li, M. Vogler, M. Claeßens, and S. Dustdar, "Towards automated iot application deployment by a cloud-based approach," in *Service-Oriented Computing and Applications (SOCA), 2013 IEEE 6th International Conference on*. IEEE, 2013, pp. 61–68.

[21] C. W. Zhao, J. Jegatheesan, and S. C. Loon, "Exploring iot application using raspberry pi," *International Journal of Computer Networks and Applications*, vol. 2, no. 1, pp. 27–34, 2015.

[22] S. D. T. Kelly, N. K. Suryadevara, and S. C. Mukhopadhyay, "Towards the implementation of iot for environmental condition monitoring in homes," *IEEE Sensors Journal*, vol. 13, no. 10, pp. 3846–3853, 2013.

[23] S. K. Datta, A. Gyrard, C. Bonnet, and K. Boudaoud, "onem2m architecture based user centric iot application development," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE, 2015, pp. 100–107.

[24] C. Doukas and I. 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, pp. 922–926.

[25] G. C. Fox, S. Kamburugamuve, and R. D. Hartman, "Architecture and measured characteristics of a cloud based internet of things," in *Collaboration Technologies and Systems (CTS), 2012 International Conference on*. IEEE, 2012, pp. 6–12.

[26] F. Tao, Y. Cheng, L. Da Xu, L. Zhang, and B. H. Li, "Cciot-cmfg: cloud computing and internet of things-based cloud manufacturing service system," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1435–1442, 2014.

[27] A. Botta, W. De Donato, V. Persico, and A. 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, pp. 23–30.

[28] "Security issues in iot: Challenges and countermeasures." [Online]. Available: https://www.isaca.org/resources/isaca-journal/issues/2019/volume-1/security-issues-in-iot-challenges-and-countermeasures

[29] M. G. Samaila, M. Neto, D. A. Fernandes, M. M. Freire, and P. R. Inácio, "Security challenges of the internet of things," in *Beyond the Internet of Things*. Springer, 2017, pp. 53–82.

[30] A.-R. Sadeghi, C. Wachsmann, and M. Waidner, "Security and privacy challenges in industrial internet of things," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 2015, pp. 1–6.

[31] Q. Jing, A. V. Vasilakos, J. Wan, J. Lu, and D. Qiu, "Security of the internet of things: Perspectives and challenges," *Wireless Networks*, vol. 20, no. 8, pp. 2481–2501, 2014.

[32] M. R. Schurgot, D. A. Shinberg, and L. G. Greenwald, "Experiments with security and privacy in iot networks," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2015 IEEE 16th International Symposium on a*. IEEE, 2015, pp. 1–6.

[33] "The fundamental of iot security." [Online]. Available: https://www.engineersgarage.com/articles/fundamental-iot-security

[34] J. Park, H. Kwon, and N. Kang, "Iot–cloud collaboration to establish a secure connection for lightweight devices," *Wireless Networks*, vol. 23, no. 3, pp. 681–692, 2017.

[35] "Ble many to many topology." [Online]. Available: https://www.novelbits.io/bluetooth-mesh-tutorial-part-1/

[36] "Ocf specification." [Online]. Available: https://openconnectivity.org/developer/specifications

[37] S. Brunozzi, "Big data and nosql with amazon dynamodb," in *Proceedings of the 2012 workshop on Management of big data systems*. ACM, 2012, pp. 41–42.

[38] "deep-dive-amazon-dynamodb." [Online]. Available: https://www.slideshare.net/AmazonWebServices/deep-dive-amazon-dynamodb

[39] "Amazon dynamodb." [Online]. Available: https://aws.amazon.com/dynamodb/

[40] "Cryptography- password-based encryption ciphers." [Online]. Available: https://developer.android.com/guide/topics/security/cryptography

[41] "Android telephonymanager." [Online]. Available: https://developer.android.com/reference/android/telephony/TelephonyManager.html

Appendix A

Source Code

A.1 Push notification message receiver and refreshed token generation

Sample codes (Java) for refreshed token and push message receiving APIs are given below…

```java
import com.google.firebase.messaging.FirebaseMessagingService;
import com.google.firebase.messaging.RemoteMessage;
import com.google.firebase.iid.FirebaseInstanceId;
import com.google.firebase.iid.FirebaseInstanceIdService;

public class MyFirebaseMessagingService extends FirebaseMessagingService {
    private static final String TAG = "MyFirebaseMsgService";

    @Override
    public void onMessageReceived(RemoteMessage remoteMessage) {
        Log.d(TAG, "FROM:" + remoteMessage.getFrom());
        //Check if the message contains data
        if(remoteMessage.getData().size() > 0) {
            Log.d(TAG, "Message␣data:␣" + remoteMessage.getData());
        }
        //Check if the message contains notification
        if(remoteMessage.getNotification() != null) {
            Log.d(TAG, "Mesage␣body:" + remoteMessage.getNotification().getBody());
            sendNotification(remoteMessage.getNotification().getBody());
        }
    }

    private void sendNotification(String body) {
        Intent intent = new Intent(this, MainActivity.class);
        intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
        PendingIntent pendingIntent = PendingIntent.getActivity(this,
            ↪ requestCode, intent, PendingIntent.FLAG_ONE_SHOT);
        Uri notificationSound = RingtoneManager.getDefaultUri(
            ↪ RingtoneManager.TYPE_NOTIFICATION);

        NotificationCompat.Builder notifiBuilder = new
            ↪ NotificationCompat.Builder(this)
                .setSmallIcon(R.mipmap.ic_launcher)
                .setContentTitle("Firebase␣Cloud␣Messaging")
                .setContentText(body)
                .setAutoCancel(true)
                .setSound(notificationSound)
                .setContentIntent(pendingIntent);
        NotificationManager notificationManager = (NotificationManager)
            ↪ getSystemService(Context.NOTIFICATION_SERVICE);
        notificationManager.notify(0 /*ID of notification*/,
            ↪ notifiBuilder.build());
    }
}

public class MyFirebaseInstanceIDService extends
    ↪ FirebaseInstanceIdService {

    private static final String TAG = "MyFirebaseInsIDService";

    @Override
    public void onTokenRefresh() {
        //Get updated token
        String refreshedToken = FirebaseInstanceId.getInstance().
            ↪ getToken();
        Log.d(TAG, "New␣Token:␣" + refreshedToken);
    }
}
```

In client app, we implement an android project to receive a push notification. Push notification message attributes are converted into JSON string. Then the JSON string is passed and processed in the communication channel as below:

```java
private Collection<String> multicast;
private HashMap<String, Object> requestAttributes;
private HashMap<String, Object> notificationAttributes;
public String toJSON(){
      JSONObject obj = new JSONObject(); // Parent object
      // create and add every notification attribute into its own
         ↪ json objects
      JSONObject not = new JSONObject();
      not.putAll(notificationAttributes);
      // add notification object to parent
      obj.put("notification", not);
      // add request attributes to parent
      obj.putAll(requestAttributes);
      if(!multicast.isEmpty()){
            // create and add all targets to the JSON array
            JSONArray ids = new JSONArray();
            ids.addAll(multicast);
            // add targets to parent
            obj.put("registration_ids", ids);
      }
      return obj.toString();
}
```

A.2 Virtual IoT Device

We implement virtual IoT device to generate an event that is sent to cloud and the cloud requests the message to the corresponding client application device through FCM server. Virtual IoT device is nothing but an Android application that trigger the lambda function of the Amazon cloud if an event is occurred. Then Amazon cloud has the communication channel with FCM. When Amazon cloud communicates with corresponding client application device id with FCM then it sends the push message to the client device. Below we are showing the code of virtual IoT device written in Java:

```java
import com.amazonaws.auth.CognitoCachingCredentialsProvider;
import com.amazonaws.mobileconnectors.lambdainvoker.
   ↪ LambdaFunctionException;
import com.amazonaws.mobileconnectors.lambdainvoker.
   ↪ LambdaInvokerFactory;
import com.amazonaws.regions.Regions;

import static android.content.ContentValues.TAG;

public class VirtualIoTDevice extends Activity {

   String COGNITO_IDENTITY_POOL = "us-west-2:da60c3fb-957c-435a-xxxx
      ↪ -4768e0fxxxxx";
   private Button mSendEventButton;

      public interface MyInterface {
            //Invoke lambda function "echo". The function name is the
               ↪  method name
            @LambdaFunction
            String FirebaseServer(Info info);
      }

      class Info {
            private String firstName;
            private String lastName;

            public Info(String firstName, String lastName) {
                  this.firstName = firstName;
                  this.lastName = lastName;
            }
      }

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      mSendEventButton = (Button) findViewById(R.id.send_event_button
         ↪ );

      mSendEventButton.setOnClickListener(new View.OnClickListener()
         ↪ {
         @Override
         public void onClick(View v) {
            init();
         }
      });}

   private void init() {
      CognitoCachingCredentialsProvider credentialsProvider = new
         ↪ CognitoCachingCredentialsProvider(
            getApplicationContext(), /* get the context for the
               ↪ application */
            COGNITO_IDENTITY_POOL, /* Identity Pool ID */
            Regions.US_WEST_2 /* Region for your identity pool--
               ↪ US_EAST_1 or EU_WEST_1*/
      );
      LambdaInvokerFactory factory = new LambdaInvokerFactory(
            getApplicationContext(),
            Regions.US_WEST_2,
            credentialsProvider);

      final MyInterface myInterface = factory.build(MyInterface.class
         ↪ );
      Info info = new Info("VirtualIoTDevice", "RVC");
            // The Lambda function invocation results in a network
               ↪ call
      new AsyncTask<Info, Void, String>() {
         @Override
         protected String doInBackground(Info... params) {
            try {
               return myInterface.FirebaseServer(params[0]);
            } catch (LambdaFunctionException lfe) {
               Log.e(TAG, "Failed␣to␣invoke␣echo", lfe);
               return null;
            }
         }
         @Override
         protected void onPostExecute(String result) {
            if (result == null) {
               return;
            }
            Toast.makeText(MainActivity.this, result, Toast.
               ↪ LENGTH_LONG).show();
         }
      }.execute(info);
   }
}
```

A.3 Local communication between the client app and IoT device

In an easy setup process, the IoT device first connects to the local network with the client device. Client-server communication protocol is established in between them. As both the client application and the IoT device perform communication by sending and receiving response, both server and client socket mechanism needs to apply to them. In our case, virtual IoT device and client app both implement the below described codes:

A.3.1 Client socket code for local network communication

```
public class ClientApp {
        private Socket mSocket;
        private static final int SERVERPORT = 6000;
        private int mIPSendType = Utils.SEND_IP_NONE;
        private InetAddress mServerAddress;
        private Handler mHandler;
        public ClientApp(InetAddress serverAddress, int sendType) {
                mServerAddress = serverAddress;
                mIPSendType = sendType;
                init();
        }
        private void init() {
                new Thread(new ClientThread()).start();
        }
        public synchronized void sendData(String str) {
                try {
                        PrintWriter out = new PrintWriter(new
                            ↪ BufferedWriter(new OutputStreamWriter(mSocket.
                            ↪ getOutputStream())),
                                        true);
                        out.println(str);
                } catch (UnknownHostException e) {
                        e.printStackTrace();
                        sendMsg("socket␣sendData␣1␣at␣Client.java␣:␣" + e.
                            ↪ getMessage() + "␣␣mSocket:␣" + mSocket);
                } catch (IOException e) {
                        e.printStackTrace();
                        sendMsg("socket␣sendData␣2␣at␣Client.java␣:␣" + e.
                            ↪ getMessage() + "␣␣mSocket:␣" + mSocket);
```

```
            } catch (Exception e) {
                  e.printStackTrace();
                  sendMsg("socket␣sendData␣3␣at␣Client.java␣:␣" + e.
                     ↪ getMessage() + "␣␣mSocket:␣" + mSocket);
            }
      }
class ClientThread implements Runnable {
      @Override
      public void run() {
            try {
                  mSocket = new Socket(mServerAddress, SERVERPORT);
                  if (mIPSendType == Utils.SEND_IP_CLIENT_To_SERVER)
                     ↪ {
                        sendData(Utils.
                           ↪ SEND_CLIENT_TO_SERVER_NODE_JOINED_REQ +
                           ↪  Utils.SEPERATOR + Utils.
                           ↪ DEVICE_MAC_ADDRESS
                                    + Utils.SEPERATOR + Utils.
                                       ↪ getIpAddress().
                                       ↪ getHostAddress());
                  } else if (mIPSendType == Utils.
                     ↪ SEND_IP_CLIENT_To_CLIENT) {
                        sendData(Utils.
                           ↪ SEND_CLIENT_To_CLIENT_NODE_JOINED_REQ +
                           ↪  Utils.SEPERATOR + Utils.
                           ↪ DEVICE_MAC_ADDRESS
                                    + Utils.SEPERATOR + Utils.
                                       ↪ getIpAddress().
                                       ↪ getHostAddress());
                  }
            } catch (IOException e) {
                  e.printStackTrace();
                  sendMsg("socket␣creation␣at␣Client.java␣:␣" + e.
                     ↪ getMessage() + "␣␣mSocket:␣" + mSocket);
            }
       }
}
      public void stop() {
            try {
                  if (null != mSocket) {
                        if (!mSocket.isClosed()) {
                              mSocket.close();
                        }
                        mSocket = null;
                  }
                  mServerAddress = null;
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
      public Socket getSocket() {
            return mSocket;
      }
      public void sethandler(Handler handler) {
            mHandler = handler;
      }
      public InetAddress getInetAddress(){
            return mServerAddress;
      }
      private void sendMsg(String str) {
            Message msgObj = mHandler.obtainMessage();
            Bundle b = new Bundle();
            b.putString("message", str);
            msgObj.setData(b);

            mHandler.sendMessage(msgObj);
      }
}
```

A.3.2 Server socket code for local network communication

```
public class LocalServer {
      private ServerSocket mServerSocket;
      Handler updateConversationHandler;
      Thread serverThread = null;
      private ArrayList<Thread> mCommunicationThreadList;
      public static final int SERVERPORT = 6000;

      public LocalServer(Handler handler) {
            updateConversationHandler = handler;
            mCommunicationThreadList = new ArrayList<Thread>();
            this.serverThread = new Thread(new ServerThread());
            this.serverThread.start();
      }
      public void stop() {
            try {
            mServerSocket.close();
            serverThread.interrupt();
            for (int i = 0; i < mCommunicationThreadList.size(); i++)
               ↪ {
                  Thread ct = mCommunicationThreadList.get(i);
                  ct.interrupt();
            }
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
class ServerThread implements Runnable {
      public void run() {
      Socket socket = null;
      try {
            mServerSocket = new ServerSocket(SERVERPORT);
      } catch (IOException e) {
            e.printStackTrace();
            sendMsg("mServerSocket␣creation␣␣at␣Server.java␣:␣" + e.
                  ↪ getMessage());
      }
      while (!Thread.currentThread().isInterrupted()) {
      try {
            socket = mServerSocket.accept();
            CommunicationThread commThread = new CommunicationThread(
                  ↪ socket);
            Thread ct = new Thread(commThread);
            mCommunicationThreadList.add(ct);
            ct.start();
      } catch (IOException e) {
            e.printStackTrace();
            sendMsg("mServerSocket␣accept␣␣at␣Server.java␣:␣" + e.
                  ↪ getMessage());
      }
      }
      }
```

```
}
class CommunicationThread implements Runnable {
    private Socket clientSocket;
    private BufferedReader input;
    public CommunicationThread(Socket clientSocket) {
        this.clientSocket = clientSocket;
        try {
        this.input = new BufferedReader(new InputStreamReader(
           ↪ this.clientSocket.getInputStream()));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void run() {
    while (!Thread.currentThread().isInterrupted()) {
    try {
        String read = input.readLine();
        if (null == read) {
            sendMsg(Utils.CLIENT_REMOVE_REQ+ Utils.SEPERATOR +
               ↪ clientSocket.getInetAddress().getHostAddress
               ↪ ());
            if (null != clientSocket) {
                clientSocket.close();
            }
            Thread.currentThread().interrupt();
            return;
        }
        Message msgObj = updateConversationHandler.obtainMessage
           ↪ ();
        Bundle b = new Bundle();
        b.putString("message", read);
        msgObj.setData(b);
        updateConversationHandler.sendMessage(msgObj);
        updateConversationHandler.post(new updateUIThread(read));
    } catch (IOException e) {
        e.printStackTrace();
        sendMsg("recever␣exception␣␣at␣Server.java␣:␣" + e.
           ↪ getMessage());
    }
    }
    }
```

```java
}
    private void sendMsg(String str) {
        Message msgObj = updateConversationHandler.obtainMessage();
        Bundle b = new Bundle();
        b.putString("message", str);
        msgObj.setData(b);
        updateConversationHandler.sendMessage(msgObj);
    }
}
```

A.3.3 Retrieve local network IP list of the connected devices

The IoT device can't connect to the client app because it does not know the IP address. Client app performs a searching operation with it's connected devices and retrieve the IP address of the IoT device. IP address retrieval codes are shown below:

```java
public void getLocalNetworkIPList(final int reachableTimeout) {
Runnable runnable = new Runnable() {
    public void run() {
        BufferedReader br = null;
        final ArrayList<ClientDataResult> result = new ArrayList<ClientDataResult>();
        try {
            br = new BufferedReader(new FileReader("/proc/net/arp"));
            String line;
            while ((line = br.readLine()) != null) {
                String[] splitted = line.split("␣+");
                if ((splitted != null) && (splitted.length >= 4)) {
                    String mac = splitted[3];
                    if (mac.matches("..:..:..:..:..:..")) {
                        boolean isReachable = InetAddress.getByName(splitted[0]).isReachable(reachableTimeout);
                        result.add(new ClientDataResult(splitted[0], splitted[3], splitted[5], isReachable));
                    }
```

```
                        }
                    }
                } catch (Exception e) {
                    Log.e(this.getClass().toString(), e.toString());
                    Log.e("Arifin", "getClientList()␣in␣WifiClient.java
                       ↪ ,␣exception:␣" + e.toString());
                } finally {
                    try {
                        br.close();
                    } catch (IOException e) {
                        Log.e(this.getClass().toString(), e.
                           ↪ getMessage());
                    }
                }

                Handler mainHandler = new Handler(mContext.getMainLooper
                   ↪ ());
                Runnable myRunnable = new Runnable() {
                    @Override
                    public void run() {
                        for (int i = 0; i < result.size(); i++) {
                            if (result.get(i).isReachable() && (
                               ↪ result.get(i).getDevice().
                               ↪ contains("wlan0")
                                       || result.get(i).getDevice
                                          ↪ ().contains("eth0")
                                       || result.get(i).getDevice
                                          ↪ ().contains("ap0")))
                                          ↪ {
                                mServerClientListener.
                                   ↪ onSendAddress(result.get(i)
                                   ↪ );
                            } else if(result.get(i).getIpAddr().
                               ↪ equals(Utils.HOTSPOT_IP) && !
                               ↪ result.get(i).isReachable()){
                                mServerClientListener.onSendMSG("
                                   ↪ Wifi␣Hotspot␣unreachable␣
                                   ↪ wifiClient.java");
                            }
                        }
                    }
```

```
            };
            mainHandler.post(myRunnable);
      }
};
      Thread mythread = new Thread(runnable);
      mythread.start();
}
class ClientDataResult {
      private String mIpAddr;
      private String mHWAddr;
      private String Device;
      private boolean isReachable;
      public ClientDataResult(String ipAddr, String hWAddr, String
         ↪ device, boolean isReachable) {
            super();
            this.mIpAddr = ipAddr;
            this.mHWAddr = hWAddr;
            this.Device = device;
            this.isReachable = isReachable;
      }
}
```

A.4 Account authorization

OAuth 2.0 client code is written in Java in the CloudPush framework. The Client app sends account id, password and account type to Amazon cloud. Then Amazon cloud verifies the account information provided by client app with proper account authorization server. Amazon cloud verifies whether provided account information is valid or not. Account authorization server can be Google, Yahoo, Samsung servers etc. Amazon cloud implements only the business layer of account authorization. It returns to client success if provided account is valid or failure otherwise.

Amazon server request the authentication server for access token. Access token is a long string of characters that serves as a credential used to access protected resources. Account registration is successful if authorization server response with a token. This process runs every time, client app login into amazon cloud. Access token retrieval code is shown below:

```
HttpPost post = new HttpPost(oauthDetails.getTokenEndpointUrl());
String clientId = oauthDetails.getClientId();
String clientSecret = oauthDetails.getClientSecret();
String scope = oauthDetails.getScope();
Map<String, String> map = new HashMap<String, String>();

List<BasicNameValuePair> parametersBody = new ArrayList<
   ↪ BasicNameValuePair>();

parametersBody.add(new BasicNameValuePair(OAuthConstants.GRANT_TYPE,
   ↪ oauthDetails.getGrantType()));
parametersBody.add(new BasicNameValuePair(OAuthConstants.CODE,
   ↪ authorizationCode));
parametersBody.add(new BasicNameValuePair(OAuthConstants.CLIENT_ID,
   ↪ clientId));
if (isValid(clientSecret)) {
      parametersBody.add(new BasicNameValuePair(OAuthConstants.
         ↪ CLIENT_SECRET, clientSecret));
}
parametersBody.add(new BasicNameValuePair(OAuthConstants.REDIRECT_URI
   ↪ , oauthDetails.getRedirectURI()));
DefaultHttpClient client = new DefaultHttpClient();
HttpResponse response = null;
String accessToken = null;
try {
      post.setEntity(new UrlEncodedFormEntity(parametersBody, HTTP.
         ↪ UTF_8));
      response = client.execute(post);
      int code = response.getStatusLine().getStatusCode();
      map = handleResponse(response);
      accessToken = map.get(OAuthConstants.ACCESS_TOKEN);
} catch (ClientProtocolException e) {
      e.printStackTrace();
      throw new RuntimeException(e.getMessage());
} catch (IOException e) {
      e.printStackTrace();
      throw new RuntimeException(e.getMessage());
}
return map;
```

A.4.1 Amazon cloud DynamoDB operations

Our implemented AWS DynamoDB Java APIs are shown below:

```
AmazonDynamoDBClient ddb = DynamoDBController.ddb();
DynamoDBMapper mapper = new DynamoDBMapper(ddb);
try {
   UserPreference userPreference = new UserPreference();
   userPreference.setDeviceID(deviceID);
   userPreference.setAccountID(accountID);
   userPreference.setIoTDevID(ioTDevID);
   userPreference.setToken(token);
   userPreference.setServerkey(serverkey);
   ...............................
   // insert/update row in cloud table
   mapper.save(userPreference);
   // delete row in cloud table
   mapper.delete(deleteUserPreference);
   // scan relevant data in cloud table
   PaginatedScanList<UserPreference> result = mapper.scan(
      ↪ UserPreference.class, scanExpression);
} catch (AmazonServiceException ex) {
   Log.e(TAG, "Error");
}
```

A.5 Lambda Function in Amazon Cloud Server

We have used AWS Lambda function that works on demand where FCM server logic is implemented in our system. A Lambda instance executes within milliseconds of an event. When an IoT device generates an event, it immediately invokes lambda function to let Amazon cloud know. Then Amazon cloud searches corresponding IoT device id in the cloud table. Following code is showing how lambda function has been implemented and triggering http post request to FCM server:

```
public void pushToFCM(Notification n) { //Lambda Function
      HttpsURLConnection con = null;
      try{
            String url = "https://fcm.googleapis.com/fcm/send";
            URL obj = new URL(url);
            con = (HttpsURLConnection) obj.openConnection();
            con.setRequestMethod("POST");

```

```java
                // Set POST headers
                con.setRequestProperty("Authorization", "key="+ FIREBASE_SERVER_KEY);
                con.setRequestProperty("Content-Type", "application/json; charset=UTF-8");

                // Send POST body
                con.setDoOutput(true);
                DataOutputStream wr = new DataOutputStream(con.getOutputStream());
                BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(wr, "UTF-8"));
                writer.write(n.toJSON());

                // close output stream
                writer.close();
                wr.close();

                // send request
                con.connect();
        }
        catch(Exception e){
                e.printStackTrace();
        }
}
```