Managing Data Consistency in IoT Devices

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Chih-Yi Huang

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying, publication, or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head
Department of Computer Science
176 Thorvaldson Building
110 Science Place
Saskatoon SK S7N 5C9
Canada

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon SK S7N 5C9
Canada

ABSTRACT

Since a system can only make a decision based on the information it has at any given time, when the information changes, the decision needs to change as well. For example, a smartphone user decides in the morning to go for a walk later in the afternoon since the weather is sunny and the temperature is above 15 degrees Celsius. The user then records the information and the decision in his smartphone. Here, the user bases their decision on two sources of information, weather condition and temperature, which determine their choice of whether to go for a walk or not. Later, the forecasted temperature changes to 8 degrees Celsius and the decision the user made early in the morning, to go for a walk in the afternoon, may need to change as well. There should be a way for the smartphone to be aware of the temperature change and re-evaluate the decision. If the decision needs to change, the smartphone needs to inform the user that a decision made previously is now invalid, and a new decision has been made.

This research presents an architecture called Local Resources' State Management System (LRSMS) to keep track of the data used in a device or a machine as well as dependencies between data. When the device recognizes a state change in any of its data, the LRSMS will tell the device to propagate the state change to all dependent data and inform the use of the changes.

The experimental results in this research demonstrate that the data propagation works correctly in the LRSMS; however, the average data update speed depends on the data dependency structure. In some data dependency structures, the average data update speed is relatively constant, while in other structures the average data update speed increases relative to the quantity of data.

ACKNOWLEDGEMENTS

Here, I would like to express my most sincere appreciation to my supervisor Professor Ralph Deters. Under his supervision, I not only obtained great skill and wide experience in computer science during my masters, but also achieve marks and increase skill in programming languages.

I would also like to thank everyone in the committee, Professor Gord McCalla, Professor Julita Vassileva and Professor Li Chen, for giving me useful suggestions to make my thesis more robust.

At last, I like to thank the encouragement and backing from my family. Without their help, I can not finish this master myself.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The growth of Internet of Things (IoT) devices facilitates the collection of sensor data in different places and allows more things to be remotely controlled through the Internet. On the other hand, the growth of IoT device usage increases the burden placed on a centralized IoT system. A centralized system will eventually be overwhelmed by the quantity of IoT devices, and this will reduce the system's overall performance. One way to address this is to change the system from a centralized system to a decentralized system, thereby turning one central machine into multiple central machines. Decentralized systems increase both the number of IoT devices the system can handle and the automaticity of the system. Each individual central machine can act individually without communication with other central machines and make its own decisions based on data stored in the machine itself. These decisions, which are also data, form a dependency relationship with other data stored in the machine. With the increased quantity of data stored in each central machine, the relationship between the data becomes more and more complex. Furthermore, connection between central machine and the devices could be unstable and result in one of the central machines making invalid decisions due to outdated data.

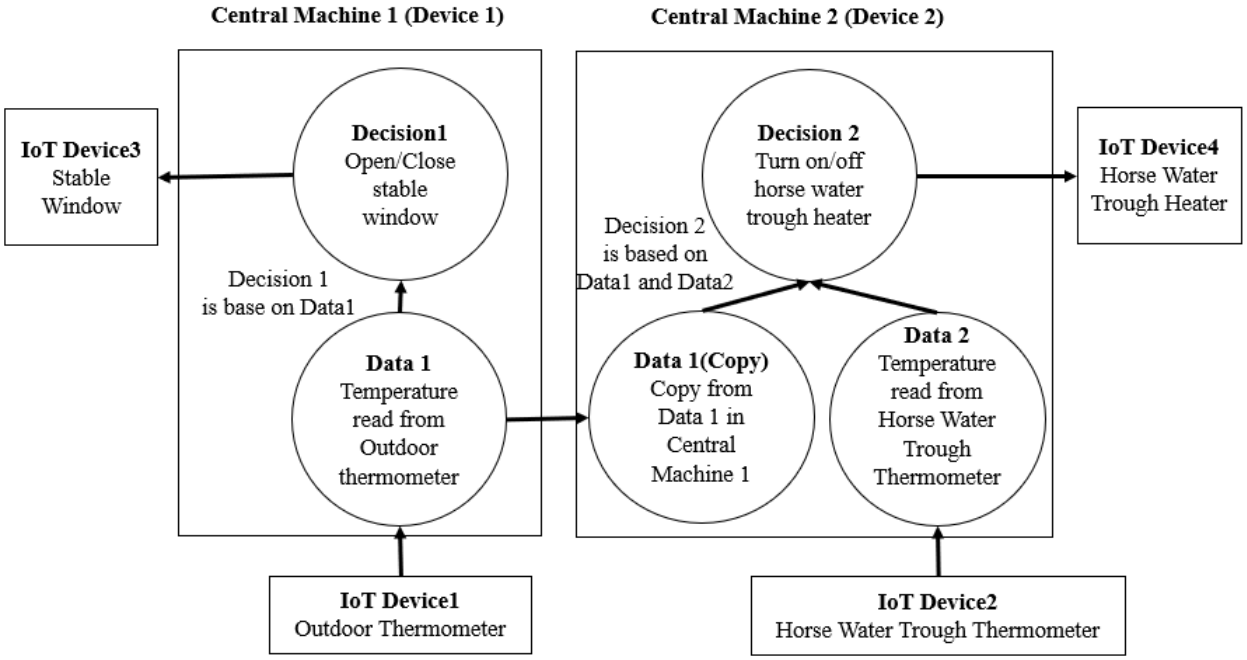


Figure 1.1 System Example

This thesis presents a system to manage data consistency and validation as well as the relationship between data in an intermittent connection environment. Figure 1.1 shows an example of how the system works. Central Machine 1 (Device 1) receives the data, outdoor temperature, from IoT Device 1, and then stores that data in Data 1. Then, Central Machine 1 uses Data 1to decide if the horse stable window needs to be opened or closed. Then Central Machine 1 signals to IoT Device 3, the stable window, to open or close. Similarly, Central Machine 2 (Device 2) receives data, the horse water trough thermometer reading, from IoT Device 2 and then stores the data in Data 2. Then, Central Machine 2 uses a copy of Data 1 (which was received from Central Machine 1) and Data 2 in order to decide if the horse water trough heater needs to be turned on or off. Based on the decision, Central Machine 2 signals IoT Device 4, the horse water trough heater, to turn on or off. A storm occurs that results in a new reading for both IoT Device 1 and IoT Device 2 and disables the connection between Central Machine 1 and Central Machine 2. Central Machine 2 then acts as a standalone machine and will make a new decision (D1) based on what it has in Data 1 and New Data 2. This new decision (D1) is based on outdated Data 1 and could be an invalid decision. A system that keeps track of these potential instances of invalid data would be very useful

when both Central Machine 1 and Central Machine 2 are reconnected. This system would indicate to Central Machine 2 that D1 may be invalid and therefore it needs to be re-evaluated.

Chapter 2 describes questions that need to be solved by the system. Chapter 3 introduces research that aligns with the Chapter 2 questions either in response or with regards to systemic operations. Chapter 4 explains the system in detail. Chapter 5 explains the results of experiments demonstrating how the system solves the previously identified problems, as well as testing of the system performance. Lastly, Chapter 6 provides a summary and discusses potential future work considerations.

CHAPTER 2

PROBLEM DEFINITION

In an environment with intermittent connectivity, a system needs to make decisions based on the information it has at any given time; however, the information a system uses can change as soon as a connection is re-established. However, the information a system uses to make those decisions can change as soon as it re-establishes the connection. Therefore, decisions made before the reconnection need to be re-evaluated after reconnection. The goal of this research is to present a system to manage the dependencies between resources (data, information, decisions), as well as alter the states of dependent resources when a new state is discovered in a primary resource. This system is assumed to be running in the background as a service in a machine or a device. To achieve the goal of this research, the following questions are addressed:

- How can the dependencies between resources be tracked?
- How can the system be informed when a resource state has changed?
- How can a resource state change be propagated to dependent resources?
- How can the same prime resource be identified in different systems?
- How can resources in other connected devices be informed when a primary resource they depend on has a new state?
- How can a resource be updated when not all its primary resources are reachable?

2.1 How can the dependencies between resources be tracked?

Keeping track of the dependencies between resources is the first step to make the system work. The system needs a way to know which resources are dependent upon others.

2.2 How can the system be informed when a resource state has changed?

When a resource state has changed, how does the system know about it? A simple solution is to let the system keep checking for updates; however, as the number of resources increases, the cost of the update checking becomes more and more expensive. Is there a better way to solve this?

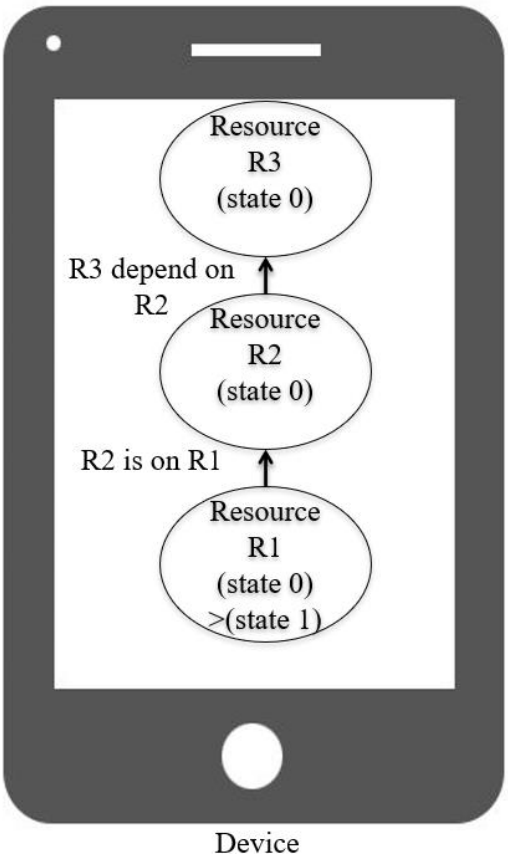


Figure 2.1 Dependent Resources

2.3 How can a resource state change be propagated to dependent resources?

After a primary resource receives a new state, how does the system tell the resources dependent on it to check their states and see if they need an update? In Figure 2.1 above, Resource R1 tells the system that its state just changed from state 0 to state 1. The system now needs to do something to let Resource R2, which is dependent on R1, know that the R1 state changed, and it might need to update its state as well. The system also needs to inform Resource R3, which is dependent on R2, that R2 is probably not up to date, so its state is probably out of date as well.

2.4 How can the same prime resource be identified in different devices?

Identifying resources is critical in this research, especially between devices. The system needs to be able to identify the same resource used in different devices so that devices are clear as to which resource they are talking about when exchanging resource information.

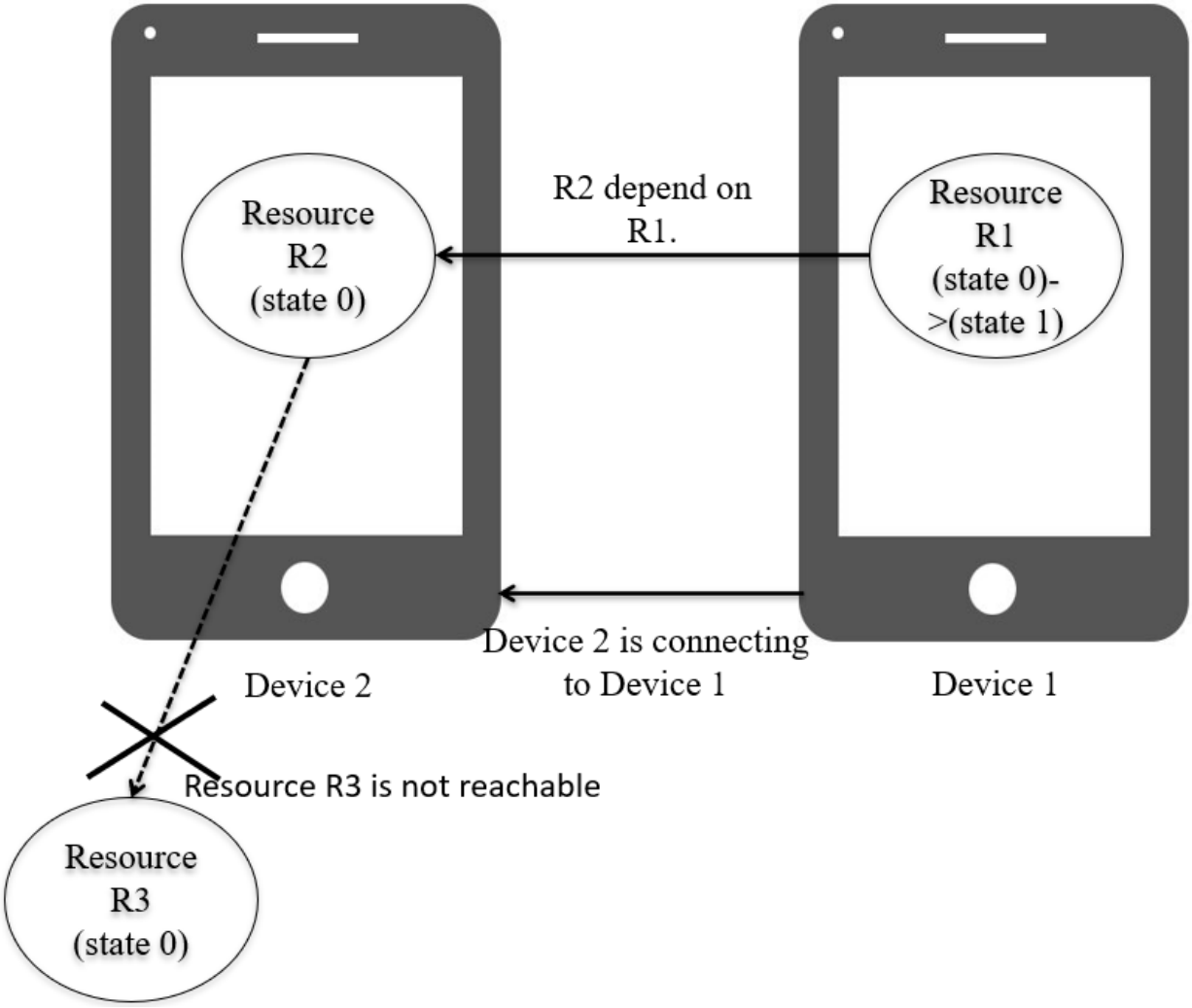


Figure 2.2. Inform Resource

2.5 How can resources in other connected devices be informed when a primary resource they depend on has a new state?

In figure 2.2 above, how to let Resource R2 in Device 2 know that Resource R1 in Device 1 which is one of the resource Resources R2 is dependent on has a new state?

2.6 How can a resource be updated when not all its primary resources are reachable?

In Figure 2.2 when the system instructs Resource R2 to check if it needs to change its state because Resource R1 in Device 1 has a new state, Resource R2 needs to access both Resource R1 and Resource R3; however, Resource R3 is not reachable for Resource R2 because there is no

connection. Therefore, Resource R2 is not able to process its new state since one of the resources it depends on, Resource R3, is not available at the time.

The six aforementioned questions are addressed by the system presented in this research. The system also aims to manage the dependencies between resources (data, information, decisions) and alter the state of dependent resources when appropriate. Experiments were conducted to evaluate the efficiency, effectiveness, and performance of the system.

CHAPTER 3

BACKGROUND REVIEW

This research touches on various areas including Context Awareness, Internet of Things (IoT), Truth Maintenance System, CAP Theorem, Representational State Transfer (REST) and Simple Object Access Protocol (SOAP) and Hypertext Transfer Protocol (HTTP) and Constrained Application Protocol (CoAP).

3.1 Context Awareness

This section discusses the concept of Context Awareness, its definition, and the features of context-aware applications.

3.1.1 Definition

Want, Hopper, Falcao, and Gibbons [1] define Context Awareness in “The active badge location system.” The term context-awareness used in computer science was first seen in 1994 by Schilit and Theimer [2] in “Context-aware computing applications.” Since 1994, many people have tried to define context-aware computing. In 1994, Schilit and Theimer [2] describe context-aware computing as computing that "adapts according to its location of use, the collection of nearby people and objects, as well as changes to those objects over time.” Pascoe, Ryan, and Morse [3], in 1998, described context awareness as a system's "capability to sense its environment." In 2000, Dey [4] defined context awareness as the characteristic by which a system "used context to provide relevant information and/or services to the use, where relevancy depends on the user's task." He further defined context as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.”

3.1.2 Features of context-aware applications

Schilit, Adams, and Want [5] categorize context-aware application based on their features.

- Proximate selection applications: Depending on the existing context, applications will retrieve information for the user manually. For example, a user asks the navigation system

on the car to show the list of nearby gas stations. In this case, the existing context is the car's current location.

- Automatic contextual reconfigurations: Opposite of proximate selection application. These kinds of applications retrieve information for the user automatically. For example, the car's navigation system shows the closest gas station on its screen all the time.
- Contextual command applications: Depending on the existing context, applications will execute commands manually. For example, a self-driving car will drive to the gas station the user picks.
- Context-triggered actions applications: These kinds of applications will execute commands automatically depending on the current context. For example, a self-driving car will drive to the closest gas station when fuel is low.

Pascoe [6] offered another way to classify context aware applications that is similar to Schilit, Adams, and Want's [5] version:

- Contextual sensing: The capability to detect contextual information and show it to the user. This is like proximate selection in Schilit, Adams, and Want's [5] version.
- Contextual adaptation: The capability to execute or modify automatically based on the current context. This is like context-triggered actions in Schilit, Adams, and Want's [5] version.
- Contextual resource discovery: The capability to detect and utilize resources. This is like automatic contextual reconfiguration in Schilit, Adams, and Want's [5] version.
- Contextual augmentation: The capability to associate data with the context. For example, putting the rating of a restaurant in the car navigation system, then next time when you or another user are near the restaurant, the rating of the restaurant is presented.

Although Pascoe [6] does not have an element to parallel contextual command, he includes contextual augmentation as an additional feature in his classification.

Dey [4] provides a way to summarize context aware application features in three ways.

- Presentation of information and services to a user: This is a combination of proximate selection and contextual command in Schilit, Adams, and Want's [5] version.

- Automatic execution of a service: Same as Context-triggered action in Schilit, Adams, and Want's [5] version.
- Tagging of context to information for retrieval: Identical contextual augmentation in Pascoe's [6] version.

Dey [4] does not include the automatic contextual reconfiguration feature nor the contextual resource discovery feature from the other versions. He perceives this as part of his first two features.

3.2 Internet of Things (IoT)

The idea of the Internet of Things, or IoT, can be traced back to the early 1990s in the paper "The Computer of the 21st Century" by Mark Weiser [7]. One of the purposes of IoT is to enable objects to interact with one another through an Internet connection to make changes on their own and thereby reduce human involvement as much as possible. Another purpose of IoT is to gather object data through the Internet, so the end user can receive that information from anywhere at any time. IoT reduces the cost to gather information and increases efficiency. Finally, IoT presents a way for end users to remotely control objects through Internet.

3.2.1 How IoT Works

According to Barrett [8]. there are specific abilities that an object wanting to join the IoT must acquire in order to become a "thing" on the Internet.

- Unique ID: In IoT world, each object needs to have a way to distinctly identify itself. A common way of doing it is to use IPv6, which use 128 bits to store an address, and can theoretically provide 2^{128} or ~$3.4*10^{38}$ addresses.

- Communication: To communicate through the Internet, the object needs to find a way to connect to the Internet. There are many ways to connect to the Internet, including Wi-Fi, Bluetooth, 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks), and Ethernet. After the connection is established, the type of communication protocols used in the connection are dependent on the methods chosen and the IoT application used to connect to Internet.

- Sensor/Reactor: Sensors need to read data from the object, and the reactor needs to respond to the signals it receives. For example, a motion detector (sensor) detects human motion then sends a signal through internet which is received by the light switch(reactor) and the light turns on.
- Remote controller: There needs to be a remote-control device that can control the object or thing through the Internet connection, such as a smartphone, a tablet, or a desktop computer.

Although not a requirement, there may be times when it is advantageous to complete the lightweight data processing locally instead of sending it to the Cloud.

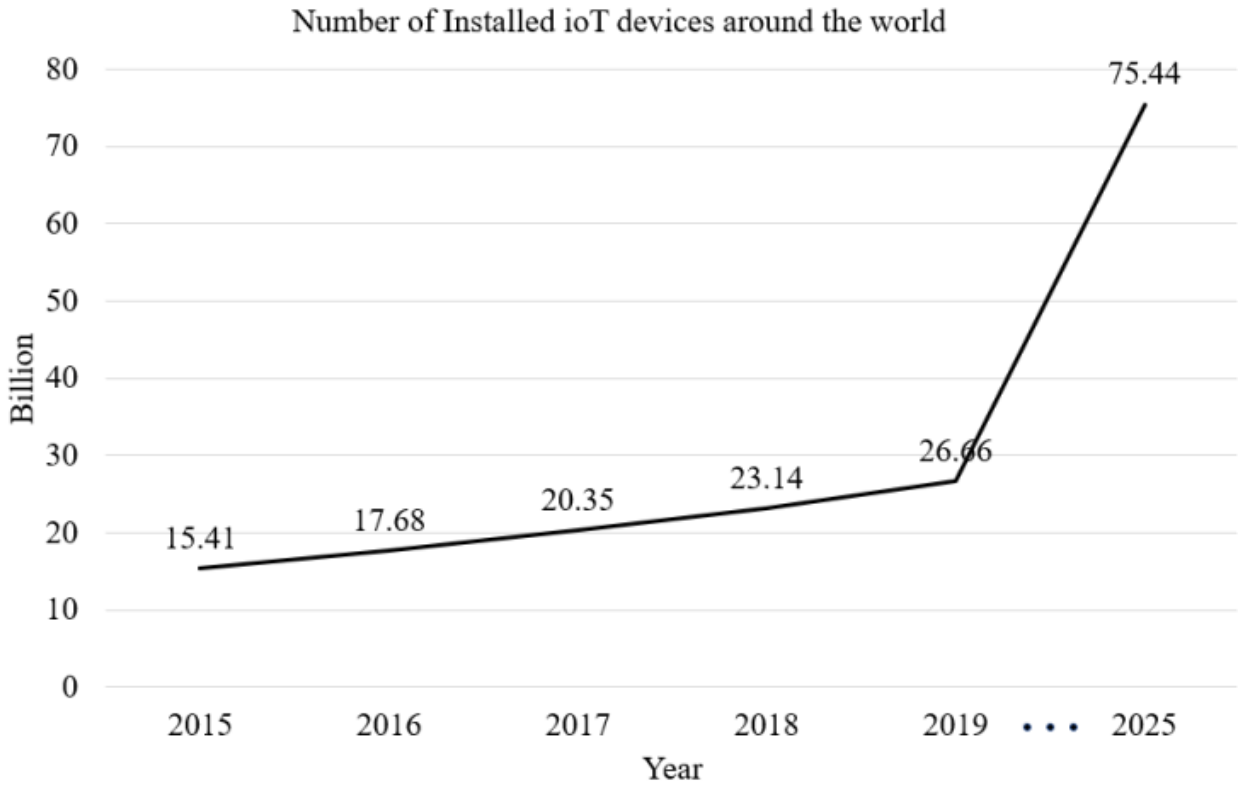


Figure 3.1 IoT devices Trends [9]

3.2.2 Trends

Figure 3.1 illustrates the growth of the number of connected IoT devices each year since 2015. The rate of growth is not as high as expected by some experts. In 2011, Evans [10] predicted that

there would be 25 billion connected IoT devices in 2015 and 50 billion devices by 2020. There were only 15.4 billion connected IoT devices by the end of 2015 and 26.6 billion connected devices by the end of 2019, both of which are significantly different from 50 billion. In 2016, Evans [11] adjusted his prediction to be 28 billion connected IoT devices in 2021. Although the rate of growth is not as fast as experts previously predicted, a positive trend can be observed and expected.

3.2.3 Problems

- Security and privacy: According to Feamster [12] "many IoT devices have long-standing, widely known software vulnerabilities that make them vulnerable to exploit and control by remote attackers." As the number of connected objects increases, the potential for an attacker to steal confidential data increases as well.
- Since there is no universal standard for IoT, it is challenging for devices to interact with other devices that run on different hardware and/or software systems.
- Project management: According to a Cisco [13] study in 2018, "60 percent of IoT initiatives stall at the Proof of Concept (PoC) stage." More than half of the IoT implementation projects still on the first stage of development which is detrimental to the promotion of the IoT.

3.3 Truth Maintenance System

Stanojevic, Vranes, and Velasevic [14] explain that truth maintenance is "an area of AI concerned with revising sets of beliefs and 'maintaining the truth in the system' when new information contradicts existing information." A Truth Maintenance System (TMS) can be a tool or an application in the computer, which adopts the idea of truth maintenance to keep the data or results in the computer application to be true and trustworthy the entire time base on the current data it has. If there are changes to any data, the system will check all the data which is related to the changed data and make the corresponding changes based on its knowledge base to make all the data, especially results in the system, true and trustworthy.

3.3.1 Purpose

A Truth Maintenance System is used to solve something that logic algorithms cannot solve, such as Constraint Satisfaction Problems (CSPs) and Scenario and Planning Problems.

- Constraint Satisfaction Problems (CSPs) are specific to a set of the variables which have constraints that limit the value that each variable can be. The problem can be solved only when all the values assigned to each variable do not violate any constraint in the constraint set.
- Scenario and Planning Problems give an initial state and final problem states along with constraints that restrict the path of travel between the initial state and final problem states. A solution of a scenario and planning problem is a path of travel from the initial state to a final state which does not violate any constraints.

The difference between Constraint Satisfaction Problems and Scenario and Planning Problems is that CSPs increase the number of variables in each step, while Scenario and Planning Problems have a set number of variables from the initial state but need to find a path to change the value of the variables to match the final state.

3.3.2 Types of TMS

The three types of Truth Maintenance Systems are Justification-based TMSs, Assumption-based TMSs, and Logic-based TMSs [14].

- A Justification-based TMS uses justification to tell which node should be derived (believed) based on current rules and constraints (which are represented as nodes that have a set value). Nodes that are set to be believed are “in” and nodes that are set to be disbelieved are “out.” For example, we might decide to go for a walk based on two factors; first, if the weather is nice, and second, if it is daytime. In this case, nice weather and daytime are “in” nodes and when those two nodes are “in”, the node “go for a walk” will be set to “in” and become true.
- An Assumption-based TMS differs from a Justification-based TMS because it can since it can have multiple contexts, whereas a Justification-based TMS can only have exactly one context. Contexts are sets of nodes that determine if the node is “in” or “out”. Take the example above. If we only know the weather is nice, but we are not sure about whether it

is daytime or not, a Justification-based TMS cannot make the node "go for a walk" be true; however, in an Assumption-based TMS, if the weather is nice and if daytime is unknown, then “go for a walk” can be set to true with an assumption that daytime is not “out”.

- A Logic-based TMS on the other hand does not use justification, but rather constraints on the label of each node. Take our previous example. The node “go for a walk” will have two constraints, daytime and weather. Only if those two constraints are satisfied, can the node “go for a walk” be set to true.

When new data is received, the truth maintenance system on the device uses this new information to check all the other data in the device to make sure the decisions and results derived from the new data are true. In the case where the system finds out a new result has been generated from the new data, the system can alert the user that something has changed, and the user can respond to the new result.

3.3.3 The Consistency of the Data

Huhns and Bridgeland [15] provide an algorithm for truth maintenance that is used to ensure the consistency of the data in each agent and the consistency of the data shared by agents. They define the data consistency state on four levels.

- Inconsistency: One or more agents has inconsistent data.
- Local Consistency: Each agent is individually consistent.
- Local-Shared Consistency: Each agent is individually consistent, and the shared data is consistent.
- Global Consistency: All data are consistent regardless of whether they are global or local. In other words, all data can be merged into one knowledge base without any state contradictions.

Huhns and Bridgeland [15] provide an algorithm in which local-shared consistency can always be determined if the following three conditions are met. First, agents are completely cooperative. Second, agents must be able to provide justifications for what they believe. Third and last, agents must have identical representation of their knowledge base. The time needed for the algorithm is $O(3^k 2^n)$, where k is the number of shared data for all agents and n is the amount of data in each agent.

If devices and Internet resources are agents, this algorithm can be used while trying to solve the problem of the inconsistency of the data between multiple devices and Internet resources, especially when there is conflict between resources. This algorithm can help us achieve a high level of local-shared consistency, which provides the system with the most reasonable data between devices and resources that can be utilized by the user to make the right decision.

3.4 CAP Theorem

CAP stands for consistency, availability, and partition tolerance. It was first referred to as the CAP Principle in 1999 [16] and introduced at the Symposium on Principles of Distributed Computing by Brewer [17]. The CAP principle was proven by Gilbert and Lynch [18] in 2002, at which point the name was changed to the CAP Theorem. The CAP Theorem is also known as Brewer's Theorem.

3.4.1 Definition

Brewer [16] shows that in a distributed environment there are three core system properties that are in a unique relationship when it comes to the design of the system. The three properties are consistency, availability, and partition tolerance. The CAP Theorem states that a system can at most have two out of three properties at any time.

- Consistency: In a distributed system, the same data (a copy) can be stored in multiple servers. When an update occurs on a server, it should be reflected on all other servers that have the same data the moment the update occurs. A system with this kind of characteristic is called a consistent system. In other words, after an update, every read operation will receive the latest data value in the system no matter which server from which it requests the value. Keep in mind that this kind of synchronization after every update is a cost and will gradually reduce the performance of the system.
- Availability: Every request received by any server in the system must return a response. Furthermore, the availability is not for a single server, it for the entire system. In other words, an available system will always respond to the user, even if some servers in the system are down.

- Partition Tolerance: In case of network failure, the system is split into multiple clusters and all communication between the two clusters is closed. A partition tolerance system should be able to withstand this kind of problem and keep functioning on both clusters.

3.4.2 Explanation

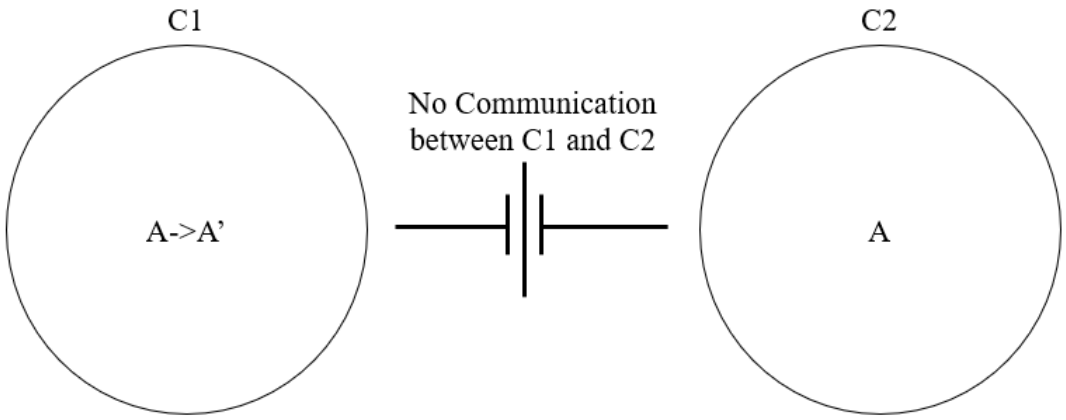


Figure 3.2 CAP Theorem Explanation

Figure 3.2 shows an explanation of why it is impossible to have all three CAP properties in a server by using the proof by contradiction method. Assume there exists a distributed system that satisfies all three properties of the CAP theorem. To satisfy partition tolerance, the system is now split into two clusters, Cluster 1 (C1) and Cluster 2 (C2). The communication between C1 and C2 is no longer available. Assume there is Data A in C1. To satisfy availability, a copy of Data A needs to be stored in C2 as well since the communication between C1 and C2 is down.

An update occurs in C1 that changes Data A to A'. Since communication is not available between C1 and C2, Data A in C2 remains as A. This shows an inconsistency in the system and contradicts our assumption, which explains why it is impossible to have a distributed system with all three CAP properties.

Although a system with all three properties is not possible, a system with two properties is achievable**.**

3.4.3 Consistency vs. Availability

In practice, no distribution system is safe from network failure, so partition tolerance is a property that needs to be included most of the time. As a result, the remaining option is to either sacrifice availability, resulting in a system with consistency and partition tolerance (CP), or to sacrifice consistency, resulting in a system with availability and partition tolerance (AP).

One of the distributed system design philosophies adopting AP is BASE, which was designed by Brewer [19] in late 1990s. BASE stands for Basically Available, Soft State, Eventually Consistent. Soft State and Eventually Consistent are two techniques used to partially compensate for the consistency property lost in the BASE system.

According to Brewer [19], another distributed system design philosophy that adopts CP is ACID. ACID is the abbreviation of Atomicity, Consistency, Isolation, and Durability.

- Atomicity: When an update is happening in a distributed system, all the operations need to be done completely or the system will be forced to roll back to its original state before the update. For example, Data A exists on 10 servers in the system. When an update happens, all 10 instances of Data A are either updated to a new state or they will all revert back to their original state.
- Consistency: Consistency in ACID is different from the consistency in CAP. The consistency in ACID means that during a system state change, the system's new state must be a valid state. This can be achieved by making sure all updates follow system rules and result in a valid system state. For example, a bank system requires the balance of all the accounts to always be positive. If a transaction (update) makes an account balance negative after completion, resulting in the system being in an invalid state, then it is not considered a valid update.
- Isolation: Isolation means that during the partition, the system can only operate on one side at most.
- Durability: This guarantees that after the update is completed, the update will be a permanent update and will not be lost, even in the event of a system failure.

3.5 Simple Object Access Protocol (SOAP) and Representational State Transfer (REST)

SOAP and REST are two possible answers to the question, “In order for client and server to communicate efficiently, what kind of web application design should be used in a specific situation?” In fact, SOAP is a protocol, and REST is an application design style; however, from an application design point of view, SOAP can be treated as a design style as well.

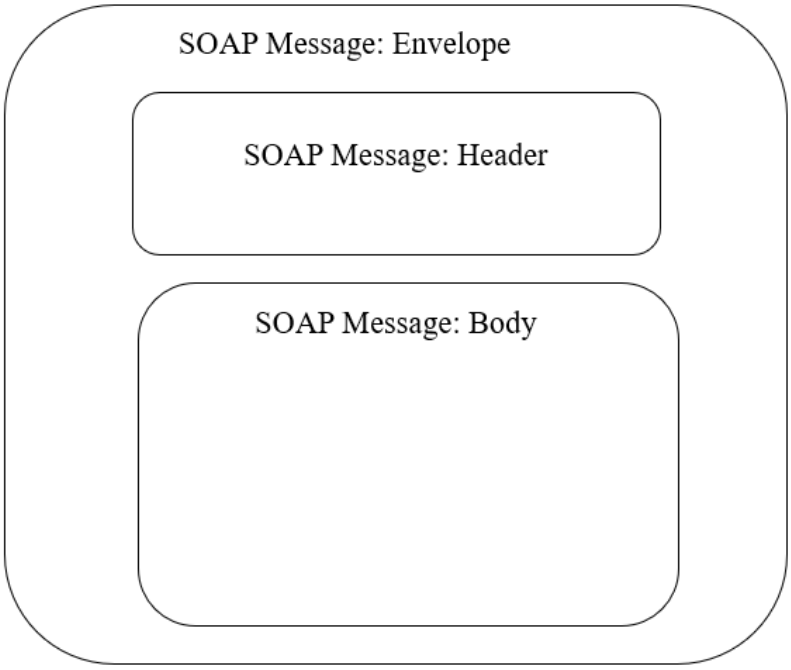


Figure 3.3 SOAP Structure [20]

```
<?xml version=“1.0” encoding= “utf-8”? >
<soap:Envelope xmlns:soap="http://www.example.org/soap-envelope" >
   <soap:Header>
   </soap:Header>
   <soap:Body>
       <message>Hello World</message>
   </soap:Body>
</soap:Envelope>
```

Figure 3.4 SOAP Message [21]

3.5.1 SOAP

SOAP is an acronym for Simple Object Access Protocol. Hadley et al. [21] describe it as a "protocol intended for exchanging structured information in a decentralized, distributed environment."

SOAP uses XML as its message format, which allows SOAP to exchange rich and structured information between programs and applications. SOAP most often uses HTTP as its application layer protocol, but some systems use SMTP instead of HTTP. Another advantage of SOAP is that a SOAP application can also become a client of another SOAP service, which gives SOAP the ability to aggregate powerful web services, making it a robust programming model. The basic structure of the SOAP message is shown in Figure 3.3. A SOAP message has elements including envelope (required), header (optional) and body (required) An envelope is used to encapsulate all the details of a SOAP message. It does this by identifying the XML message as a SOAP message and wrapping a header and a body. Headers contain the metadata, and the body contains the payload. Figure 3.4 shows an example of a SOAP message in XML.

3.5.2 REST

REST stands for Representational State Transfer. Fielding [22] proposed this application architectural style in his PhD thesis in 2000. REST does not have any rules about how this should be done at an implementation level, but instead provides a guideline for the design level. REST often runs on top of HTTP and is commonly used in web service development. Web services using REST as their guideline for designing their applications are often called RESTful web services. In REST, defining resources is one of the first steps. Each resource will contain an identity to represent itself and should not be too large but still contain all the information it represents. Resources can also contain links to other resources if they are related. In a RESTful web service, web pages are usually defined as a resource and the URI is used as the identity of the resource. RESTful web services also make good use of the Hypertext Transfer Protocol (HTTP). Clients use HTTP to communicate with the server and use HTTP request methods GET, POST, PUT and DELETE to retrieve, create, update, and delete resources. In addition to defining resources, a REST design will also have the following architectural constraints.

- Uniform interface: Each resource should have one and only one URI, and it will never be confused with other resources by the client.
- Client Server: The client and the server are not dependent on each other. While the interface between server and client stays the same, both server and client can develop their own content or even delete all the content without interfering with one another.
- Stateless: The interaction between the client and the server is stateless, which means the server will not store the client's last visit information. All the requests made by the client are treated as new requests. If the client runs a stateful application, then it must contain all the information necessary in each request. For example, a client wants to do an update after login; however, a RESTful service server will not hold the login information for the client, so the client needs to include the authentication result or login information in the update request.
- Cacheable: All resources should apply caching to themselves when caching is possible and mark themselves as cacheable. This will gradually improve the performance for the client side and server side. Caching could be implemented on either client or server side.
- Layer system: A REST system can be a multiple layered system. For example, you can have all APIs on Server A, mass computing routines on Server B, and have all the data stored on Server C. This allows the communication between the client and the server to continue even if Server B or Server C go down for maintenance or any other reason.
- Code on demand (optional): Although most of the time, a RESTful service will return the resource static representation state to the client, it is possible to return an executable code. For example, JavaScript can be sent to the client to reduce the computing power needed on the server. This is commonly seen on web page UI interfaces.

3.5.3 REST vs. SOAP

SOAP is a protocol, whereas REST is an architectural design, but it is still possible to compare them from a designer point of view. Below are the differences between SOAP and REST.

```
{“MESSAGE": "Hello World”}
```

Figure 3.5 REST Payload

- Complexity and Bandwidth: SOAP adds another layer on top of the application layer, which gradually increases the size of the payload. REST, on the other hand, tries to utilize the application layer functions, which reduces the size of the payload and reduces the bandwidth needed. Figure 3.5 shows an example of a payload used in REST. In comparing it to Figure 3.4, both figures try to send the message “Hello World” in their payload. The size of the payload is significantly different.
- Message Format: SOAP can only use XML. REST can use many different data formats including JSON, HTML, XML, or even custom data formats, which makes it more compatible for different clients.
- Security: When a certain level of authentication of a client is required, SOAP standard SOAP 1.2 provides lots of features. REST, however, has no such security features, so, when the need for data security is high, REST is not the first choice.
- Caching: SOAP has no data caching. REST supports data caching on both server and client sides. This means a client can reuse this data without making a new request to the server. This is a significant advantage in an intermittent connection environment.
- State Maintained: SOAP supports the application to maintain the client state between requests. In REST, client state is not stored in the server.

3.5.4 REST or SOAP

When connection stability is reliable and the history of the client request is needed, SOAP is probably the better choice; however, in an intermittent connection environment, using REST to design applications seems to be a better idea than using SOAP. Since a REST-designed application

requires lower bandwidth and has a caching feature, it is more friendly in an unstable connection situation than a SOAP-designed application.

3.6 Hypertext Transfer Protocol (HTTP) and Constrained Application Protocol (CoAP)

Both HTTP and CoAP are application layer protocols. HTTP is very famous and frequently used across the Web. CoAP on the other hand is more commonly seen in the IoT world.

3.6.1 HTTP

HTTP stands for Hypertext Transfer Protocol. It is designed to create or improve the communication between clients and servers. In 1989, Tim Berners-Lee initiated the development of HTTP at the European Organization for Nuclear Research (CERN). Tim Berners-Lee and his team then released a series of RFC documents, and one of the most famous is RFC 2616 [23], a widely used (before 2014) version of HTTP (HTTP/1.1). Later in 2015, HTTP version 2 (HTTP/2) was published and is presently used by more than 50% of websites.

Features

Listed below are some interesting features of HTTP.

- Synchronous: A client sends a request to the server, then the client waits until the server responds.
- Underlying Protocol Independent: Though most HTTP connections in the world are based on TCP/IP protocol, HTTP does not require using TCP/IP as its transport protocol. It is possible to adapt UDP or other similar protocols as the underlying protocol for HTTP.
- Stateless: After initiating a connection and receiving a response from a server, the client will close the connection. If the client needs to send more requests to the server, it cannot use the old connection, but has to initiate a new connection.

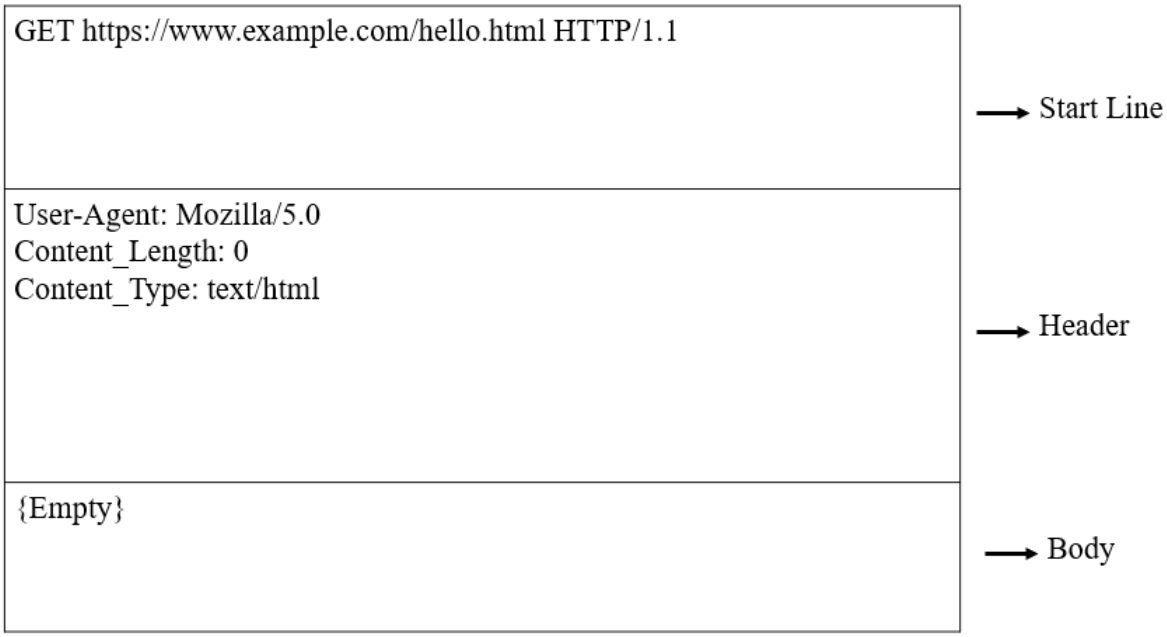


Figure 3.6 HTTP Request/Response Model

Request/Response Model

Each TCP connection can only process one HTTP request and response. In order to process multiple requests, a client needs to open multiple connections simultaneously, and this is inefficient and resource consuming. Figure 3.6 shows an example where a client requests a web page at first, then after receiving the response it opens two more requests simultaneously to retrieve images used in the web page. A total of three TCP connections are created to fetch one item, which does not seem to be very efficient. In HTTP/2, the number of connections needed to fetch the same content has been gradually reduced due to HTTP/2 features such as server push and multiplexing multiple requests over a single connection. HTTP protocol is still considered to be a synchronous and stateless connection protocol.

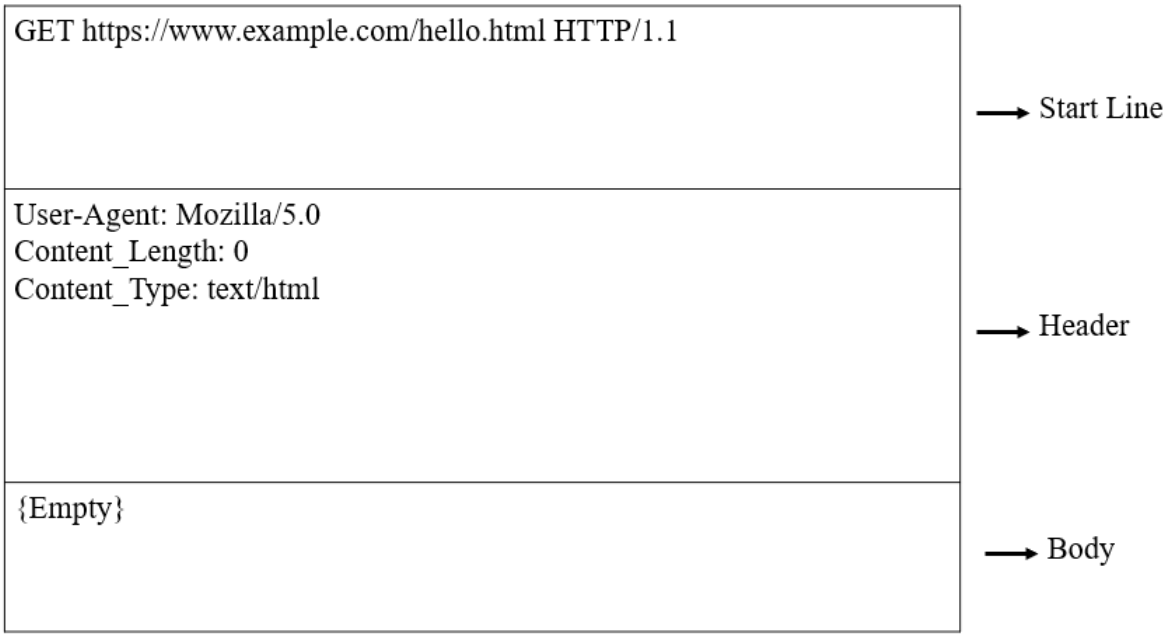


Figure 3.7 HTTP Request Message [24]

Message Format

A request message is shown in Figure 3.7. It consists of three sections: start line, header, and body.

- The start line contains three elements: First, an HTTP operation such as GET or POST, second, a Uniform Resource Identifier (URI) like https://www.example.com/hello.html, and third, the HTTP version such as HTTP/1.1.
- The header section contains multiple headers. Each header provides different information such as browser information, payload size, or payload datatype. Every HTTP header follows a basic structure, a name string followed by a colon (‘:’), then a value string.
- The body is where payload appears. Not all HTTP requests have a body. A simple GET request normally does not contain a body in its request.

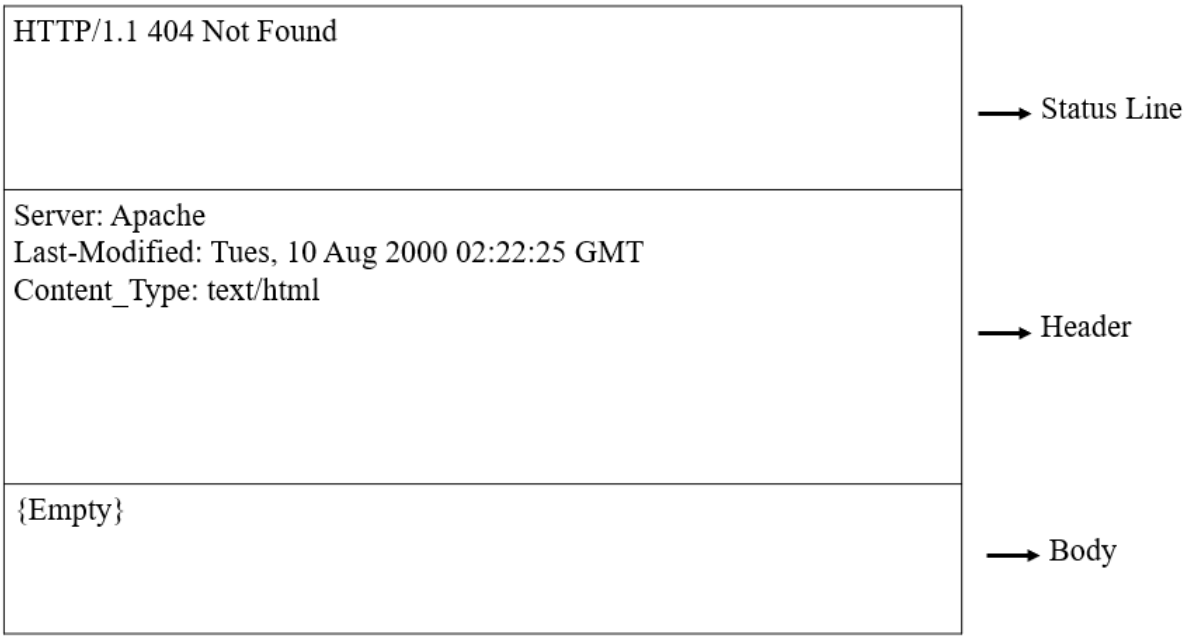


Figure 3.8 HTTP Response Message [26]

Like a request message, a response message consists of three sections: status line, header, and body. See Figure 3.8 as an example.

- The status line contains the HTTP version such as HTTP/1.1, status code such as 404, and a status text such as Not Found.
- The headers in the response message have the same structure as a request message header structure. The information contained in the response header section are server type, payload last modified date, payload type, etc.
- Like request messages, not all response messages contain a body. The payload is only attached to the message when it is needed.

3.6.2 CoAP

CoAP is the abbreviation for Constrained Application Protocol. It is an Internet Application Protocol defined in RFC 7252 [25] in 2014 for constrained networks such as low-power networks, and constrained devices such as microcontrollers. CoAP is commonly used for machine-to-machine (M2M) applications. The default protocol for CoAP is User Datagram Protocol (UDP).

Features

Listed below are some features of CoAP.

- Emulating HTTP: CoAP uses similar operation codes to HTTP like GET or POST, which make it easy to translate to HTTP for integration.
- Asynchronous: After the client sends a request to the server, the client can continue working before receiving the response from the server.
- Overhead: The smallest CoAP message is only four bytes if it does not contain a token, option, and payload. Due to its compact size, it has low overhead and is easy to parse.
- Caching: To efficiently respond to a request, CoAP is designed to support caching on an end point and on an intermediary such as a proxy.

Message Types and Request/Response Model

The four types of messages used in the CoAP are confirmable message, non-confirmable message, acknowledgment message, and reset message. A message exchange cycle in CoAP consists of two messages. The first message is either a confirmable or a non-confirmable message, which is sent from the sender to the receiver. If the first message was a confirmable message, then a second message is sent back from the receiver to the sender, which can be either an acknowledgment or a reset message. If the first message was a non-confirmable message, no second message will be sent.

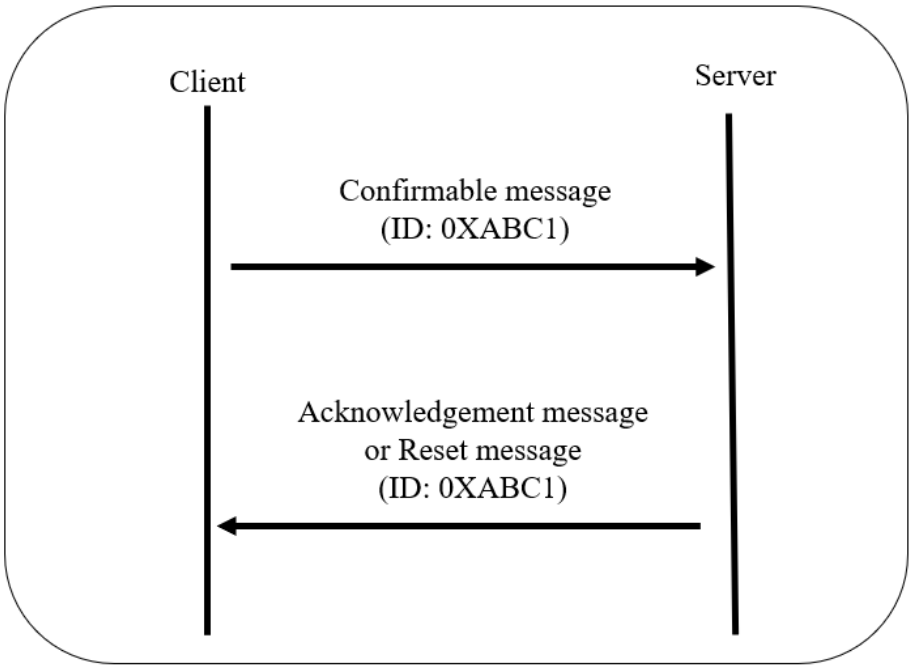


Figure 3.9 CoAP Confirmable Message [26]

Confirmable messages are reliable messages when sending messages to a server. Confirmable messages are sent to the server repeatedly until the server sends a response message back. It is important to note that the ID of both request message and response message must be the same. Figure 3.9 shows the process for exchanging a confirmable message as a request message and then receiving an acknowledgment message or reset message as a response message.

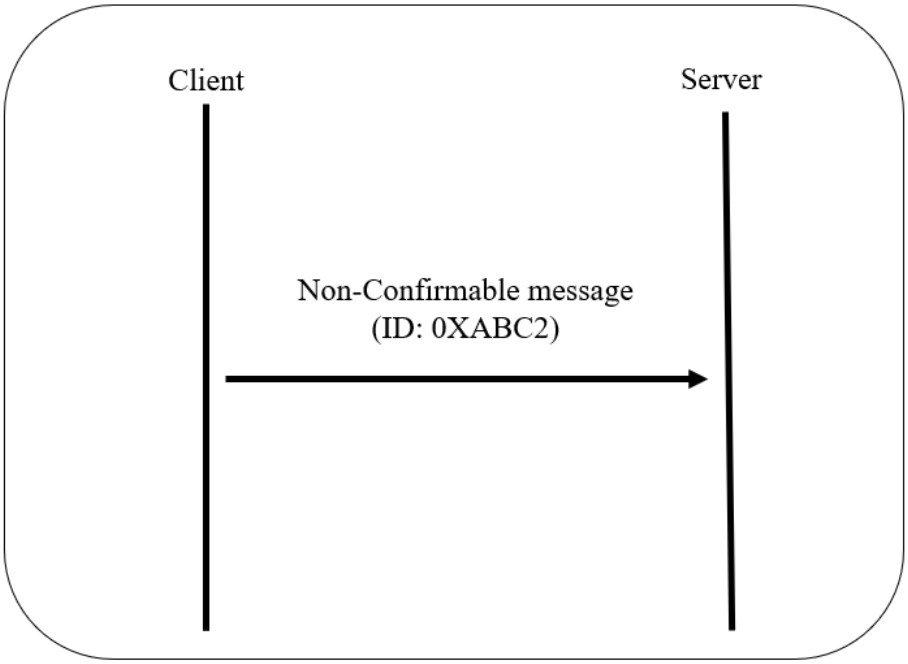


Figure 3.10 CoAP Non-Confirmable Message [26]

As Figure 3.10 shows, a non-confirmable message is an unreliable message since it is not required to receive a response back from the server. Non-confirmable messages are often used to send noncritical messages to the server. For example, this applies to information read from a sensor. A non-confirmable message still contains a unique ID as well.

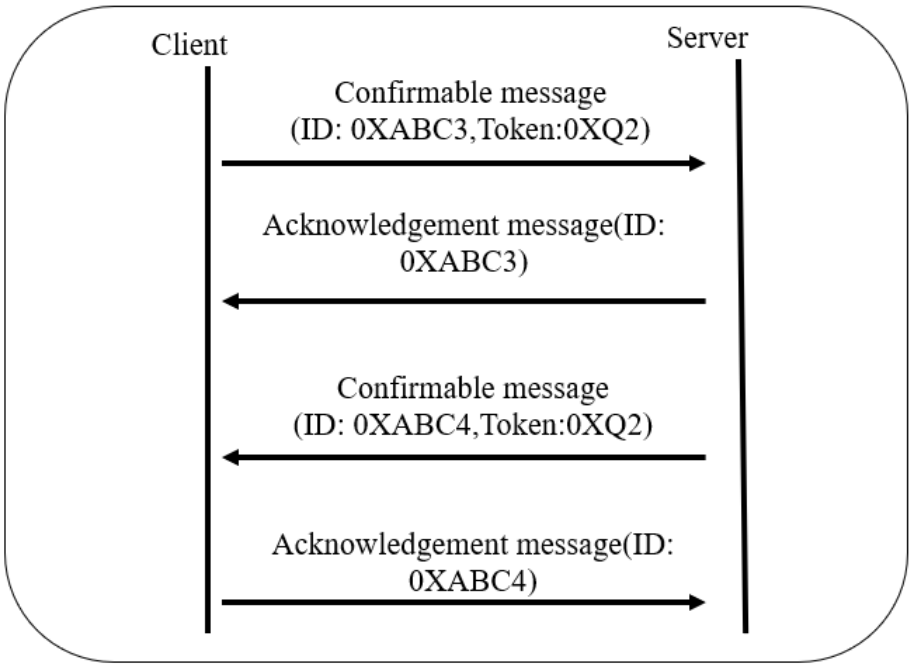


Figure 3.11 CoAP Request/Response [26]

It is possible that a server receives a confirmable message but cannot provide the result immediately. In this case, the server still needs to send an acknowledgment message with an empty result back to the client. A token, which is contained in the confirmable message, is used in this case to match the query and the result. In Figure 3.11, after the server sends back the acknowledgment message to the client and finishes the process for the query, the server sends a new confirmable message to the client with a different message ID but the same token.

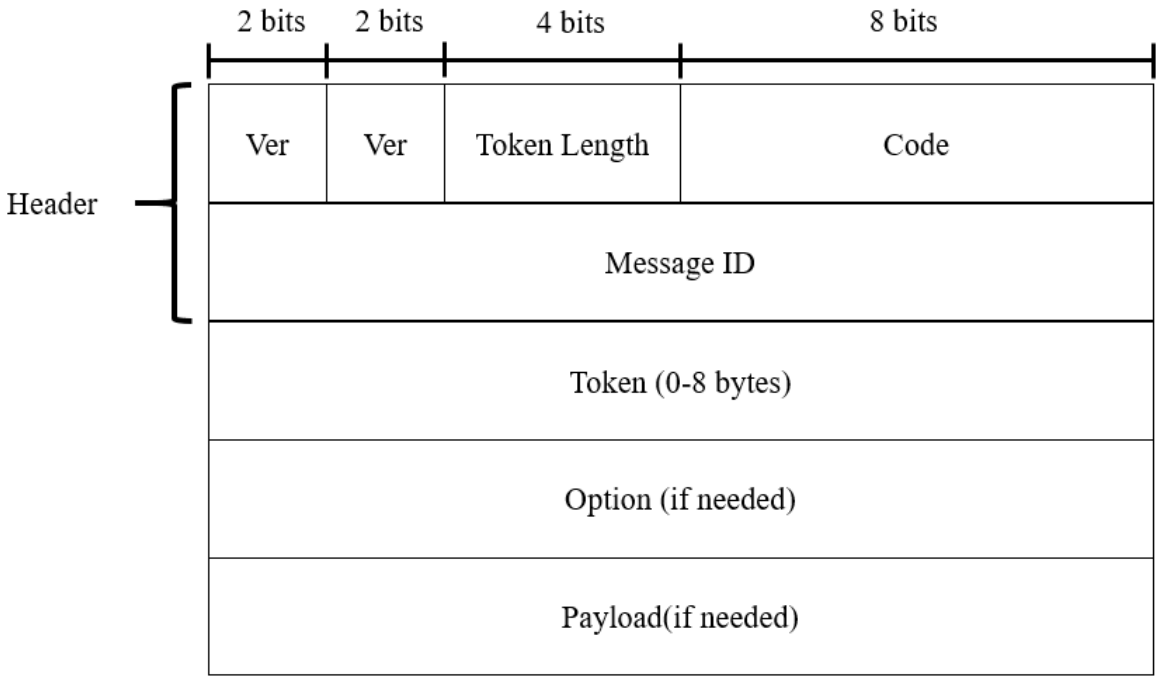


Figure 3.12 CoAP Message Format [26]

Message Format

The entire CoAP message needs to fit within a UDP datagram to avoid fragmentation. Figure 3.12 shows the message format of CoAP. The first five parts of the CoAP message together are called the CoAP header.

- The first two bits are used to store the version number.
- The next two bits are used to store the message type mentioned above. A confirmable message is stored as 0 and a non-confirmable message is stored as 1.
- The next four bits are used to tell the length of the token, which can have up to eight bytes of length.
- The next eight bits are used to store the request/response code, like an HTTP status code. For example, the HTTP GET method is stored as 1 in CoAP message code and the HTTP POST method is stored as 2 in CoAP message code.
- The last sixteen bits are used to store the message ID, which matches the acknowledgment/reset message with the confirmable message.

Altogether, the CoAP header length is four bytes in length and is also the smallest size a CoAP message can have. A token is stored after the header section, followed by the options section, then the payload section.

3.6.3 CoAP vs. HTTP

- Synchronous: HTTP is synchronous and CoAP is asynchronous. This makes HTTP less efficient unless multi-processing is used.
- Underlying protocol dependent: HTTP is not dependent on specific transport layer protocol, but CoAP is designed to work on top of UDP. In the situation where UDP is not an available choice, using CoAP might cause more problems and result in more work to do.
- Caching: Both HTTP and CoAP support caching, which reduces the reliability of the data but increases the performance.
- Session state: HTTP and CoAP are both stateless protocols. Servers using HTTP or CoAP do not need to keep information about the client's previous requests. A stateful application or web service must find another way to store user information, such as browser cookies or extra variables.

3.6.4 CoAP or HTTP

In an environment where multi-processing is not an issue and connection bandwidth is high, using HTTP will not be a problem; however, in a constrained environment where processing power and connection bandwidth are low, CoAP should perform better than HTTP.

3.7 Summary

The goal of this research is to build a system to solve the problem of maintaining consistent updates in a distributed system. The descriptions in this chapter give ideas about how to achieve this research goal. The system needs to be a context-aware application so it can be informed when a resource state has changed (2.2). Since this system is assumed to be used in an IoT system, the unique ID feature of the IoT device will identify the same prime resource used in different systems (2.4). The system also needs to adopt the idea of a truth maintenance system so the dependencies

between resources can be tracked (2.1) and the resource state change can be propagated to dependent resources (2.3). The system will also be an accessible and partition tolerant (AP) system from the CAP point of view, so a resource can still be updated when not all of its primary resources are reachable (2.6). Last, the service layer protocol of the system will use CoAP to inform other system when a primary resource has a new state (2.5).

CHAPTER 4

SYSTEM ARCHITECTURE

This research presents a system to reduce the inconsistency of resources stored in machines in an intermittent connection environment. This system is called Local Resources' State Management System (LRSMS). The system is assumed to be a stand-alone system and runs as a service in a device to help manage the resources' state in a machine. The word "Local" in the name means the system only tries to manage the internal resources' state; however, it does not mean LRSMSs cannot communicate, exchange information, and help each other to manage the resources in other LRSMS devices. The word "Resource" in the name refers to all the resources in the device, no matter if the resource is generated from the device itself, such as a smartphone location, or the resource is a cache value received from other devices, such as an IoT device. Given that this system is trying to maintain the consistency of the data in the local device, its better to start by defining resources in this architecture.

Resource

A resource can be information received from other devices or information generated from a local device. Information received from other devices is often stored as cache data and is not easy to keep up to date. Examples of information received from other devices can include a web page downloaded from the Internet, a temperature read from a thermal reading sensor, or an incoming phone call received by a smartphone. Information generated from a local device is often stored as regular data and is easier to keep up to date compared to the cache data received from other devices; however, these kinds of data are mostly generated based on other cache data, so if the source cache data is not up to date, then it is hard to say that this data is up to date. Examples of information generated from a local device can include a wake-up alert, a grocery shopping list, or a message sent from a smartphone. Every resource stored in the device has a state and when an update occurs, the resource state will change as well as its dependent resources.

4.1 General Architecture

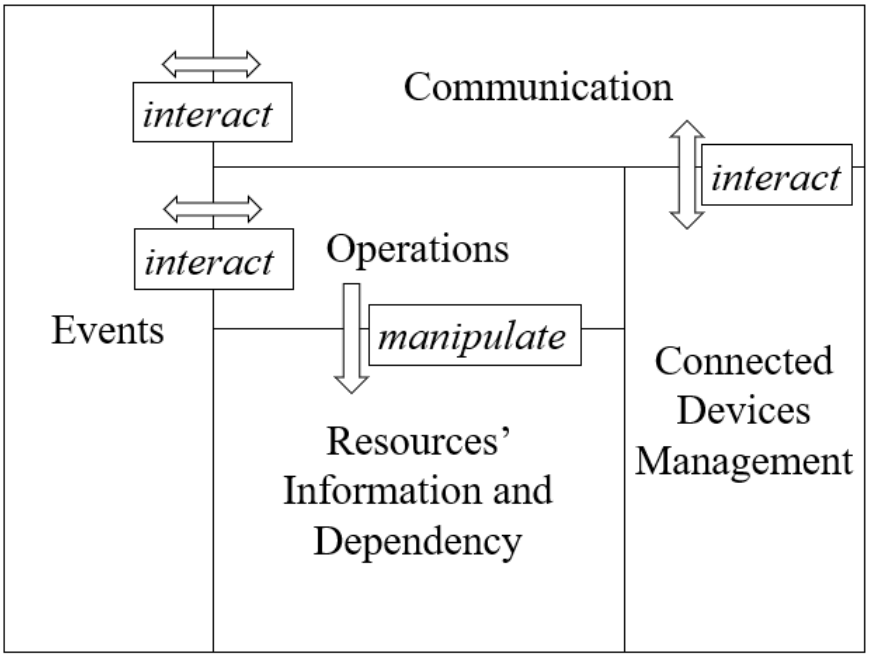


Figure 4.1 LRSMS Components

Figure 4.1 shows the overall picture of a LRSMS. A LRSMS consists of five components: an events component, a resources' information and dependency structure component, an operation component, a connected devices management component, and a communication component.

Referring back to the previous example in Chapter 1, Figure 1.1, after the connection is re-established, a synchronizing event occurs between the LRSMS system (System 1) in Central Machine 1 and the LRSMS system (System 2) in Central Machine 2. Both System 1 and System 2 use their own communication component to communicate with each other. During the synchronizing event, the first step from System 2's point of view is to add System 1 to its connected devices management component. Then, the second step triggers the get all resources' ID operation, the get state operation, and the "Get Resource Operation" in System 2 in sequence in order to retrieve the latest data stored in Data 1, which is the latest outdoor temperature. After getting the latest Data 1, System 2 will update Data 1 state information that is stores in the resources' information and dependency component. The third step is to update Decision 2 based on the new Data 1 and store the new Decision 2 state information in the resources' information and dependency

component as well. At the end of the synchronizing event, the copy of Data 1 in System 2 is updates, as is Decision 2.

4.1.1 Events

Events handle requests received from applications in the device as well as from other LRSMS in the other devices. Events in LRSMS trigger one or more operations and may trigger another event as well. Events result in a series of changes in the LRSMS.

4.1.2 Resources' Information and Dependency

Only resource information such as resource ID and resource state are stored in the resources' information and dependency component; the resource data is not. Resources' dependencies are also stored in this component. The dependency between resources is such that one resource generates its content based on another resource. For example, Resource A stores the temperature read from a thermal reader and Resource B is a decision to go for a walk or not based on the temperature stored in Resource A. In this case, Resource B is depending on Resource A and a dependency relationship between Resource A and Resource B is established.

4.1.3 Operations

Operations are the key component in the LRSMS. The main role of an operation is to manipulate the resource information stored in the resources' information and dependency component. The seven types of operations used in the LRSMS are Create, Get All Resources' ID, Get State, Get Resource, Update Alert, Update, and Delete.

4.1.4 Connected Devices Management

The connected devices management component stores the information about connected devices and the resources stored in those connected devises. Connected devices management is used by the communication component to send out or redirect resource state change information.

4.1.5 Communication

The communication component is used to exchange information between LRSMSs and to interact with the local device applications. It tells the operation component what to do based on messages

it receives, and it sends out messages to applications in the local device or the LRSMS on other devices base on operation component direction. This research will use CoAP as default protocol.

4.2. Events

There are six kinds of events in a LRSMS system: register, get resource, update alert, update, synchronize and delete events.

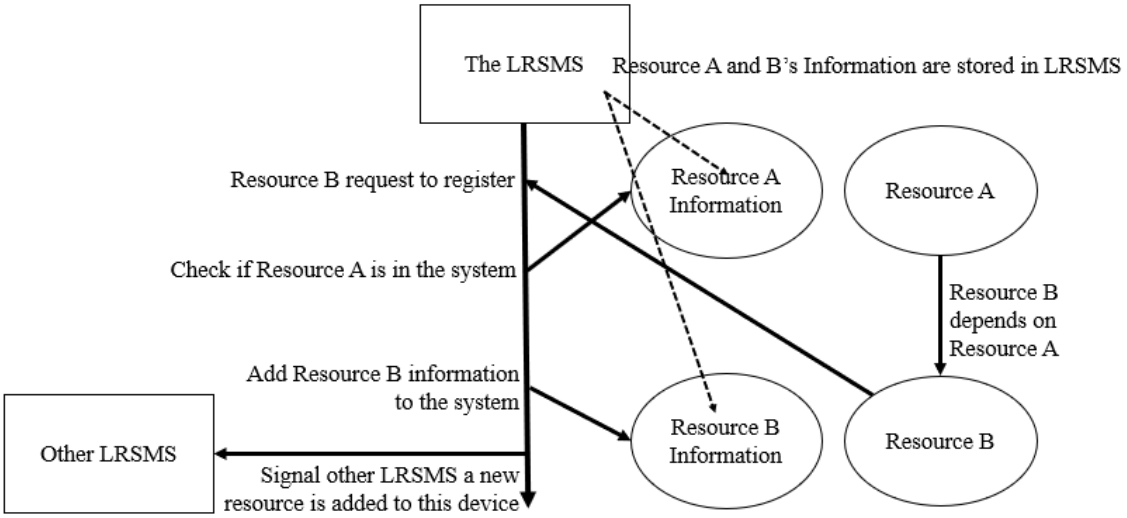


Figure 4.2 Register Event Example

Register Event

"Register Event" occurs when an application in the device wants to add a new resource's information to the LRSMS. The "Create Operation" is the only operation being used in this event. This event will try to add a new resource's information into the resources' information and dependency component of the LRSMS. In the case where one of the new resource's supporting resource information pieces does not exist in the LRSMS, the new resource's information will not be added to the system. The LRSMS will also signal other connected LRSMSs to let them know a

new resource has been add in this device. Figure 4.2 shows an example of a “Register Event”. Resource B tries to register itself to the LRSMS. Since Resource B depends on Resource A, the LRSMS first checks if Resource A has been registered with the system, then it adds Resource B’s information into the Resources’ Information and Dependency component and then signals the other LRSMS at the end.

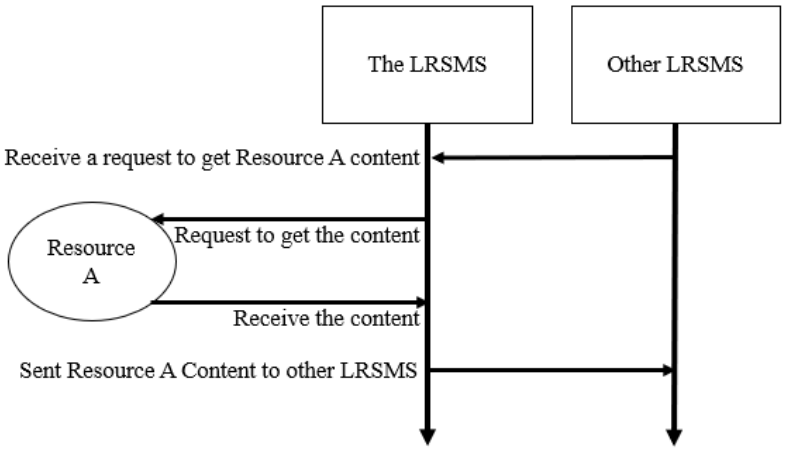


Figure 4.3 Get Resource Event Example

Get Resource Event

The “Get Resource Event” happens when other LRSMSs in other devices want to get the latest resource content from the local device. “Get Resource Operation” is the only operation being used in this event. When the LRSMS receives a “Get Resource Event”, it will get the content from the resource in the device and send that content to the requesting LRSMS using the communication component in the system. Figure 4.3 shows an example of a “Get Resource Event”. The LRSMS first receives a request from other LRSMS to get Resource A content in the device. The LRSMS then retrieves Resource A content, and then sends the content to the other LRSMS.

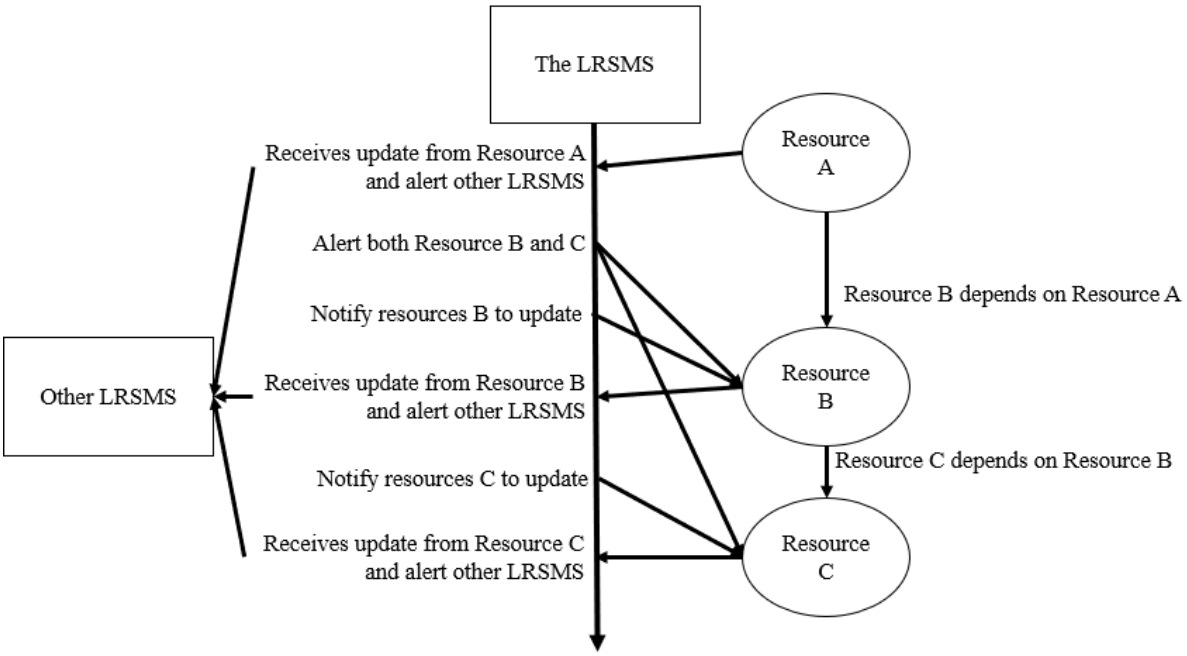


Figure 4.4 Update Event Example

Update Event

An "Update Event" happens when a resource in the device has a new state. "Update Alert Operation" and "Update Operation" are both being used in this event. Figure 4.4 shows how an LRSMS handles an "Update Event. When the LRSMS receives an update of Resource A, the LRSMS sends alerts to all other connected LRSMSs to let them know Resource A now has a new state. Then, the LRSMS sends update alerts to both Resource B and Resource C, indicating that their content might be out of date and need an update. Then, the system notifies Resource B to update its content first since Resource B only depends on Resource A and Resource A is up to date. After the LRSMS receives an update message from Resource B, it sends alerts to all other connected LRSMSs again to let them know Resource B now has a new state. Then the LRSMS notifies Resource C to update its content. Lastly, the system receives an update from resource C and sends alerts to all other connected LRSMS again to let other LRSMS know Resource C now has a new state.

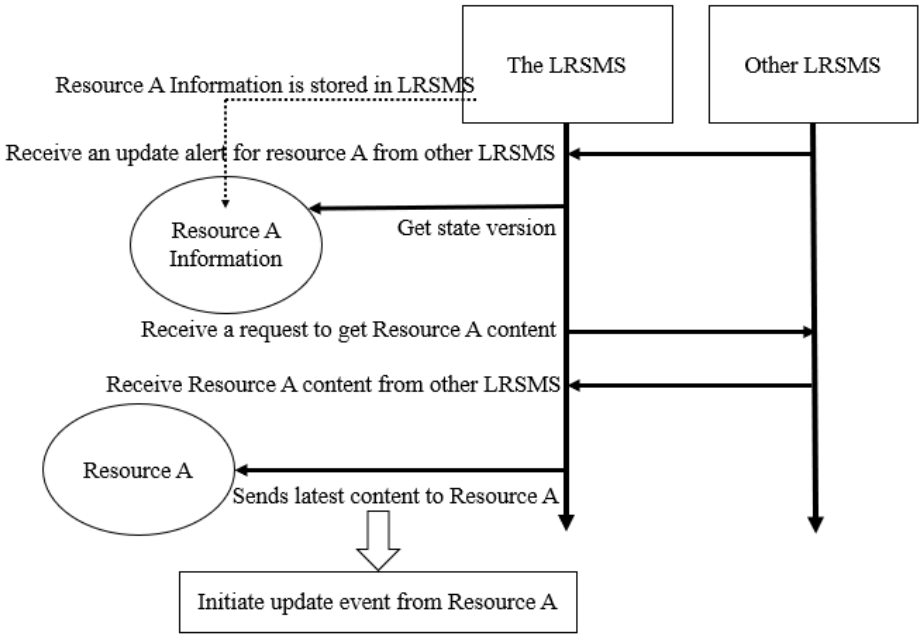


Figure 4.5 Update Alert Event Example

Update Alert Event

An "Update Alert Event" happens when an application in the local device wants to get the latest resource content from another device. "Update Alert Event" uses the get state operation in the system. The LRSMS first receives an update alert from another LRSMS, then it uses the get state operation to make sure this new update alert has a newer version than the version in the system. Then, the LRSMS sends a request to the other LRSMS and asks for the latest resource content. After it receives the resource content, it will pass the content to the application and an "Update Event" will subsequently occur. Figure 4.5 shows an example of an" Update Alert Event". The LRSMS first receives an update alert from another LRSMS for Resource A. The LRSMS checks with the resource information stored in the system to make sure Resource A's version stored in the system is older than the new one. Then, the LRSMS sends a request to the other LRSMS to get Resource A content from the other devices. After receiving Resource A content, the LRSMS sends it to the application in the local device. This event is normally followed by an "Update Event".

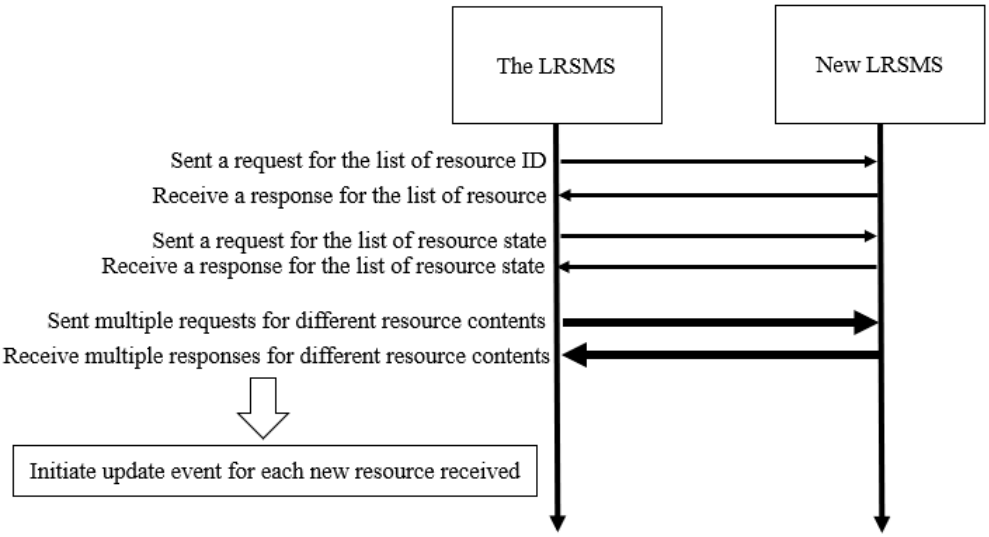


Figure 4.6 Synchronize Event Example

Synchronize Event

"Synchronize Event" happens when a new LRSMS connects to the system. Its an aggregate of get all resources' ID, get state, get resource, update alert and "Update Operations". "Synchronize Event" tries to make all the resources in the device perform an update based on the new resources' state received from the new connected LRSMS. Figure 4.6 shows an example of a "Synchronize Event". The LRSMS first asks to get all resources' ID in the New LRSMS. After receiving the list of resources' ID, the LRSMS asks for the resource state for all the resource it shares with the New LRSMS. After receiving the states of all the resources, it shares them with the New LRSMS, and checks if the resources in the new LRSMS have a newer state version than those stored int he LRSMS. Lastly, the LRSMS sends multiple requests to get resources which have a newer version than the local system has. This event is normally followed by one or more "Update Events".

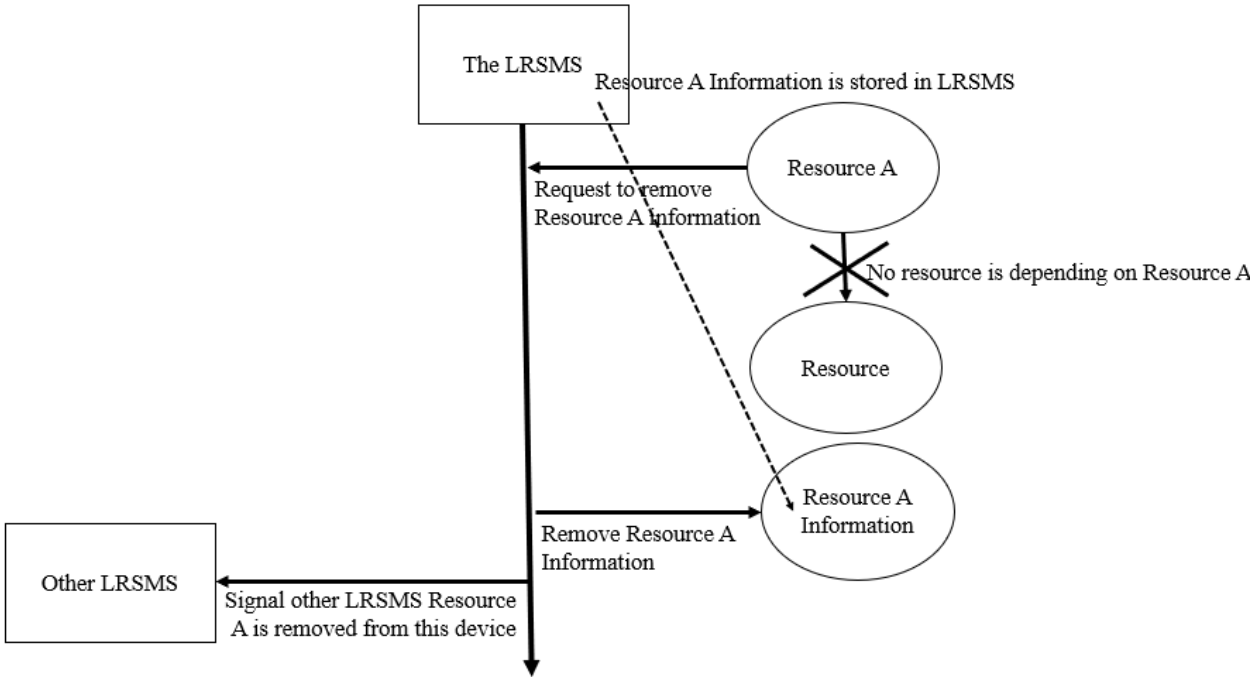


Figure 4.7 Delete Event Example

Delete Event

"Delete Event" happens when an application in the device wants to remove one of its resource's information from the LRSMS. "Delete Operation" is the only operation being used in this event. When the LRSMS receives a "Delete Event", it checks if that resource has any dependent resources. If there is one resource depending on the delete target resource, the "Delete Operation" will terminate. If there are no resources depending on the delete target resource, the target resource's information will be removed from the system and the system no longer monitors its state. The LRSMS will also signal other connected LRSMSs to let them know a resource has been removed from this device. Figure 4.7 shows an example of a delete resource event. The LRSMS receives a request from Resource A to delete its information from the system. Since no other resource is depending on Resource A, the system then removes Resource A's information in the Resources' Information and Dependency component then signal other LRSMS at the end.

4.3 Resources' Information and Dependency Structure

The Resources' information and dependency structure component stores resources' information and the dependencies between resources. It can be considered as the database of the LRSMS.

4.3.1 Resource Information

Figure 4.8 Resource Information

Resource information is stored as a key-value pair in an LRSMS where the key is the ID of the resource and the value is the resource's information. The LRSMS uses a map (dictionary) to collect all the resource information key-value pairs. This should allow an LRSMS to find a specific resource's information faster compared to storing resources' information in an array.

As Figure 4.8 shows, six things are stored in resource information: ID, supporting resources, dependent resources, timestamp, the resource reference, and flag.

- ID is used to identify the resource. Although no specific format of the ID is required to be used in a LRSMS, as long as the ID used in the system is unique and format is the same, it is recommended to use a global standard to ID the resource so when connected to other

systems, the other systems can identify the resource as well. In this research, a Uniform Resource Location (URL) will be used as the ID of the resource.

- Supporting resources store a list of the resources' IDs which the resource used to generate its content.
- Dependent resources store a list of resources' IDs of the resource dependents.
- Timestamp stores the resource's last update time. No specific format is required to store the timestamp, but the same timestamp format should be used in a LRSMS. One thing to remember is when storing the timestamp, it should either be stored in UTC, or the time zone information should be included.
- Resource reference is used to connected to the resource itself and could be a call back function or a link.
- Flag is a Boolean value. By default, flag should be set to false. Flag set to true means either the resource is updating its content now or at least one of the resources it depends on is out of date. If a resource information flag is set to true and all the resources it depends on are up to date (all resources' information flag are set to false), then the LRSMS will try to notify the resource to update its content.

4.3.2 Resources Dependency Structure

Both dependent resources and supporting resources are stored in the resource information. In general, this is a redundancy, and it is a waste of memory space since the same resource dependency is stored in two different resources; however, this will increase the performance of the LRSMS. The dependent resources list is used in an "Update Alert Operation" and the supporting resources list is used in an "Update Operation". Removing either one will cause the system to iterate through every resource information in a certain operation, and waste more and more time when the quantity of resources information stored in the system is higher and higher.

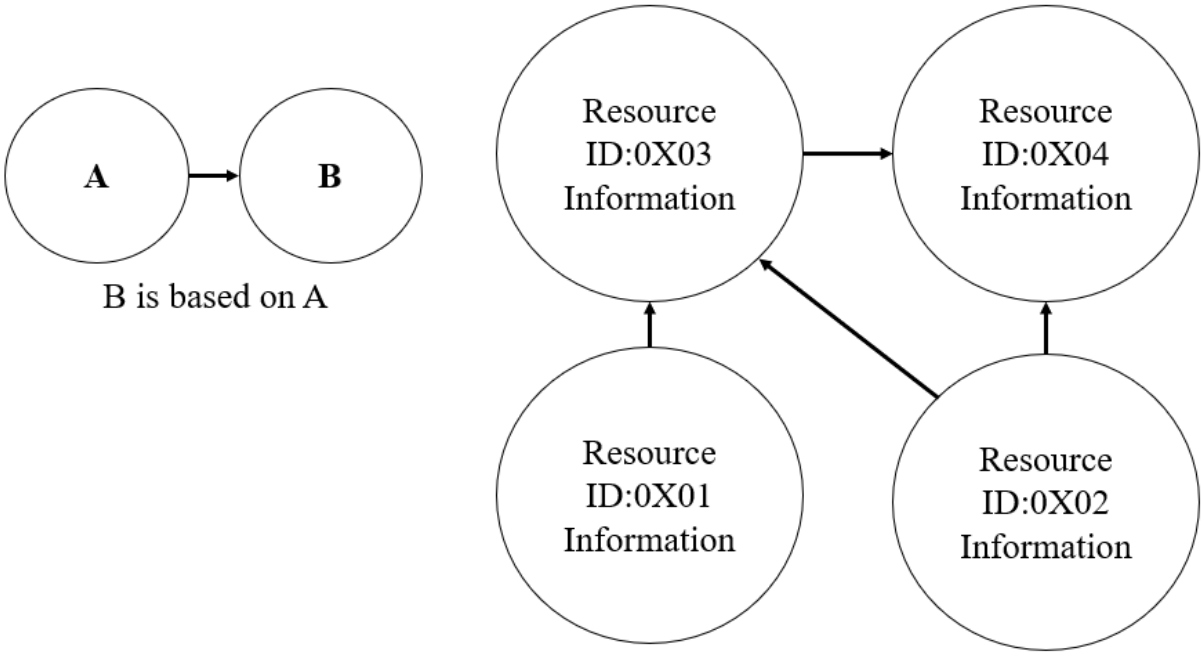


Figure 4.9 Resource Dependency Structure Example

Figure 4.9 shows an example of resources dependency structure. Resource 0X01 is not depending on any resource; it has an empty supporting resources list, and a dependent resources list contains only one element {0X03}. Resource 0X02 is not depending on any resource and it has an empty supporting resources list and a dependent resources list containing two elements {0X03,0X04}. Resource 0X03 is depending on both Resource 0X01 and Resource 0X02, so it has a supporting resources list with two elements {0X01,0X02} and a dependent resources list with one element {0X04}. Lastly, Resource 0X04 is depending on both Resource 0X03 and Resource 0X02, but no other resource is depending on it, so it has a supporting resources list with two elements {0X02,0X03} and an empty dependent resources list.

4.4 Operations

An LRSMS uses different operations to complete its tasks or events. There are seven types of operations used in the LRSMS: create, get all resources' ID, get state, get resource, update alert, update, and delete operations.

4.4.1 Create Operation

The “Create Operation” adds a new resource information as a key value pair into the Resources’ information and dependency component. The resource ID, the supporting resources ID list, the last update time of the resource and the resource reference are needed in order to make the new resource information. This new resource ID will be added to all supporting resources information’s dependent resource list. If any supporting resource is not in the system, the “Create Operation” will fail and no new resource information will be added into the system,

4.4.2 Get All Resources’ ID Operation

The “Get All Resources’ ID Operation” returns all the resources’ ID stored in the LRSMS. Nothing needs to be provided to get the result of this operation. This operation is mostly used when the LRSMS connects to a new device and a new LRSMS is found.

4.4.3 Get State Information Operation

The “Get State Information Operation” returns the resource state version which is the timestamp of the last update. A resource ID is the only thing that needs to be provided to get the version of the resource. This operation is mostly used when the LRSMS receives an update of a resource signal from other devices and wants to check if that update is a newer update or an older update.

4.4.4 Get Resource Operation

The “Get Resource Operation” returns the resource content, or the resource cached in the device. Like the “Get State Information Operation”, only a resource ID needs to be provided to get the latest version of that resource. This operation is mostly used when a LRSMS on the other device finds out the local device has a newer version of a resource it uses.

4.4.5 Update Alert Operation

The “Update Alert Operation” sets the flag of the resource information to 'true' and uses the resource reference stored in resource information to alert the resource that its content could be out of date. A recursive function is used in this operation to propagate the alert operation to all related resources until all related resources information’s flags are set to 'true', and all related resources receive an alert call. A resource ID is the only thing needed to start the “Update Alert Operation”.

This operation is mostly used after an LRSMS receives an update signal and passes the state version check.

4.4.6 Update Operation

The “Update Operation” notifies a resource to update its content. If at least one of the supporting resources information’s flags is set to true, then this operation will be terminated since it is not advantageous to update content using outdated data. Giving the resource ID to the “Update Operation" will start the procedure.

4.4.7 Delete Operation

The “Delete Operation” will remove the resource information stored in the system. Resource ID is the only thing that needs to be provided to do this operation. This operation will fail if the resource dependent list has at least one element in it, as this indicates that there is another resource in the device that is using the resource as a supporting resource and thus the resource shall not be removed.

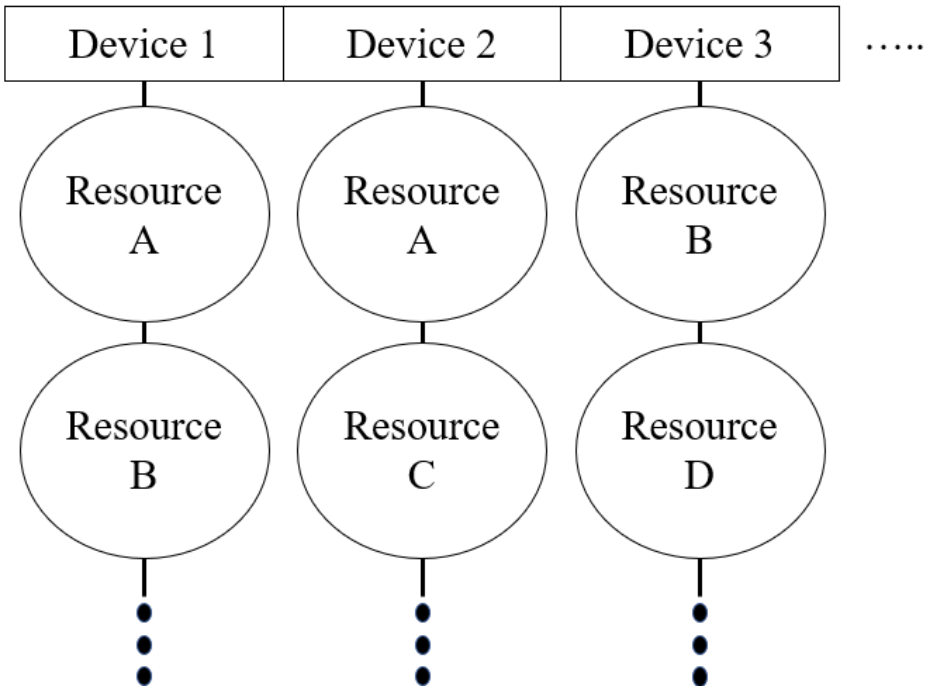


Figure 4.10 Devices and Resources in Connected Devices Management

4.5 Connected Devices Management

The connected devices management component stores connected devices and corresponding resources' ID in each device as a collection of key-value pairs (A Map or a Dictionary). The keys are the connected devices' IDs, and the value is a list of sources' IDs stored in the corresponding device's LRSMS. A device is added to the map (or dictionary) when a new device is connected to the local device. The list of the resources' IDs is generated during the "Synchronize Events". The device is removed from the map (or dictionary) when the LRSMS tries to send a message to the device and receives a time out response, which suggests the connection is lost. The resource list stored in each key-value pair changes when the device performs a "Register Event" or a "Delete Event". Figure 4.10 shows an example of a conceptual view of the devices and resources store in a connected devices management component.

4.6 Communication

This research chooses CoAP to be the default communication application layer protocol in the LRSMS, due to the LRSMS being designed to run in an environment with intermittent connection. In this research, the default transfer data format used in CoAP payload is JavaScript Object Notation (JSON). The LRSMS will run as a CoAP server in the device. Every application in the device will use CoAP messages to interact with the LRSMS. Though there are four different types of CoAP messages, only comfortable message and acknowledge message will be used in the LRSMS in this research. In this research, the LRSMS will only use six types of message code in CoAP which are Get, Post, Put, Delete, Valid, and Bad Request.

4.6.1 Basic Message Exchange and Request/Response Cycle

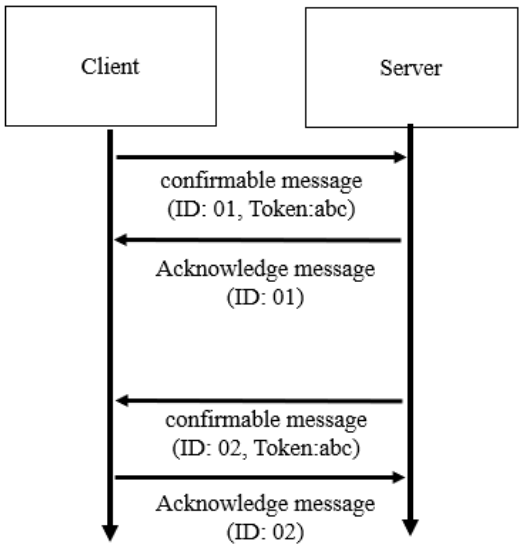


Figure 4.11 Request/Response Cycle Example

A complete message exchange cycle in CoAP is defined in this research as an acknowledge message sent from the client to the server followed by an acknowledge message sent from the server to the client to let the client know the server has received the message. A complete request/response cycle has two message exchange cycles. The first message exchange initiates from the client and is sent to the server as a request message, then the server will return an acknowledge message when it receives the message. The second message exchange is initiated from the server when the server finishes the request and sends the result to the client as a response message, then the client should return an acknowledge message when it receives the message. The same token is used in both confirmable messages cycles to associate the request and the response. Figure 4.11 shows a complete request/response cycle example.

4.6.2 Message Codes

In this research, six types of message code are used in CoAP int he LRSMS. Get, Post, Put, and Delete are used in request messages and Valid and Bad Request are used in response messages.

- The Get message code is used to ask for the resource content which is being used in the “Get Resource Event” and the “Synchronize Event”.
- The Post message code is used to create new resource information in the LRSMS and is being used in the “Register Event”.
- The Put message code is used to update LRSMS content including the resource's information stored in the Resources’ Information and Dependency component, as well as to update the resource stored in the device in the Connected Devices Management component. Put message code is being used in the “Register Event”, the “Update Event”, the “Update Alert Event”, the “Synchronize Event” and the “Delete Event”.
- The Delete message code is used to delete resource information in the LRSMS and is being used in the “Delete Event”.
- The Valid message code is used as a response message to indicate that the request succeeded.
- The Bad Request code is used as a response message to indicate that the request failed.

CHAPTER 5

PERFORMANCE EVALUATION AND DISCUSSION

The goal of this chapter is to evaluate the LRSMS system to see if it solves the problem listed in Chapter 2 as well as to test the performance of the system including propagation speed, CPU usage and memory used with different quantity of resources, applications, and devices.

The experiment can be divided into four different settings: single device with a single application, a single device with multiple applications, multiple devices with multiple applications, and multiple devices synchronization.

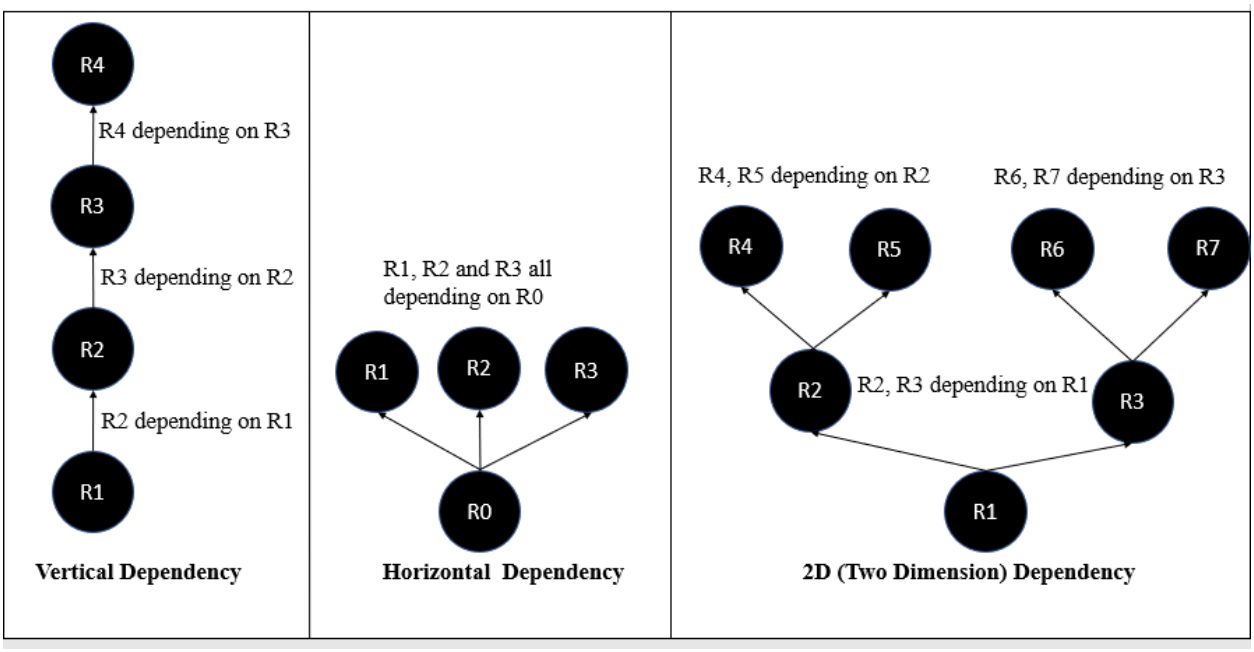


Figure 5.1 Resources Dependency Examples

Three different resource dependencies are used in experiments: vertical dependency, horizontal dependency and 2D (Two Dimension) dependency. Figure 5.1 shows an example of different resource dependencies. Vertical dependency means Resource Rn (n > 1) always depends on Rn-1. Horizontal dependency means Resource Rn (n > 1) always depends on the same resource, R0. 2D dependency means Resource R2n and Resource R2n+1 both depend on resource Rn. The three

resource dependencies will be tested individually in the first three settings (excluding multiple devices synchronization) multiple times; this encompasses the first nine evaluations in this chapter. The last evaluation, synchronize evaluation, uses a different resource dependency and device setting and will be explained later in the Section 5.4. The update time for each resource is set to be a constant and is very close to 0 in order to reduce the complexity and speed up the process of the experiment.

Table 5.1 Machine Detail

Hardware/Software	Details
Operating System	Windows 10 Education (version:20H2)
CPU	Intel(R) Core (TM) i5-7500 CPU @ 3.40GHz 3.40 GHz
Memory	RAM 16 GB
Implement Language	Golang (version: 10.3)

The experiment simulates devices and application. Details about the machine used in the experiment are shown in Table 5.1.

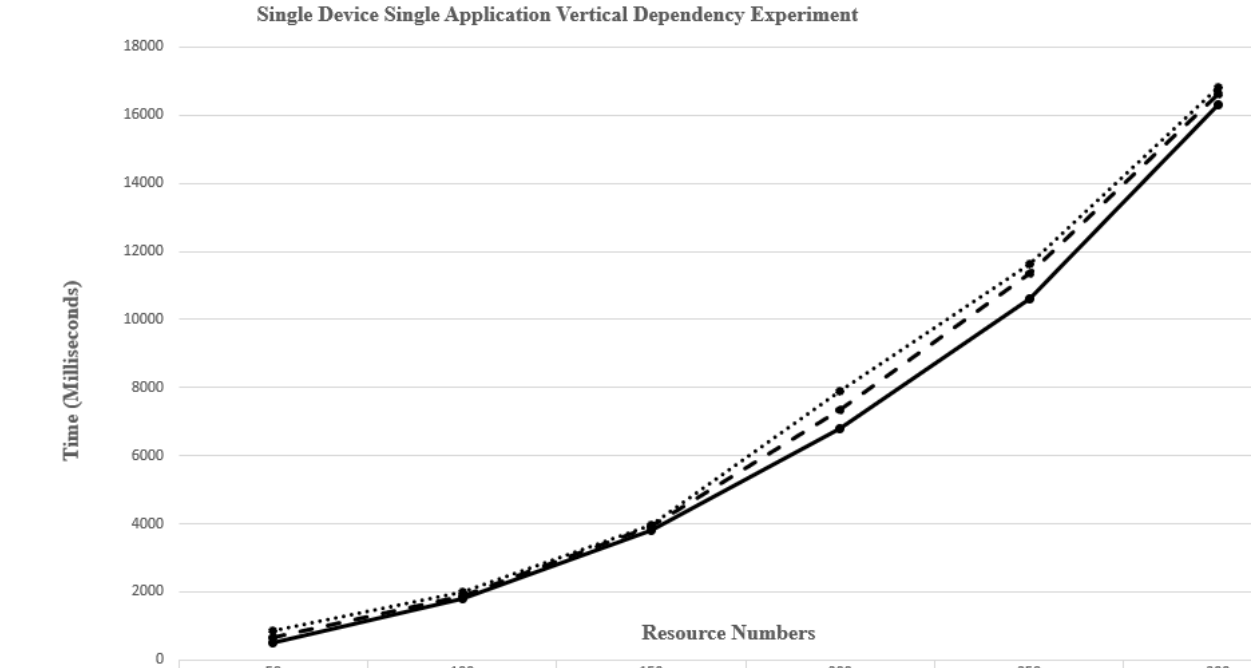
As long as the dependent resources update after the primary resource is updated in all ten evaluations, the dependencies between resources can be tracked (2.1), the system is informed when a resource state has changed (2.2), and a resource state change can be propagated to dependent resources (2.3). As long as the dependent resources update after the primary resource is updated in multiple devices evaluations and synchronize evaluation, the system can identify the same primary resource in different systems (2.4) as well as inform resources in other connected devices when a primary resource on which they depend has a new state (2.5). Lastly the synchronize evaluation shows what happened after the primary resources are reconnected (2.6).

5.1 Single Device Single Application Evaluation

In this section, all resources will be stored and used in a single application and the application will be used in a single device.

5.1.1 Single Device Single Application Vertical Dependency (SDSAVD)

In this experiment, resources are stored as vertical dependencies. The elapsed time between when the first resource updates itself and the last resource updates itself is recorded. For example, in Figure 5.1, the time needed for Resource R4 to finish updating itself after R1 was updated is recorded. This experiment tests different quantities of resources, with each run ten times. The minimum running time, maximum running time and average running time is presented for each quantity of resources used in the experiment.



	50	100	150	200	250	300
MinRunTime	493	1791	3821	6792	10619	16300
AverageRunTime	652	1893	3934	7337	11344	16626
MaxRuntime	847	1987	3981	7873	11636	16797

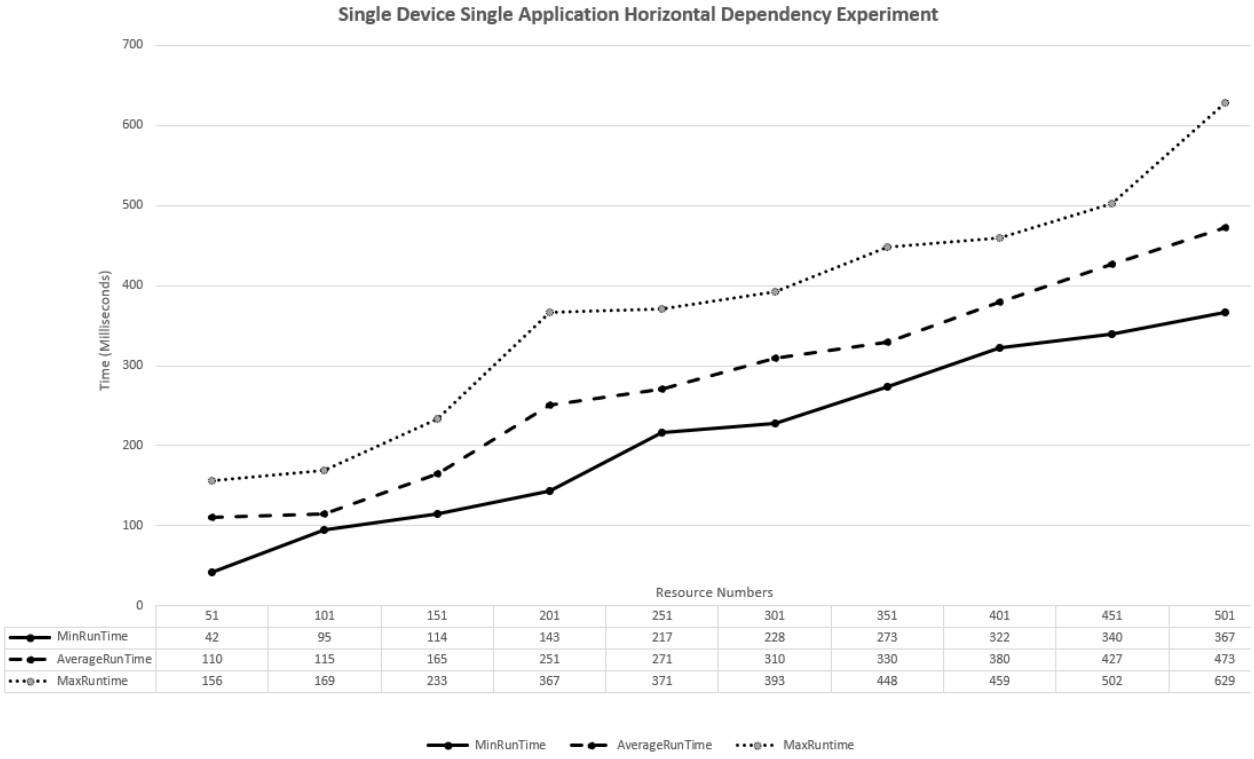
Figure 5.2 Single Device Single Application Vertical Dependency Experiment

Figure 5.2 shows the result of the Single Device Single Application Vertical Dependency Experiment. The quantities of resources used in the experiment are 50, 100, 150, 200, 250, and 300. In the column where 50 resources are used, the minimum run time for 50 resources to finish

all updates is 493 milliseconds and the maximum run time is 847 milliseconds. The maximum run time takes about 1.7 times more than the minimum run time. This means the result of the experiment is not always the same even if settings are the same and the difference can be 70% more. In the row of average run time, the number in the chart is increasing exponentially while the number of resources used is increasing linearly. This indicates that as the quantity of resources to be updated increases, the average update speed for a single resource increases as well. The average CPU usage in this experiment is 28.48 percent (including both kernel mode and user mode), and the memory used in this experiment is around 11.8 MB.

5.1.2 Single Device Single Application Horizontal Dependency (SDSAHD)

In this experiment, resources are stored as horizontal dependencies. The time it takes for all resources to update since the first resource is updated is recorded. For example, Figure 5.1 displays the time needed for all three resources (R1, R2, R3) to finish updating after R1 is updated. Each quantity of resources is being tested thirty times. The minimum running time, maximum running time and average running time is presented for each quantity of resources used in the experiment.



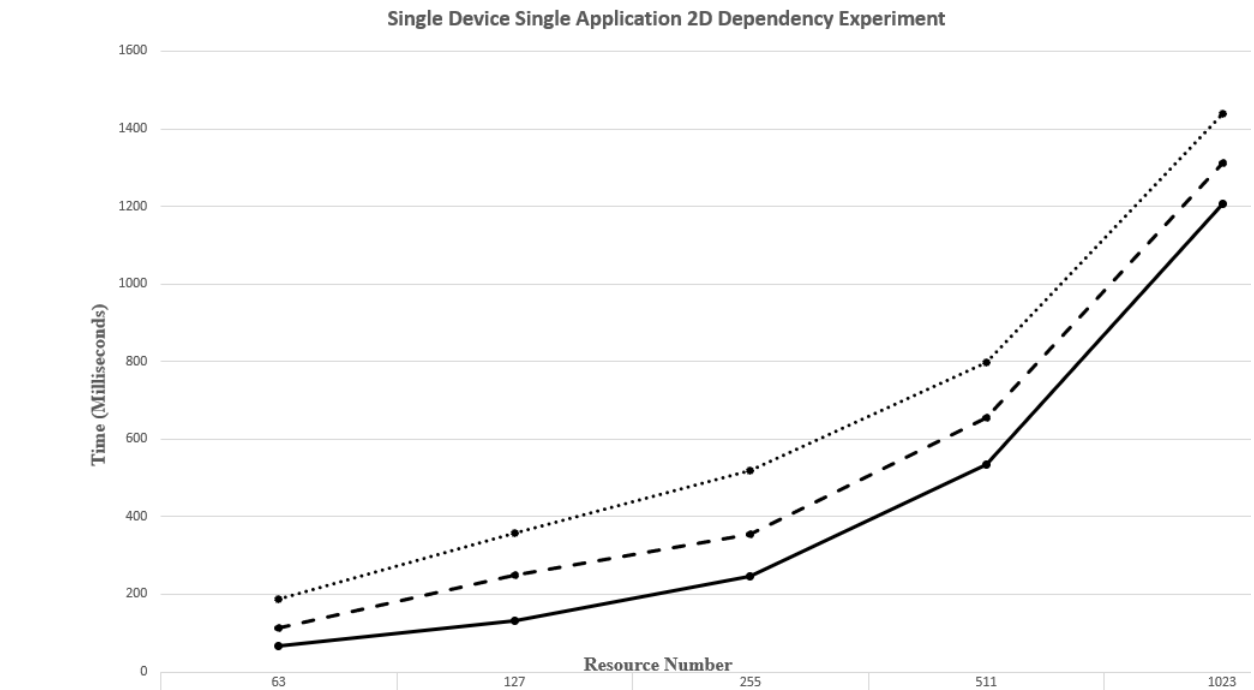
	51	101	151	201	251	301	351	401	451	501
MinRunTime	42	95	114	143	217	228	273	322	340	367
AverageRunTime	110	115	165	251	271	310	330	380	427	473
MaxRuntime	156	169	233	367	371	393	448	459	502	629

Figure 5.3 Single Device Single Application Horizontal Dependency Experiment

Figure 5.3 shows the result of the Single Device Single Application Horizontal Dependency Experiment. Ten different quantities of resources are used in the experiments: 51, 101, 151, 201, 251, 301, 351, 401, 451, and 501. In the column where 51 resources are used, the minimum run time for 51 resources to finish all updates is 42 milliseconds and the maximum run time is 156 milliseconds. The maximum run time takes about 3.7 times more than the minimum run time. This means the result of the experiment is not always the same even if settings are the same, and the difference can be 270% more. In the row of average run time, the number in the chart is increasing close to a linear speed while the number of resources used is increasing linearly. This indicates that as the quantity of resources to be updated increases, the average update speed for a single resource remains almost constant. The average CPU usage in this experiment is 23.25 percent (including both kernel mode and user mode), and the memory used in this experiment is around 11.8 MB.

5.1.3 Single Device Single Application 2D Dependency (SDSA2D)

In this experiment, resources are stored as 2D dependencies. The time taken for all resources to update after the first resource is updated is recorded. For example, Figure 5.1 displays the time needed for all six resources (R2, R3, R4, R5, R6, R7) to finish updating after R1 is updated. Each quantity of resources is run 30 times. The minimum running time, maximum running time and average running time are presented for each quantity of resources used in the experiment.



	63	127	255	511	1023
MinRunTime	66	133	246	534	1205
AverageRunTime	113	250	356	656	1312
MaxRuntime	187	359	520	797	1439

Figure 5.4 Single Device Single Application 2D Dependency Experiment

Figure 5.4 shows the result of the Single Device Single Application 2D Dependency Experiment. Five quantities of resources are used in the experiments: 63, 127, 255, 511, and 1023. In the column where 127 resources are used, the minimum run time for 127 resources to finish all updates is 133 milliseconds and the maximum run time is 359 milliseconds. The maximum run time takes about 2.7 times more than the minimum run time. This means the result of the experiment is not always the same even if settings are the same, and the difference can be 170% more. In the row of average run time, the number in the chart is increasing close to an exponential speed while the number of resources used is increasing exponentially. Both increase close to a multiple of two. This indicates that as the quantity of resources to be updated increases, the average update speed for a single resource also increases. The average CPU usage in this experiment is 47.40 percent (including both kernel mode and user mode), and the memory used in this experiment is around 16.2 MB.

5.1.4 Compare Result in Different Dependency in Single Device Single Application

Table 5.2 Different Dependency Average Update Time Per Resource in Single Applications

	50/51/63 resources	100/101/127 resources	250/251/255 resources
Vertical Dependency	Average update time 13.0ms	Average update time 18.9ms	Average update time 45.3ms
Horizontal Dependency	Average update time 2.1ms	Average update time 1.1ms	Average update time 1.1ms
2D Dependency	Average update time 1.8ms	Average update time 1.9ms	Average update time 1.4ms

Table 5.2 shows the average update time per resource (in milliseconds) using different resource dependencies. The table shows the average update speed per resource is between 1.4ms and 1.9ms for both horizontal dependency and 2D dependency, regardless of the number of resources used in the whole process. On the other hand, the average update speed per resource keeps increasing for vertical dependency when the number of total resources increases. This table also demonstrates that vertical dependency updates need significantly more time than the other two dependencies.

5.2 Single Device Multiple Applications Evaluation

In this section, every resource will be stored and used in a different application and all the applications are used in a single device.

5.2.1 Single Device Multiple Applications Vertical Dependency (SDMAHD)

In this experiment, resources are stored as vertical dependencies and each resource is stored in different applications. The time taken for the last resource to update after the first resource is updated is recorded. Different quantities of resources are tested, with each run ten times.

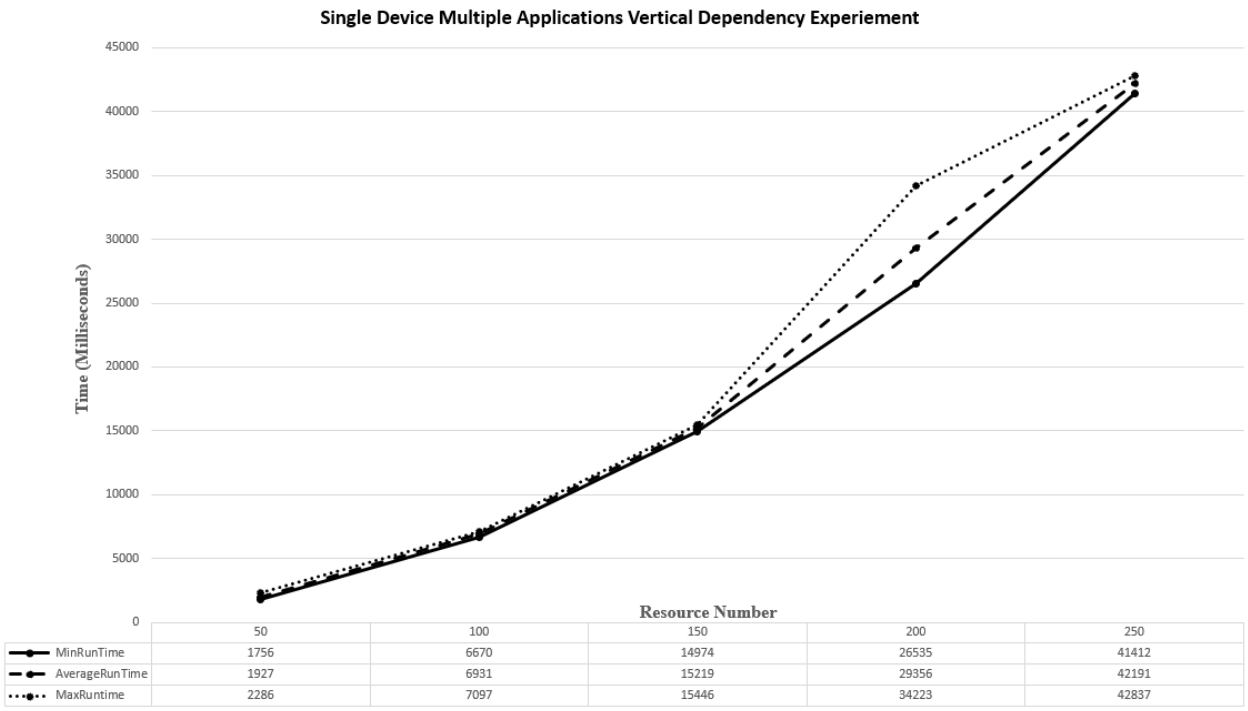
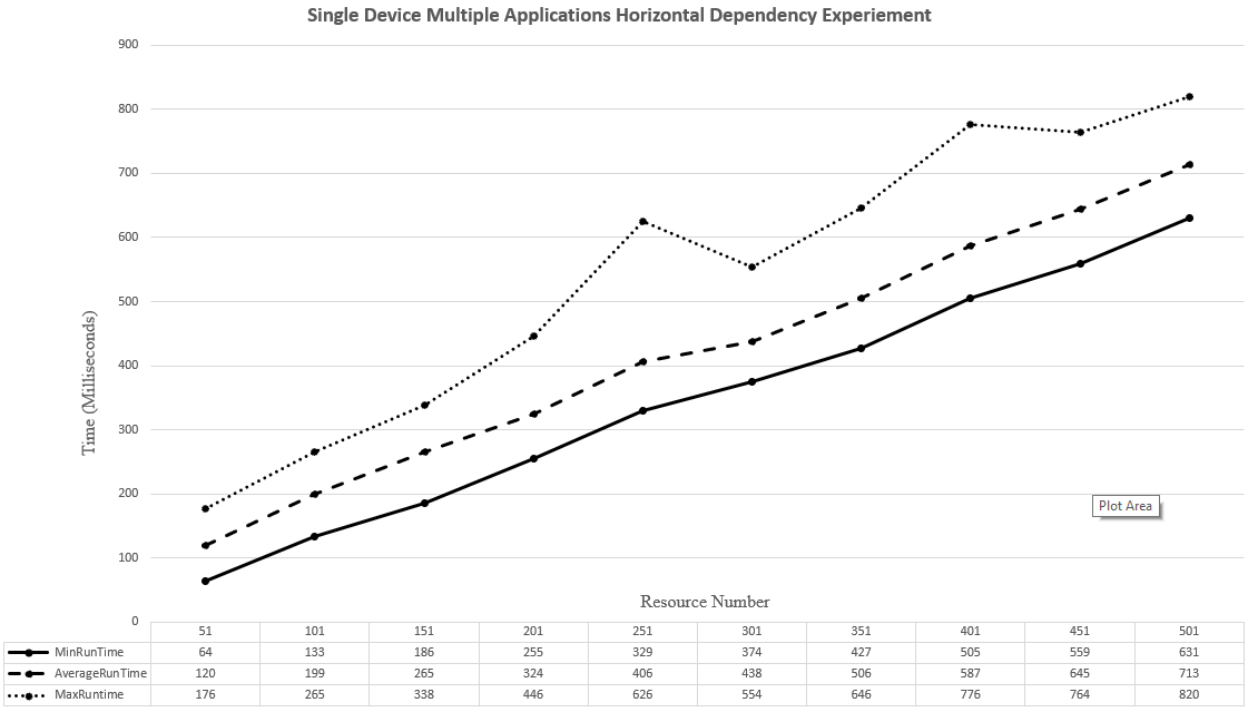


Figure 5.5 Single Device Multiple Applications Vertical Dependency Experiment

Figure 5.5 shows the result of the Single Device Multiple Application Vertical Dependency Experiment. The five resource quantities used in the experiment are 50, 100, 150, 200, and 250. In the column where 50 resources are used, the minimum run time for 50 resources to finish all updates is 1756 milliseconds and the maximum run time is 2286 milliseconds. The maximum run time is about 1.3 times more than the minimum run time. This means the result of the experiment is not always the same even if settings are the same, and the difference can be 30% more. In the row of average run time, the number in the chart is increasing faster than a linear speed while the number of resources used is increasing linearly. This indicates that as the quantity of resources to be updated increases, the average update speed for a single resource also increases. The average CPU usage in this experiment is 29.87 percent (including both kernel mode and user mode), and the memory used in this experiment is around 12.4 MB.

5.2.2 Single Device Multiple Applications Horizontal Dependency (SDMAHD)

In this experiment, resources are stored as horizontal dependency and each resource is stored in a different application. The time it takes for all resources to update after the first resource is updated is recorded. The experiment tests different quantities of resources 30 times each.



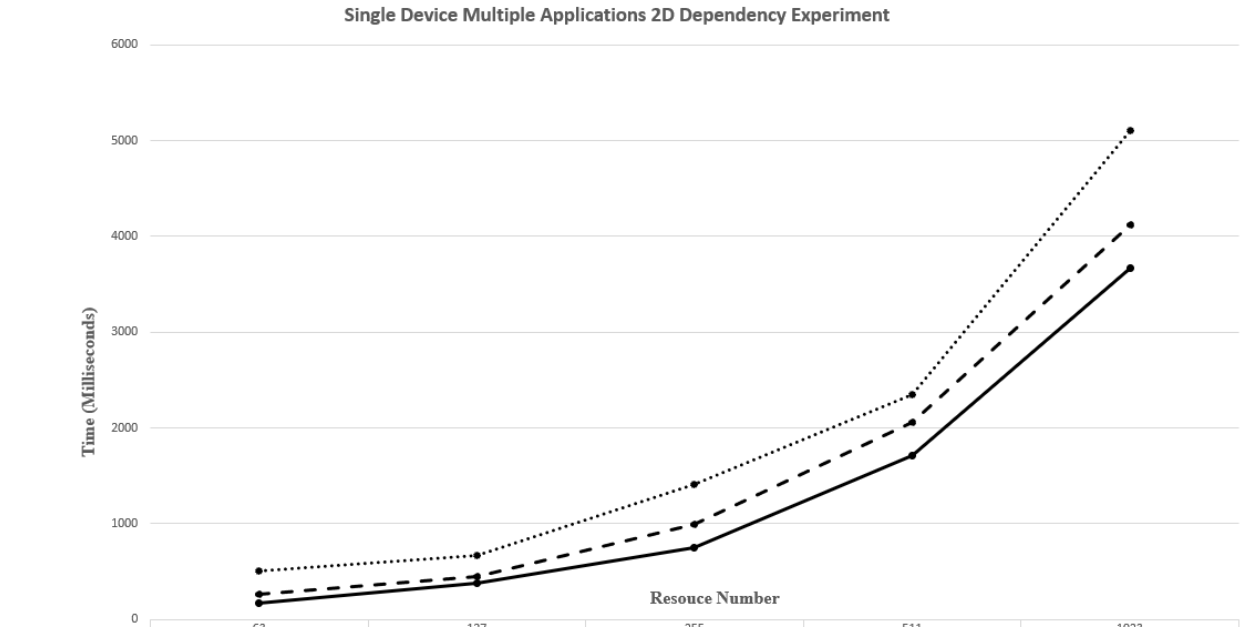
	51	101	151	201	251	301	351	401	451	501
MinRunTime	64	133	186	255	329	374	427	505	559	631
AverageRunTime	120	199	265	324	406	438	506	587	645	713
MaxRuntime	176	265	338	446	626	554	646	776	764	820

Figure 5.6 Single Device Multiple Applications Horizontal Dependency Experiment

Figure 5.6 shows the result of the Single Device Multiple Application Horizontal Dependency Experiment. The ten resource quantities used in the experiment are 51, 101, 151, 201, 251, 301, 351, 401, 451, and 501. In the column where 51 resources are used, the minimum run time for 51 resources to finish all updates is 64 milliseconds and the maximum run time is 176 milliseconds. The maximum run time takes about 2.75 times more than the minimum run time. This means the result of the experiment is not always the same even if settings are the same, and the difference can be 175% more. In the row of average run time, the number in the chart is increasing close to a linear speed while the number of resources used is increasing linearly. This indicates that as the quantity of resources to be updated increases, the average update speed for a single resource stays almost constant. The average CPU usage in this experiment is 34.35 percent (including both kernel mode and user mode), and the memory used in this experiment is around 13.0 MB.

5.2.3 Single Device Multiple Applications 2D Dependency (SDMA2D)

In this experiment, resources are stored as 2D dependency and each resource is stored in a different application. The time takes for all resources to update after the first resource is updated is recorded. Different quantities of resources are being tested in the experiment and each is run 30 times. The minimum running time, maximum running time and average running time are presented for each quantity of resources used in the experiment.



	63	127	255	511	1023
MinRunTime	167	380	745	1708	3664
AverageRunTime	256	450	989	2057	4125
MaxRuntime	499	667	1404	2350	5108

Figure 5.7 Single Device Multiple Applications 2D Dependency Experiment

Figure 5.7 shows the result of the Single Device Multiple Applications 2D Dependency Experiment. The five quantities used in the experiments are 63, 127, 255, 511, and 1023. In the column where 63 resources are used, the minimum run time for 63 resources to finish all updates is 167 milliseconds and the maximum run time is 499 milliseconds. The maximum run time takes about 3 times more than the minimum run time. This means the result of the experiment is not always the same even if settings are the same, and the difference can be 200% more. In the row of average run time, the quantity in the chart is increasing close to an exponential speed while the quantity of resources used is increasing exponentially. Both increase close to a multiple of two. This indicates that as the quantity of resources to be updated increases, the average

update speed for a single resource stays approximately the same. The average CPU usage in this experiment is not as stable as previous experiments. When the number of resources increases, the CPU usage increases as well. When 63 resources are used, the average CPU usage is around 15% and when 1023 resources are used, the average CPU usage is around 65%. The memory used in this experiment increases proportionally to the number of resources as well. When 63 resources are used, the memory usage is around 11.0MB and when 1023 resources are used, the memory usage is around 22.7MB.

5.2.4 Compare Result in Different Dependency in Single Device Multiple Applications

Table 5.3 Different Dependency Average Update Time Per Resource in Multiple Applications

	50/51/63 resources	100/101/127 resources	250/251/255 resources
Vertical Dependency	Average update time 38.5ms	Average update time 69.3ms	Average update time 168.8ms
Horizontal Dependency	Average update time 2.3ms	Average update time 2.0ms	Average update time 1.6ms
2D Dependency	Average update time 4.0ms	Average update time 3.5ms	Average update time 3.9ms

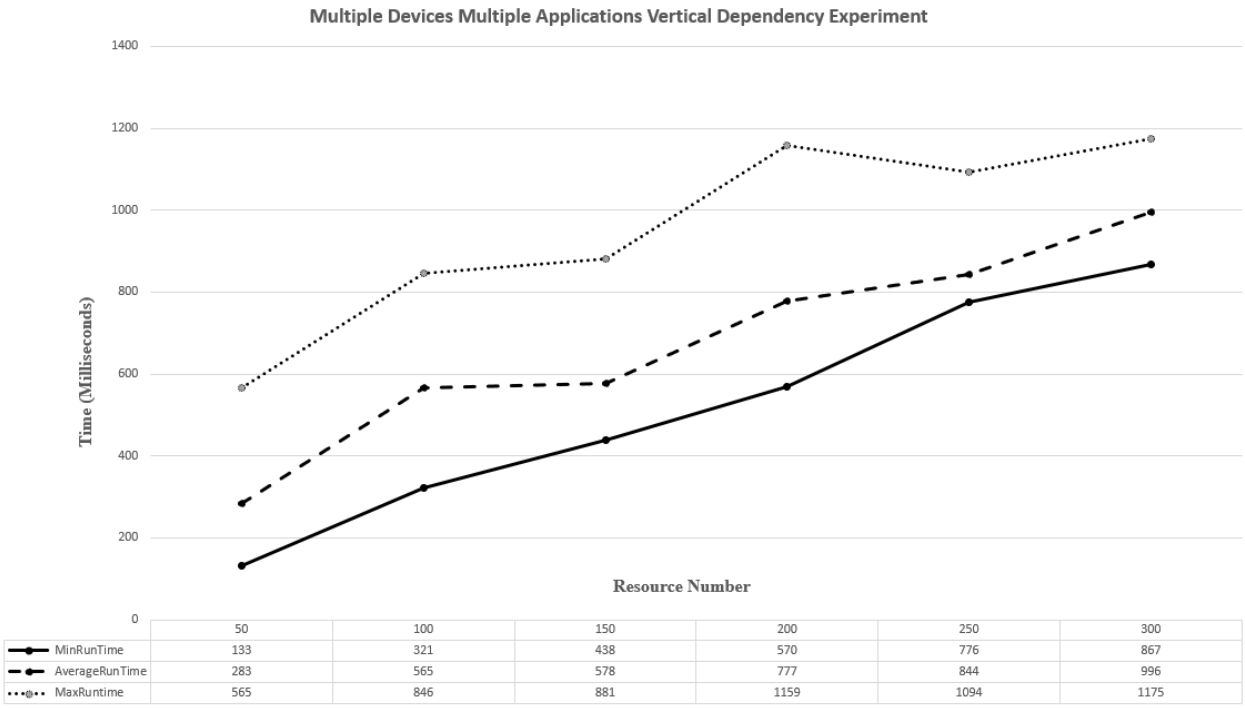
Table 5.3 shows the average update time per resource using different resource dependencies in multiple applications in a single device. The table shows the average update speed per resource is between 1.6ms and 2.3ms for horizontal dependency and 3.5ms and 4.0ms for 2D dependency. The average resource updates time is in a constant range regardless of the number of resources used in the process. On the other hand, the average update speed per resource keeps increasing for vertical dependency when the number of total resources increases. This table also demonstrates that vertical dependency updates for each resource need significantly more time than other two dependencies.

5.3 Multiple Devices Multiple Applications Evaluation

In this section, every resource will be stored and used in different applications and each application will be used in different devices.

5.3.1 Multiple Devices Multiple Applications Vertical Dependency (MDMAVD)

In this experiment, resources are stored as vertical dependencies, each resource is stored in a different application, and each application is used in a different device. The experiment records the time it takes for the last resource to update itself after the first resource is updated. Each quantity of resource is tested ten times.



	50	100	150	200	250	300
MinRunTime	133	321	438	570	776	867
AverageRunTime	283	565	578	777	844	996
MaxRuntime	565	846	881	1159	1094	1175

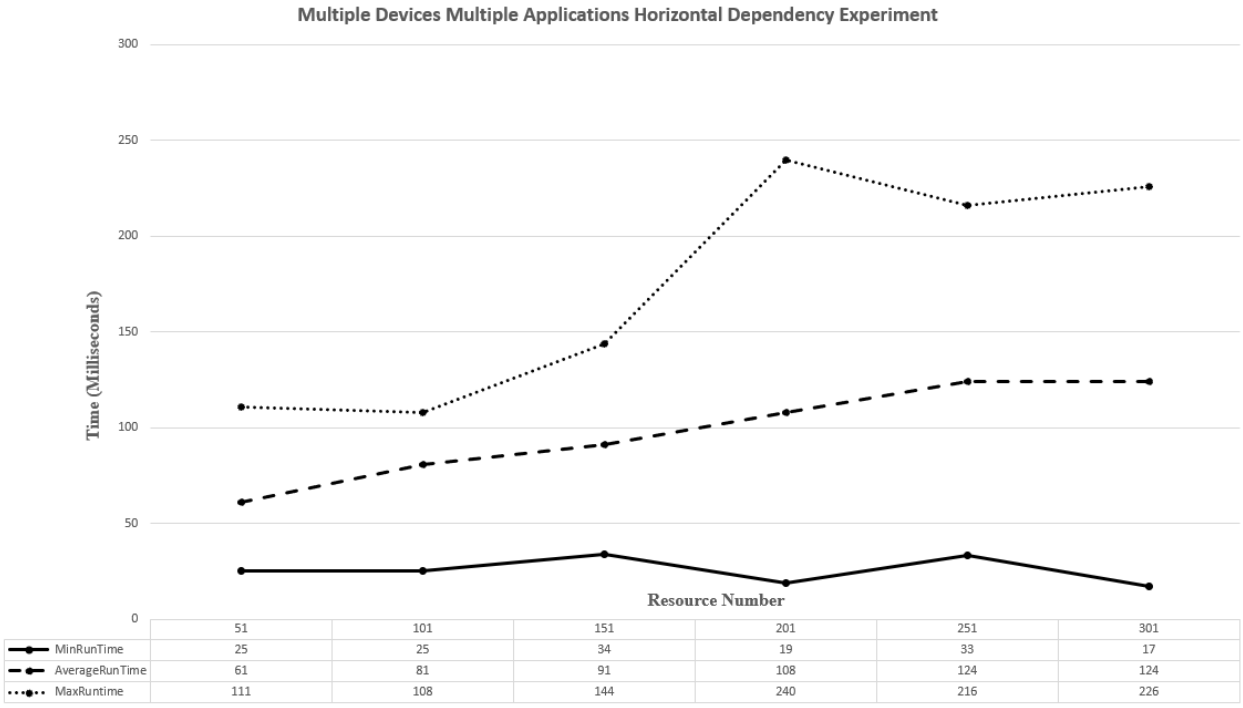
Figure 5.8 Multiple Devices Multiple Applications Vertical Dependency Experiment

Figure 5.8 shows the result of the Multiple Device Multiple Application Vertical Dependency Experiment. The six quantities used in this experiment are 50, 100, 150, 200, 250, and 300. In the column where 50 resources are used, the minimum run time for 50 resources to finish all updates is 133 milliseconds and the maximum run time is 565 milliseconds. The maximum run time takes about 4.2 times longer than the minimum rum time. This mean the result of the experiment is not

always the same even if settings are the same, and the difference can be 320% more. In the row of average run time, the number in the chart is increasing close to a linear speed while the number of resources used is increasing linearly. This indicates that as the quantity of resources to be updated increases, the average update speed for a single resource stays approximately the same. The average CPU usage in this experiment increases when the number of resources increases. When 50 resources are used, the average CPU usage is around 5% and when 300 resources are used, the average CPU usage is around 10%. The memory used in this experiment increases proportionally to the number of resources as well. When 50 resources are used, the memory usage is around 13.0MB and when 300 resources are used, the memory usage is around 136MB.

5.3.2 Multiple Devices Multiple Applications Horizontal Dependency (MDMAHD)

In this experiment, resources are stored as horizontal dependencies and each resource is stored in different applications, and each application is used in different devices. The time it takes for all resources to update after the first resource is updated is recorded. Different quantities of resources are being tested in the experiment, and each is run 30 times.



	51	101	151	201	251	301
MinRunTime	25	25	34	19	33	17
AverageRunTime	61	81	91	108	124	124
MaxRuntime	111	108	144	240	216	226

Figure 5.9 Multiple Devices Multiple Applications Horizontal Dependency Experiment

Figure 5.9 shows the result of the Multiple Devices Multiple Application Horizontal Dependency Experiment. The six quantities of resources used in the experiments are 51, 101, 151, 201, 251, and 301. In the column where 301 resources are used, the minimum run time for 301 resources to finish all updates is 17 milliseconds and the maximum run time is 226 milliseconds. The maximum run time takes about 13.3 times longer than the minimum run time. This means the result of the experiment is not always the same even if settings are the same, and the difference can be 1230% more. In the row of average run time, the number in the chart is increasing close to a linear speed while the number of resources used is increasing linearly. This indicates that as the quantity of resources to be updated increases, the average update speed for a single resource remains almost constant. The average CPU usage in this experiment increases when the number of resources is increased. When 51 resources are used, the average CPU usage is around 5% and when 300 resources are used, the average CPU usage is around 20%. The memory used is this experiment increases proportionally to the number of resources as well. When 50 resources are used, the memory usage is around 13.0MB and when 300 resources are used, the memory usage is around 916.4MB.

5.3.3 Multiple Devices Multiple Applications 2D Dependency (MDMA2D)

In this experiment, resources are stored as 2D dependencies, each resource is stored in a different application, and each application uses a different device. The time it takes for all resources to update after the first resource is updated is recorded. Different quantities of resources are being tested in the experiment and each is run 30 times. The minimum running time, maximum running time, and average running time are presented for each quantity of resources used in the experiment.

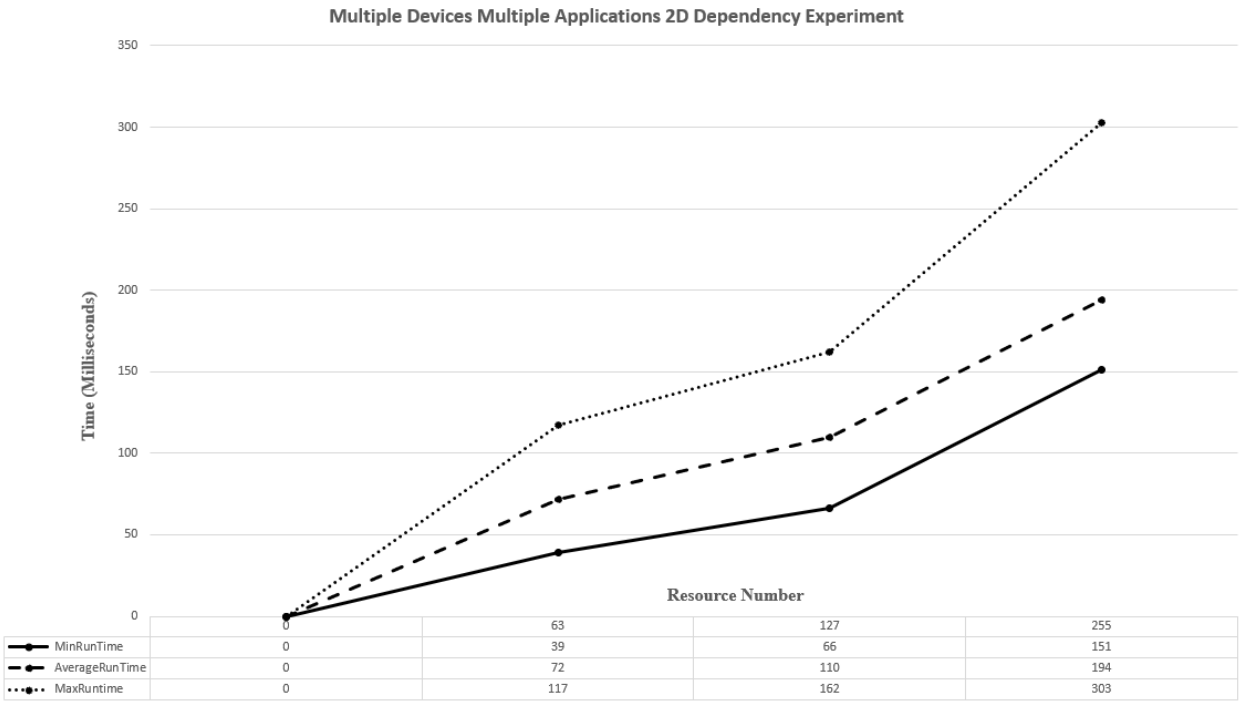


Figure 5.10 Multiple Devices Multiple Applications 2D Dependency Experiment

Figure 5.10 shows the result of the Multiple Devices Multiple Applications 2D Dependency Experiment. This experiment uses 63, 127, and 255 resources as quantities. In the column where 63 resources are used, the minimum run time for 63 resources to finish all updates is 39 milliseconds and the maximum run time is 117 milliseconds. The maximum run time takes 3 times longer than the minimum run time. This means the result of the experiment is not always the same even if settings are the same, and the difference can be 200% more. In the row of

average run time, the number in the chart is increasing close to an exponential speed while the number of resources used is increasing exponentially. Both increase close to a multiple of two. This indicates that as the quantity of resources to be updated increases, the average update speed for a single resource will stay almost constant. The average CPU usage in this experiment is not as stable as previous experiments. When the number of resources increases, the CPU usage increases as well. When 63 resources are used, the average CPU usage is around 5% and when 255 resources are used, the average CPU usage is around 15%. The memory used is this experiment increases proportionally to the number of resources. When 63 resources are used, the memory usage is around 15.0MB and when 255 resources are used, the memory usage is around 109.4MB.

5.3.4 Compare Result in Different Dependency in Multiple Devices Multiple Applications

Table 5.4 Different Dependency Average Update Time Per Resource in Multiple Devices

	50/51/63 resources	100/101/127 resources	250/251/255 resources
Vertical Dependency	Average update time 5.66ms	Average update time 5.65ms	Average update time 3.37ms
Horizontal Dependency	Average update time 1.20ms	Average update time 0.80ms	Average update time 0.49ms
2D Dependency	Average update time 1.14ms	Average update time 0.87ms	Average update time 0.76ms

Table 5.4 shows the average update time per resource using different resource dependencies in multiple applications in multiple devices. The table shows the average update speed per resource is between 0.49ms and 1.2ms for horizontal dependency, 0.76ms and 14ms for 2D dependency, and 3.37ms and 5.66ms for vertical dependency. Regardless of the number of resources used in the process, the average resource update time is in a constant range for all three dependencies. This

table also demonstrates that vertical dependency updating for each resource needs more time than other two dependencies.

5.4 Synchronize Evaluation

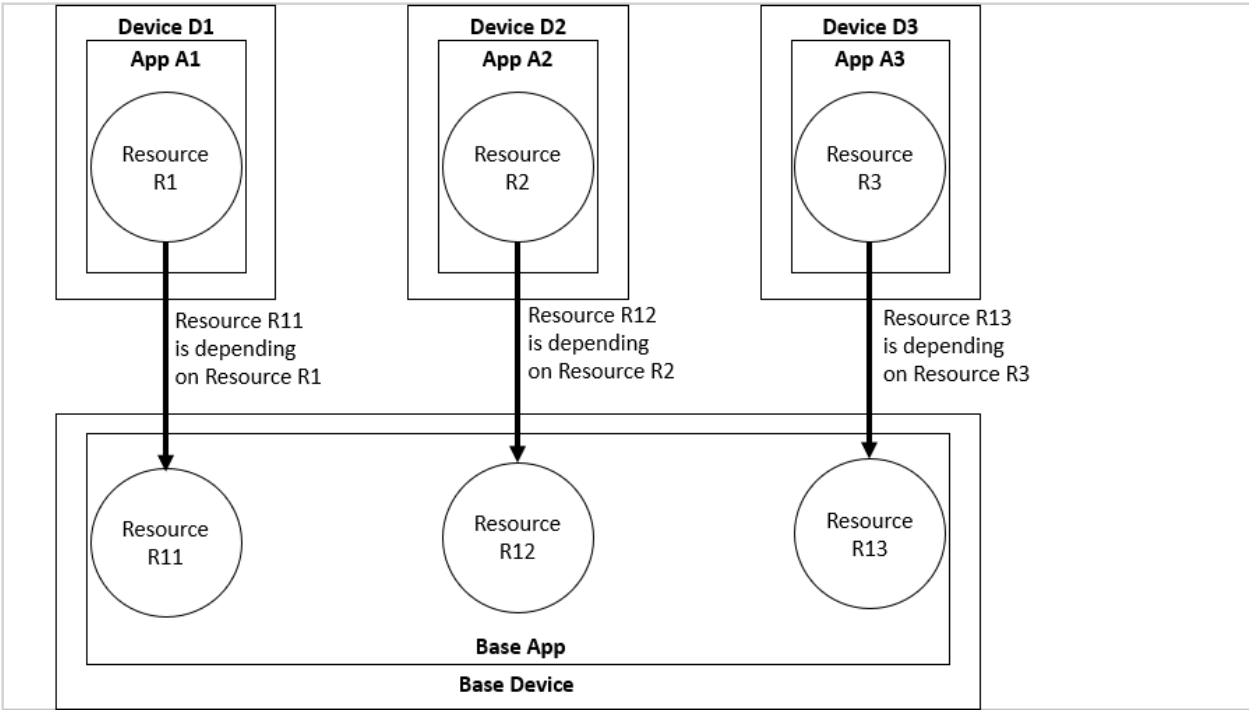


Figure 5.11 Synchronize Experiment Setting Example

In this experiment, n general resources will be stored in n different applications, and each application will be used in a different device. A base device with a base application and n base resources stored in the application will also be used. Each resource in the base application will depend on a different general resource. Figure 5.11 shows an example of the experiment settings. Three general resources, R1, R2 and R3, are stored in three different applications, A1, A2 and A3. Each application is used in a different device, D1, D2, and D3. Another three resources, R11, R12, and R13 are created, and each depends on a different general resource, R1, R2, and R3. Resources R11, R12, R13 are stored in the bass app and the bass app is used in the base device. At the beginning of the experiment, base devices are disconnected from all other devices. An update will occur in all general resources, R1, R2, and R3. After the update is finished, the base device is connected to all other devices and synchronization between the base device and other devices starts.

Different quantities of resources are being tested in the experiment and each is run 10 times. The time it takes for all resources to update after the connection, the CPU usage, and the memory used are recorded.

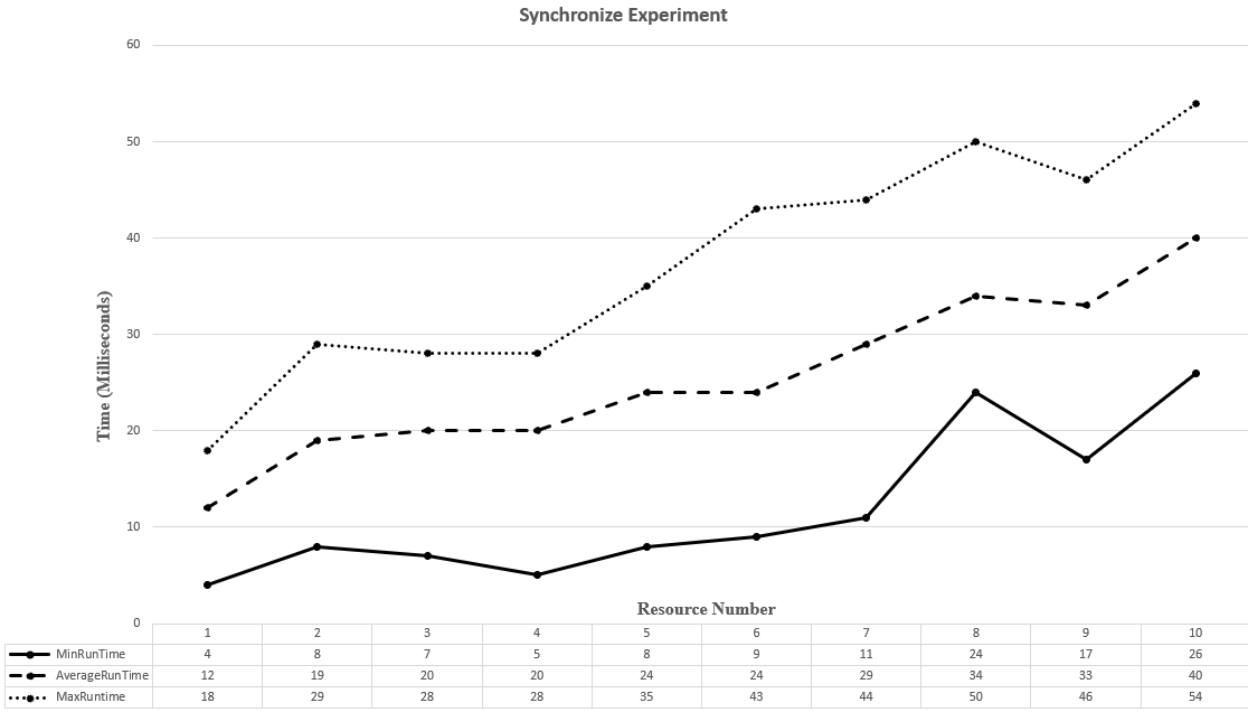


Figure 5.12 Synchronize Experiment

Figure 5.12 shows the result of the Synchronize Experiment. Ten different quantities of resources, from 1 to 10, are used in the experiments. In the column where four resources are used, the minimum run time to finish updating is 5 milliseconds and the maximum run time is 28 milliseconds. The maximum run time takes 5.6 times longer than the minimum run time. This means the result of the experiment is not always the same even if settings are the same, and the difference can be 460% more. In the row of average run time, the number in the chart is increasing close to a linear speed while the number of resources being used is increasing linearly. This indicates that as the quantity of resources to be synchronized increases, the average update speed for a single resource remains almost constant. The average CPU usage in this experiment is around 2.5 percent (including both kernel mode and user mode), and the memory used is this experiment is around 10.8 MB.

5.5 Overall Evaluation

Table 5.5 Overall Evaluation

	Experiments	Resource Average Update Speed	CPU Usage	Memory Usage
1	SDSAVD	***increase based on the number of total resources from 13ms to 45.3ms***	Relatively constant 28.48%	Relatively constant 11.8MB
2	SDSAHD	relatively constant from 1.1ms to 2.1ms	Relatively constant 23.25%	Relatively constant 11.8MB
3	SDSA2D	relatively constant from 1.4ms to 1.9ms	Relatively constant 47.40%	Relatively constant 16.2MB
4	SDMAVD	***increase based on the number of total resources from 38.5ms to 168.8ms***	Relatively constant 29.87%	Relatively constant 12.4MB
5	SDMAHD	relatively constant from 1.6ms to 2.3ms	Relatively constant 34.35%	Relatively constant 13.0MB
6	SDMA2D	relatively constant from 1.1ms to 2.1ms	***increase based on the number of total resources from 15% to 65%***	***increase based on the number of total resources from 11.0MB to 22.7MB***
7	MDMAVD	relatively constant from 1.1ms to 2.1ms	***increase based on the number of total resources from 5% to 10%***	***increase based on the number of total resources from 13.0MB to 136.0MB***
8	MDMAHD	relatively constant from 1.1ms to 2.1ms	***increase based on the number of total resources from 5% to 20%***	***increase based on the number of total resources from 13.0MB to 916.0MB***
9	MDMA2D	relatively constant from 1.1ms to 2.1ms	***increase based on the number of total resources from 5% to 15%***	***increase based on the number of total resources from 15.0MB to 109.0MB***
10	Sync	relatively constant from 1.1ms to 2.1ms	Relatively constant 2.5%	Relatively constant 10.8MB

Table 5.5 illustrates that the average update speed for each resource stays consistently between 1.1ms and 2.3ms in most of the experiments; Single Device Single Application Vertical Dependency (SDSAVD) and Single Device Multiple Applications Vertical Dependency (SDMAVD) are the exceptions. This means that regardless of the quantity of resources or applications used in the same device, if resource dependencies are vertical the update time for each resource increases when the number of total resources increases. In all three multiple device experiments, the CPU usage and memory usage increases when the total quantity of resources used increases. The CPU usage increases when multiple devices are used and multiple CoAP messages are sent simultaneously. Although this will gradually increase the processing power needed, this will also reduce the overall processing time. The memory usage increases due to fact that the quantity of devices increases.

5.6 Conclusion

The experimental results demonstrate that the system addresses the questions posed in Chapter 2. Since in all ten evaluations all of the dependent resources update after the primary resource is updated, the dependencies between resources can be tracked (2.1), the system is informed and flagged when a resource state has changed (2.2), and a change in resource state is propagated to dependence resources in the experiment (2.3). In evaluations involved multiple devices and synchronization, the dependent resources update after the primary resource, so the system can identify the same resource present in different systems (2.4) and the resources in other connected devices are informed and updated when a resource on which they depend has a new state (2.5). Lastly, in the synchronize evaluation, the resources in the general application updated, which demonstrates that a resource can be updated even when not all its primary resources are reachable (2.6).

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Data consistency has always been a challenge in the IT world. Keeping data up-to-date in an intermittent connection environment is even more difficult than a steady connection environment. This research presents an architecture to manage of data (resources) used in the system and trigger them to update themselves when one of the resources on what they depend on has a new state.

6.1 Solution for Problems

The LRSMS addresses the posed research questions to build a system that can manage the dependencies between resources (data, information, decisions) and alter the states of dependent resources when a new state is discovered in a primary resource.

1. How can the dependencies between resources be tracked?

The dependencies of a resource can be found in are source's information in the LRSMS Resources' Information and Dependency component. The supporting resources for a resource can be found in the supporting list stored in the resource information in the LRSMS. The dependent resources for a resource can be found in the dependent list stored in the resource information in the LRSMS as well.

2. How can the system be informed when a resource state has changed?

When a local resource state has changed, the application which owns that resource should notify the LRSMS and an "Update Event" should be triggered.

3. How can a resource state change be propagated to dependent resources?

The "Update Event" in the LRSMS should be able to handle the update propagation. When all the supporting resources for a resource are up-to-date or when the last supporting resource for a resource finishes its update, the "Update Event" in the LRSMS will send an update notification to the resource.

4. How can the same prime resource be identified in different devices?

If the resource ID format is different between devices, then it is almost impossible to identify the same resources. This research uses Uniform Resource Locator (URL) of the resource's source as the resource ID. When the URLs of two resources' sources are the same, the two resources' IDs will be the same, and they are considered as the same resource.

5. **How can resources in other connected devices be informed when a primary resource on which they depend a new state?**

When a resource in Device B has a new state, the LRSMS in Device B should send out an alert message to the LRSMS in Device A and trigger the "Update Alert Event" in the LRSMS of Device A. If a new device connects, "Synchronize Events" will be triggered and update all connected devices' resources as needed.

6. **How can a resource be updated when not all its primary resources are reachable?**

A cache value of the unreachable resource should have been saved in the application when the application last used it. If another resource needs to use that unreachable resource to update its content, then it should use that cache value stored in the application to update its content.

6.2 Future Work

6.2.1 Data Base

No data base is used in the system, so the information stored in the LRSMS is lost when the device shuts down. When the device restarts, applications in the device need to register the resources with the system again. Adding a data base into the LRSMS solves this problem and permanently keeps the information used in the LRSMS.

6.2.2 Connection Channels and Protocols

CoAP is the only connection protocol used in LRSMS and this limits the abilities of LRSMSs to interact with each other. Since LRSMS is designed to maintain the data consistency in an intermittent connection environment, adding more connection channels such as Bluetooth or HTTP will increase the potential to interact with different devices and improve its overall performance.

6.2.3 Security

At this point, no security mechanism is implemented in the system, which makes the system vulnerable and unreliable. Connection authentication and message encryption can be added to the system to increase the protection of the system and improve the security while exchanging messages.

6.2.4 Reliability

The system is currently unable to assess if a resource cache received from another device is reliable. A reliability mechanism could be added to the system to increase the confidence of using a new resource.

6.2.5 Experiments' Improvement in Performance Evaluation

The experiments conducted in Chapter 5 were completed in a single machine using simulated devices and applications. Using real devices with real applications installed in them could lead to a more reliable experiment. For instance, the connection latency difference between devices is being ignored in these research experiments since the experiment was run on the same machine.

REFERENCES

[1] W. Roy, A. Hopper, V. Falcao, and J. Gibbons, “The Active Badge location system,” ACM Transactions on Information Systems, vol.10, no.1, pp. 91-102, Jan. 1992. Available: http://www.parc.xerox.com/csl/members/want/papers/ab-tois-jan92.pdf

[2] B. N. Schilit and M. M. Theimer, “Disseminating active map information to mobile hosts,” IEEE Network, vol.8, no.5, pp. 22-32, Sep-Oct. 1994. Available: ftp://ftp.parc.xerox.com/pub/schilit/AMS.ps.Z

[3] J. Pascoe, N. Ryan, and D Morse, “Human-Computer-Giraffe Interaction – HCI in the field,” in *Human Computer Interaction with Mobile Devices Workshop, May 1998, Glasgow, Scotland.* Available: http://www.dcs.gla.ac.uk/~johnson/papers/mobile/HCIMD1.html#_Toc420818982

[4] A. K. Dey, “Providing Architectural Support for Building Context-Aware Applications,” Ph.D. thesis, Georgia Institute of Technology, 2000. Available: https://www.cc.gatech.edu/fce/ctk/pubs/proposal.pdf

[5] B. N. Schilit, N. Adams, and R. Want, “Context-aware computing applications,” *in the Proceedings of the 1st International Workshop on Mobile Computing Systems and Applications, IEEE Dec. 8-9, 1994, Santa Cruz, CA.* pp. 85-90. Available: ftp://ftp.parc.xerox.com/pub/schilit/wmc-94-schilit.ps

[6] J. Pascoe, “Adding generic contextual capabilities to wearable computers,” in *proceedings of the 2nd IEEE International Symposium on Wearable Computers (ISWC'98), IEEE Oct. 19-20. 1998, Pittsburgh, PA.* Available: http://www.cs.ukc.ac.uk/pubs/1998/676/content.zip

[7] M. Weiser, "The Computer of the 21st Century," ACM SIGMOBILE Mobile Computing and Communications Review, 1999 Available: https://www.lri.fr/~mbl/Stanford/CS477/papers/Weiser-SciAm.pdf

[8] TEDx Talks, John Barrett. The internet of things. (Oct 5, 2012). Accessed: Jan. 2021. [Online Video]. Available: https://www.youtube.com/watch?v=QaTIt1C5R-M.

[9] J. Anthony, “10 IoT Trends for 2020/2021: Latest Predictions According To Experts,” FinancesOnline, Boston, MA, USA. Accessed: Jan. 2021. [Online] Available: https://financesonline.com/iot-trends/

[10] D. Evans, “How the Next Evolution of the Internet Is Changing Everything” CISCO, San Jose, CA, USA, vol. 1, pp. 1–11, 2011. Available: https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf

[11] A. Nordrum, "Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated," IEEE Spectrum, New York, NY, USA, Aug 18. 2016. Accessed: Jan 2021. [Online] Available: https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated

[12] N. Feamster, "Mitigating the Increasing Risks of an Insecure Internet of Things," Princeton University, Princeton, NY, USA, Feb 18. 2017. Accessed: Jan 2021. [Online] Available: https://freedom-to-tinker.com/2017/02/18/mitigating-the-increasing-risks-of-an-insecure-internet-of-things/

[13] C. Johansen, “Cisco Survey Reveals Close to Three-Fourths of IoT Projects Are Failing,” CISCO, San Jose, CA, USA, May 13. 2017. Available: https://newsroom.cisco.com/press-release-content?articleId=1847422

[14] M, Stanojevic, S. Vranes, and D. Velasevic, “Using truth maintenance systems. A tutorial,” IEEE Expert: Intelligent Systems and Their Applications pp. 46-56, 1994. Available: https://dl.acm.org/doi/10.1109/64.363270

[15] M. N. Huhns and D. M. Bridgeland, “Multiagent Truth Maintenance,” IEEE Transactions on Systems, Man, and Cybernetics, vol. 21 no.6 pp. 1437-1445. Nov-Dec. 1991. Available: https://ieeexplore.ieee.org/document/135687

[16] A. Fox and E. Brewer, "Harvest, Yield and Scalable Tolerant Systems," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems, IEEE CS, Mar. 30, 1999.* pp. 174–178. Available: https://ieeexplore.ieee.org/abstract/document/798396/citations#citations

[17] E. Brewer, "Towards Robust Distributed Systems," in Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, Jul. 2002. Available: https://dl.acm.org/doi/10.1145/343477.343502

[18] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," ACM SIGACT News. Jun. 2002. Available: https://dl.acm.org/doi/10.1145/564585.564601

[19] E. Brewer, "CAP twelve years later: How the 'rules' have changed", Computer, vol.45, no.2. pp. 23–29. Feb. 2012. Available: https://ieeexplore.ieee.org/abstract/document/6133253

[20] M. Hadley, M Gudgin, Eds., “SOAP Version 1.2 Part 1: Messaging Framework,” W3C, WD-soap12-part1-20011002, Available: https://www.w3.org/TR/2001/WD-soap12-part1-20011002/

[21] M. Hadley, M Gudgin, Eds., “SOAP Version 1.2 Part 1: Messaging Framework,” W3C, CR-soap12-part1-20021219, Available: http://www.w3.org/TR/2002/CR-soap12-part1-20021219

[22] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. thesis, University of California, Irvine, 2000. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[23] R. Fielding, Eds., “Hypertext Transfer Protocol -- HTTP/1.1,” IETF, RFC 2616, Jun. 1999. Available: https://tools.ietf.org/html/rfc 2616

[24] D. Gourley, B. Totty, M. Sayer, A. Aggarwal, and S.Reddy. “HTTP: The Definitive Guide,” O,REILLY. Available: https://dzone.com/articles/coap-protocol-step-by-step-guide

[25] Z. Shelby, K. Hartke, and Carsten Bormann, “The constrained application protocol (CoAP),”

IETF, RFC 7252, Jun. 2014. Available: https://tools.ietf.org/html/rfc7252

[26] F. Azzola. “CoAP Protocol: Step-by-Step Guide,” DZone. Available: https://www.oreilly.com/library/view/http-the-definitive/1565925092/ch01s05.html