Integrating Game Engines into the Mobile Cloud as Micro-services

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
Qi Liu

©Qi Liu, March 2018. All rights reserved.

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 or 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 of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5C9
Canada

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

Abstract

Game engines have been widely adopted in fields other than games, such as data visualization and game-based education. As the number of mobile devices owned by each person increases, extra resources are available in personal device clouds, expanding typical learning space to outside of the classroom and increasing possibilities for teacher-student interactions.

Owning multiple devices poses the problem of how to make use of idle resources on devices that are slightly dated or lack portability compared to newer models. Such resources include CPU power, display, and data storage.

In order to solve this problem, an architecture is proposed for mobile applications to access these resources on various mobile devices. The main approach used here is to divide an application into several modules and distribute them over a personal device cloud (formed by same-user-owned devices) as micro-services. In this architecture, game engines will be incorporated as a render module to tap in its rendering capability. Additionally, modules will communicate using CoAP which has minimal overhead.

To evaluate the feasibility of such architecture, a prototype is implemented and deployed over a mobile device, and tested in a modest context that is similar to real life settings.

Acknowledgements

I would like to express my gratitude to my two supervisors Dr. Ralph Deters and Dr. Li Chen, for all their guidance, patience, and support throughout the duration of this research.

I would also like to thank the members of my defence committee, Dr. Julita Vassileva, Dr. Gordon McCalla, and Dr. Chris Zhang, for sharing their insight and feedback. Their guidance is greatly appreciated.

Contents

List of Tables

List of Figures

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
BLE	Bluetooth Low Energy
CoAP	Constrained Application Protocol
CoRE	Constrained RESTful Environments
CRUD	Create, Read, Update, and Delete
DTLS	Datagram Transport Layer Security
FPS	Frame Rate Per Second
GIS	Geographic Information System
GUI	Graphical User Interface
HLAPI	High-Level API
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
LLAPI	Low-Level API
mGBL	mobile Game-Based Learning
MCC	Mobile Cloud Computing
MMORPG	Massively Multiplayer Online Role-playing Game
MSA	Micro service Architecture
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVP	Model-View-ViewModel
NPC	Non-player Character
OS	Operating System
PC	Personal Computer
REST	Representational State Transfer
RPG	Role-playing Game
SIG	Bluetooth Special Interest Group
SOA	Service-orientation Architecture
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VR	Virtual Reality
WAP	Wireless Access Point
WPAN	Wireless Personal Area Network
XML	Extensible Markup Language

Chapter 1
Introduction

Implementing mobile applications face problems such as constrained resources, limited energy, and inconsistent network connections [9]. With the constant improvements in high-bandwidth wireless networking and processing power in mobile devices, and the development of 3d graphics APIs such as OpenGL ES for these devices [18], the field of mobile visualization was created and is considered to have great potential [48].

In recent years, the application of game engines in scientific visualization and Geographical Information Systems (GIS) has been explored [23] and is gaining popularity, because of the drastic cost reduction compared to traditional scientific visualization softwares, and the significantly lower requirement for hardware [14]. Modern commercial game engines not only provide users with intuitive tools for developing 3d interactive experiences, but also offer seamless multi-platform shipment. Moreover, mobile Game-Based Learning (mGBL) has gained much momentum. Mobile applications tend to offer simpler interaction and interface compared to PCs, which benefits users of all age groups. Additionally, various built-in sensors (e.g., Accelerometer, GPS) provide more ways to interact with app contents to bring users better immersive experiences [21].

It is currently common for an individual to own more than one mobile device, whether they are smart phones or tablets. as shown in Figure 1.1, the average numbers of devices owned per user until February 2016 was over 3 [4], suggesting there are idle computational resources and storage in a mobile-device-formed device cloud. If an application can be divided into modules and deployed over the device cloud, extra hardware and software resources could be used.

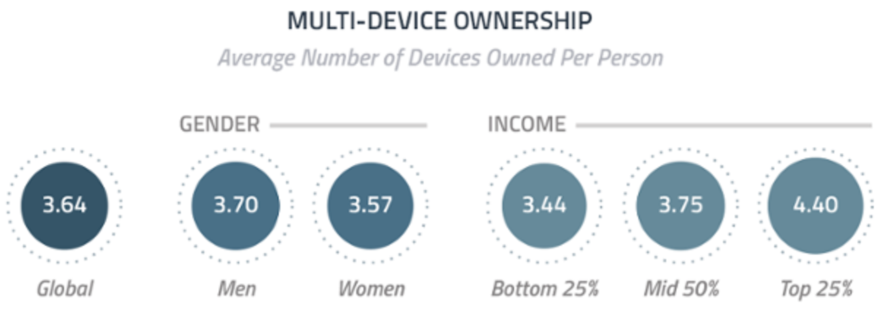


Figure 1.1: Average Number of Devices Owned Per Person [4]

The remainder of this paper is structured as follows: Chapter 2 further explains the research problem, followed by related works in Chapter 3; Chapter 4 discusses the architecture of the proposed solution; and

Chapter 5 focuses on the evaluation plan. Finally, a conclusion is given in Chapter 6.

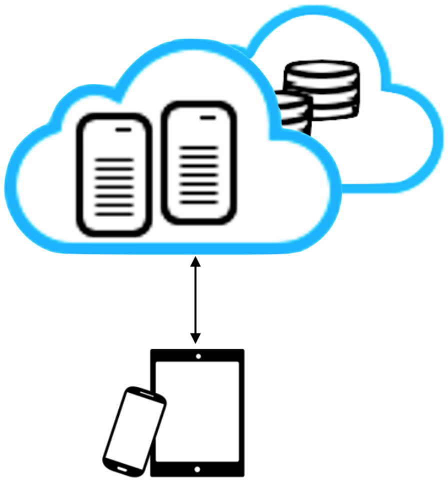


Figure 1.2: Mobile Cloud Computing

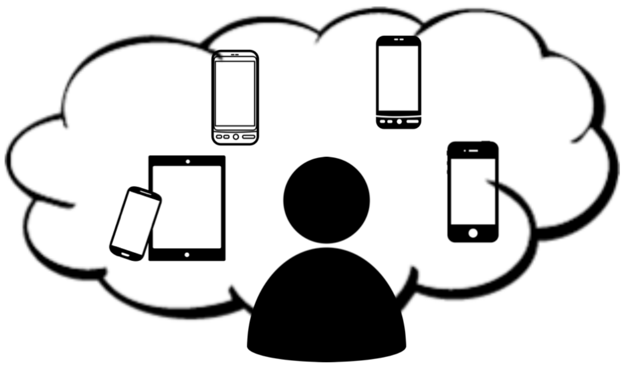


Figure 1.3: Cloud of User Devices

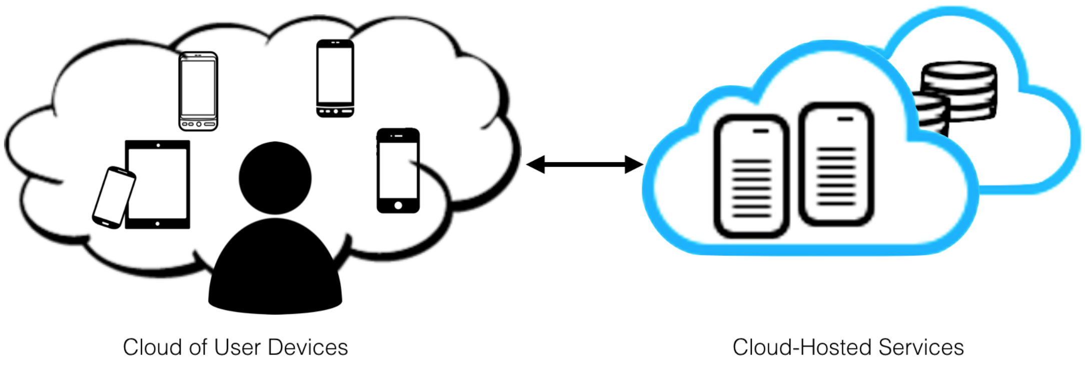


Figure 1.4: Combing Device Cloud with Cloud Services

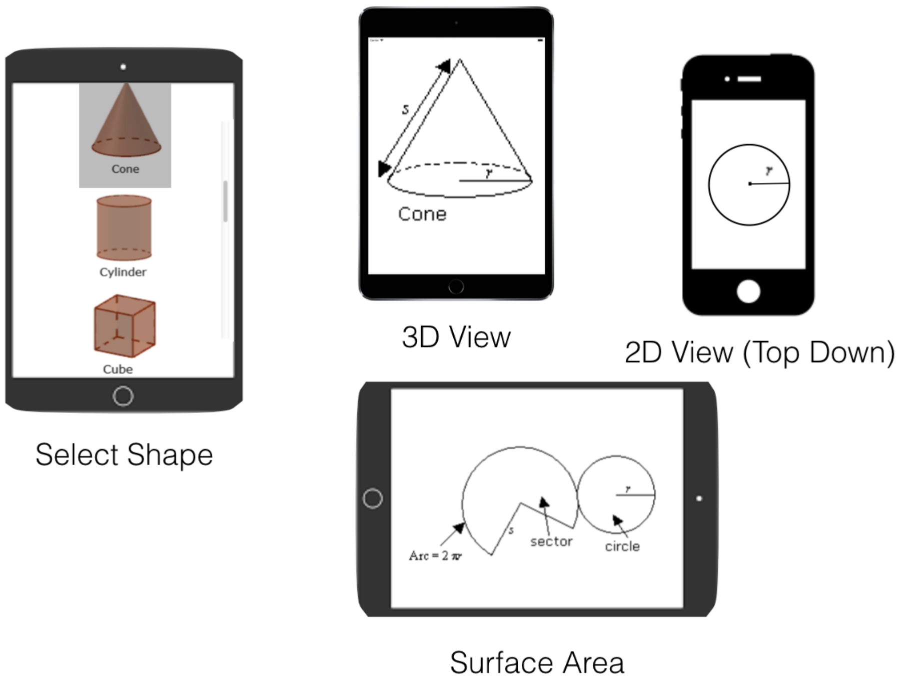


Figure 1.5: Example of a Solid Geometry Education App in a Personal Device Cloud

Chapter 2

Problem Statement

It is common for a person to own multiple mobile devices which form a personal device cloud [20]. Among them, older devices are often not used. Some of them may have big screens or powerful CPUs that could be turned into controllers for other applications, as shown in Figure 2.1. In order to access these idle resource, the question of how to divide applications into components, and deploy them over a device cloud, and work coordinately arises.

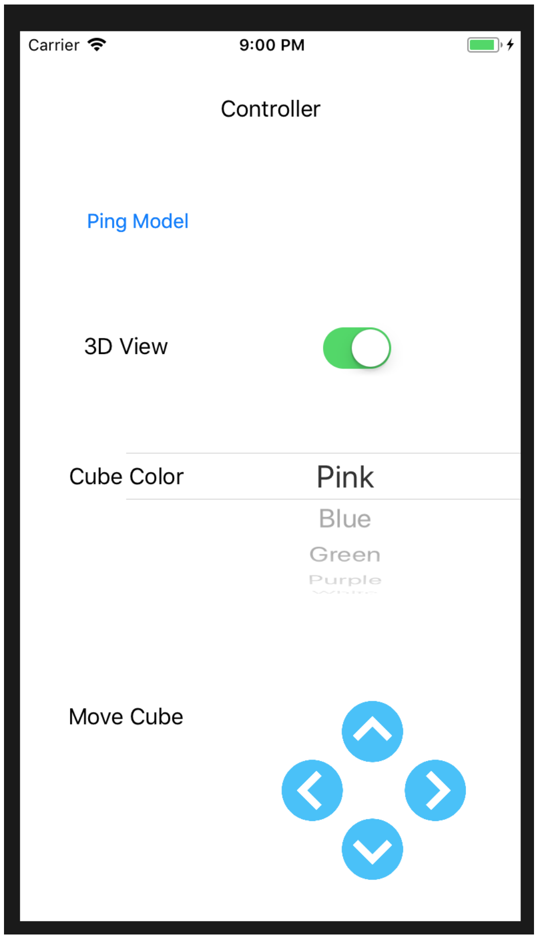


Figure 2.1: Screen of Controller App

Commercial game engines are gaining momentum as tools for building serious applications, especially in visualization, because of their ability to achieve reasonable frame rates and to require less hardware.

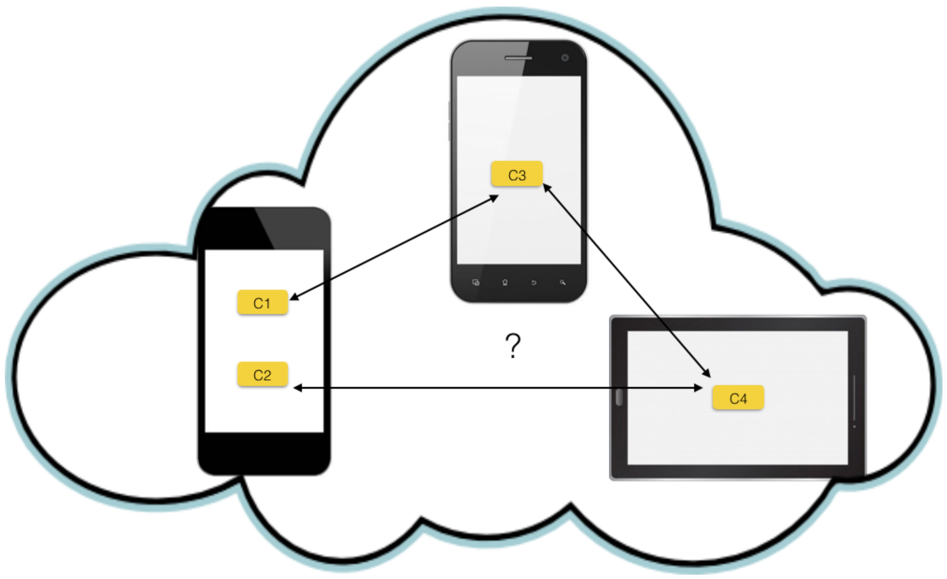


Figure 2.2: Communication Between Distributed Components on Different Mobile Devices

In summary, the main research problems are:

1. How can components of an application be spread over multiple mobile devices?
2. How can distributed components coordinate with each other?
3. How effective are the distributed components working as one application?

To solve such problems, the main research goals are:

1. To propose an architecture that integrates game engines into a collection of distributed applications inside a personal mobile device cloud as micro-services;
2. To implement network technologies that facilitate communication between micro services; and
3. To evaluate the efficiency of such architecture by implementing a prototype and conduct evaluations on the performance under different scenarios.

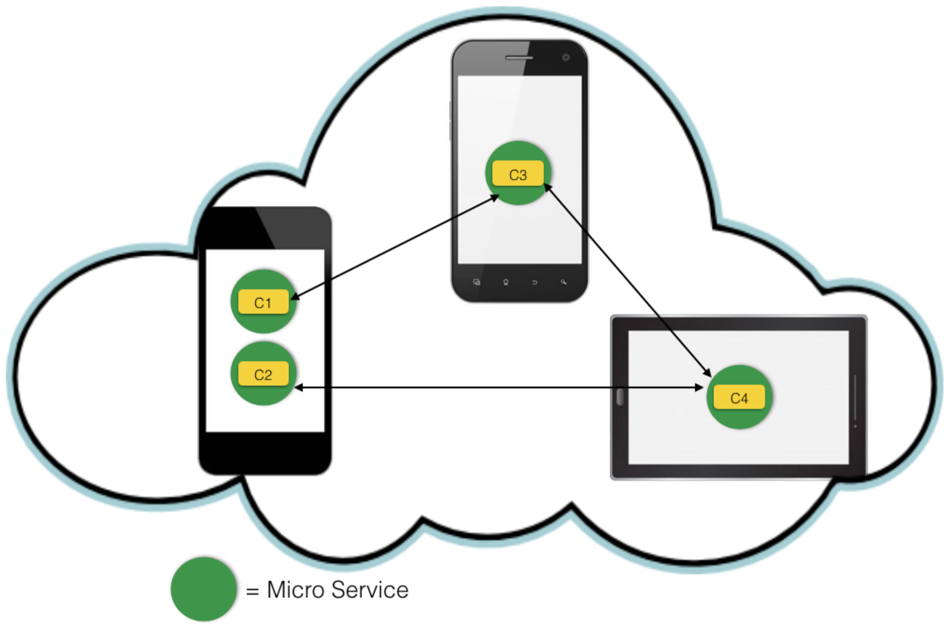


Figure 2.3: Components as Micro-Services

Chapter 3

Related Work

In order to find a solution to the proposed problem, related studies are reviewed in this chapter. Game engines and their application in scientific and collaborative environments is introduced in section 3.1, followed by a comparison of popular software design patterns in section 3.2. In section 3.3, the architecture style, Representational State Transfer (REST) is reviewed, followed by a review of Constrained Application Protocol (CoAP) and a comparison with Hypertext Transfer Protocol (HTTP) in section 3.4. Lastly, applicable network technologies for communication inside device clouds are reviewed in section 3.5.

3.1 Game Engines

Due to the popularity of computer games, a considerable amount of research has been devoted towards the development of game engines. Modern game engines now come with whole sets of tools for developing interactive experiences. As a result, the use of game engines in the building of non-gaming applications has become increasingly popular [49] [45]. The following sections provide a overview of game engines, and the their applications in serious gaming.

3.1.1 Introduction

A game engine is a software framework designed for the creation and development of video games for consoles, mobile devices and personal computers.

Since game systems are complicated, developing a game from the ground up can be expensive and slow. Given training with the latest technologies, the rapidly changing market, and the development of high-end technologies, a game engine is often needed for the development of a modern game. Game engines can cater to building games of different genres, in the same way different models of cars can be built using the same model of engines [44].

Game engines contain core components including rendering engine, physics engine, GUI, and network so that game developers do not have to develop these features every time they implement a game idea. From handling low-level graphical optimizations to guaranteeing reasonable Frame Rate Per Second (FPS), to importing common asset formats, game engines essentially do the difficult work of game development so developers can focus on the visual atmosphere, story, and other factors that are important to creating a successful game.

One of the most important features of a game engine is platform abstraction, which guarantees that games are able to ship to multiple platforms with a minimal change of source code.

3.1.2 Unity 3d

Unity [42] is an engine and framework for both game and application development. It has been gaining momentum because of the easily understood interface, detailed documentation, and compatibility with major platforms. It is one of the most extensively used game engines for mobile game development and has an active development community [43].

The game engine supports importing assets from major 3d applications such as 3ds Max [2], Maya [3], Cinema 4D [15], the open source tool Blender [11], and others. Its wide usage is a result of the broad range of asset formats that it supports [30].

The Unity engine is C++ based, and scripting in Unity uses either C# or JavaScript. The user-generated code runs on Mono version 3.5 or the Microsoft .NET Framework 3.5. Unity allows testing of a game in development without the need to export or build. Debugging is done with MonoDevelop [47] which can be launched within Unity.

Workflow within Unity adopted the classic Object-oriented design, where every entity in game is an object; but it also enforces a component-based architecture [41], where properties are defined by components and a game object is merely a container of different combinations of components. Each component is a self-contained function that performs a specific task. Components can be built-in components or generated from user-created scripts. Component-based architecture allows functions in different domains (e.g., physics, rendering, sound, and AI), and to achieve decoupling of code while increasing reusability of code. Some components work the best in combination with others; for example, rigid body and collider components enable collision detection against other game objects. In addition, components can be easily swapped in and out of game objects during live edits which makes testing considerably easier. The main drawback is that components add another level of indirection that turn a game object into a cluster of components, each of which needs to be instantiated, initialized, and wired.

Multi-thread is supported in Unity script, while Unity APIs, because they are not thread-safe, can only be called from the main thread. When expensive or long-term operations are being computed in Unity, threads can still be useful (for example, AI, pathfinding, network communication, and file operations). One factor to be noted is that synchronization between threads might be more expensive than computing data in the main thread, and it is necessary to do performance tests. It has also been reported in the developer community that behaviour of multi-thread games can be unpredictable or can run inconsistently running on different platforms. Unity also supports the use of coroutines [40] in its main thread. Coroutine methods can be executed piece by piece over time, but all processes are still done in the single main thread, with the result that if a coroutine attempts to execute a time consuming operation, the application may still freeze.

3.1.3 Serious Games

Game engines have been long applied in the development of non-games or “serious games”. The concept of serious gaming was first introduced in 1970 in the book Serious Games [1] by Clark Abt. In the book, his references were primarily to the use of board and card games. Mike Zyda gave an update explaination to the term in 2005 [49]: “Serious game: a mental contest, played with a computer in accordance with specific rules, that uses entertainment to further government or corporate training, education, health, public policy, and strategic communication objectives.”

Since then, game engines have become popular tools in architecture visualization, Geographic Information System (GIS), military training, and scientific simulation. According to [14], game engines favour real-time rendering over physical correctness and data accuracy compared to professional scientific simulations or visualization softwares. The price for the final product is very low compared to professional softwares. As a result, game engines are a popular choice when data accuracy is not crucial .

Moloney and Harvey [26] proposed a collaborative virtual environment developed based on a game engine for architectural design education. It allows asynchronous collaboration, utilizing real time communication which is not possible when using typical architectural visualization software. They also stated that architectural visualization has benefited from the improvement in Virtual Reality (VR) technologies and how game engines are explored as feature-rich but low-cost alternatives to high-end virtual reality software.

Craighead [8] proposed a Distributed Tutoring Framework that uses Match Server provided by a game engine. The framework is composed of three elements: an immersive game, a master server to store player information and handle multi-player collaboration, and an intelligent tutoring agent.

In conclusion, game engines have been applied extensively in scientific visualization. Many studies have been conducted on the application in distributed systems, relying on networking solutions that game engines provide. A few distributed game engine architectures are proposed, but are not designed to work in a mobile environment, and do not rely on existing game engines to take advantage of their advanced rendering capability.

3.2 Design Patterns

In order to divide functionalities of a distributed application and assign them to different modules, several design patterns, including the Model-View-Controller (MVC) and its related patterns, are reviewed in this section. In addition, to facilitate coordination between modules, the Micro-Service Architecture (MSA) is explored.

3.2.1 MVC

Syromiatnikov and Weyns [38] discussed the difference between existing MV* patterns classified in three main families. They stated that all MV* patterns are based on the idea of separation of concerns, and came to the conclusion that MVC patterns are the leading patterns for synchronizing user interfaces with domain data.

Below is a brief introduction of the Model-View-Controller (MVC) architecture which is based on the idea of separation of presentation. It focuses on clear division of domain objects and presentation, as well as the Model-View-View Model (MVVM) pattern and Model-View Presenter pattern (MVP), developed with the intention of eliminating the disadvantages of MVC.

The MVC pattern is one of the first attempts at serious UI work. The idea of separated presentation is the foundation of MVC, and is the most influential on later frameworks [12]. It depicts a clear separation of domain objects that are modelled after the real world, and a presentation that is the GUI elements on the screen [33]. Thus the Model in MVC, is responsible for maintaining domain data and logic is completely ignorant of the UI. For each element on the screen, one View-Controller pair is assigned, the Controller handles user input, updates domain data, and triggers domain logic inside the Model, and the View updates its data display by observing the Model.

An issue that MVC fails to address is the fact that the view logic does not fit into the domain logic on domain objects, where the Model should be responsible only for managing domain data and logic. This can be addressed by the adoption of a Presentation Model in the MVVM pattern.

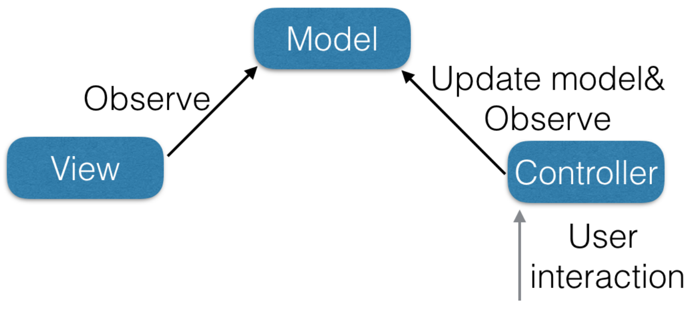


Figure 3.1: The Model-View-Controller pattern

3.2.2 MVVM and MVP

The MVVM pattern solves the problem that MVC has by using a Presentation Model (or View Model) that wraps around the domain model and handles the view state by providing extra properties and logic. The View-Controller pair in MVC are combined into a single View component which observes and operates on the Presentation Model and updates the View accordingly, without direct reference to the domain model. In this pattern, domain data is handled by the Model; view state and user interaction is handled by the View Model; and rendering of user interface and display of data is handled by the View. The MVP architecture is inspired by both Forms and Controls and MVC, where the Forms and Controls blend reusable widgets with an application specific code, and MVC has a separated presentation and domain model [38].

There are two main types of MVPs: the Supervising Controller pattern and the Passive view pattern. MVP is similar to traditional MVC in the sense that the Presenter is similar to a looser form of the Controller. But the Presenter, unlike the Controller, is responsible for changing the view as well.

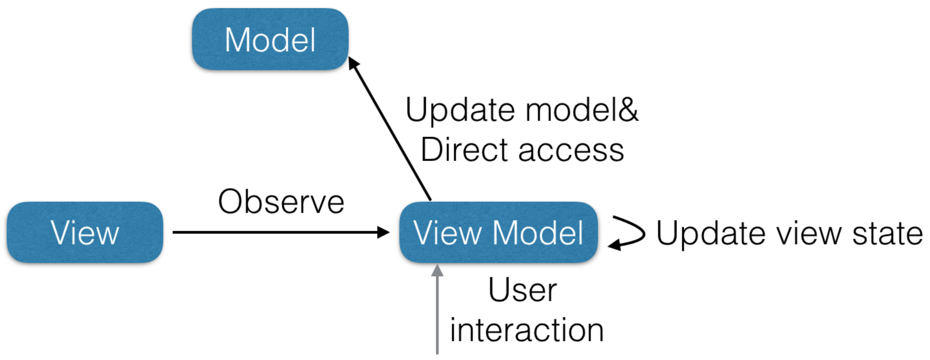


Figure 3.2: MVVM pattern

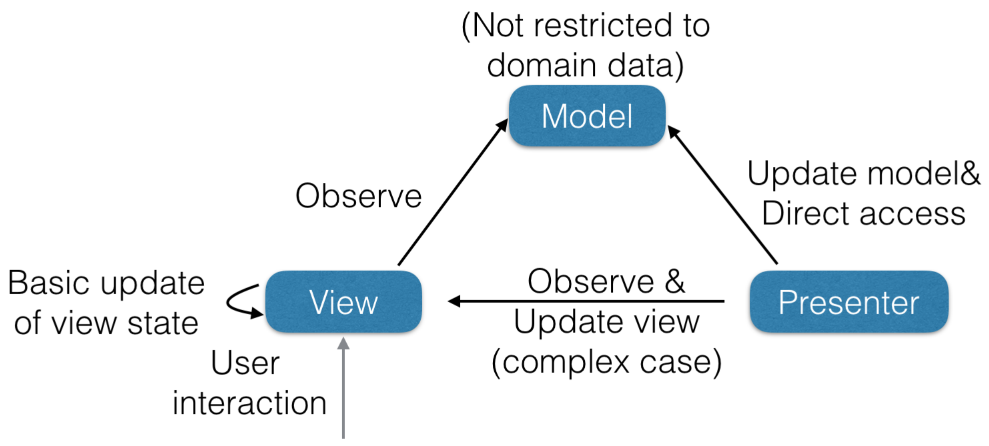


Figure 3.3: Supervising Control Pattern

3.2.3 Micro-Service Architecture

Micro-Service Architecture (MSA) is a software architecture style in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs [13]. These services are small, highly decoupled, and focus on doing a small task [28], facilitating a modular approach to

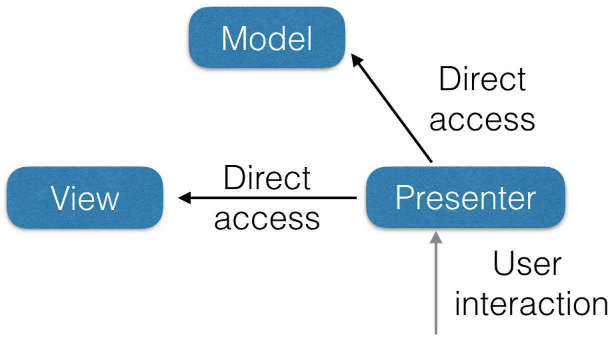


Figure 3.4: Passive View Pattern

system-building. MSA is distinct from a Service-Orientated Architecture (SOA) in that the latter aims at integrating various applications whereas several micro-services belong to one application only.

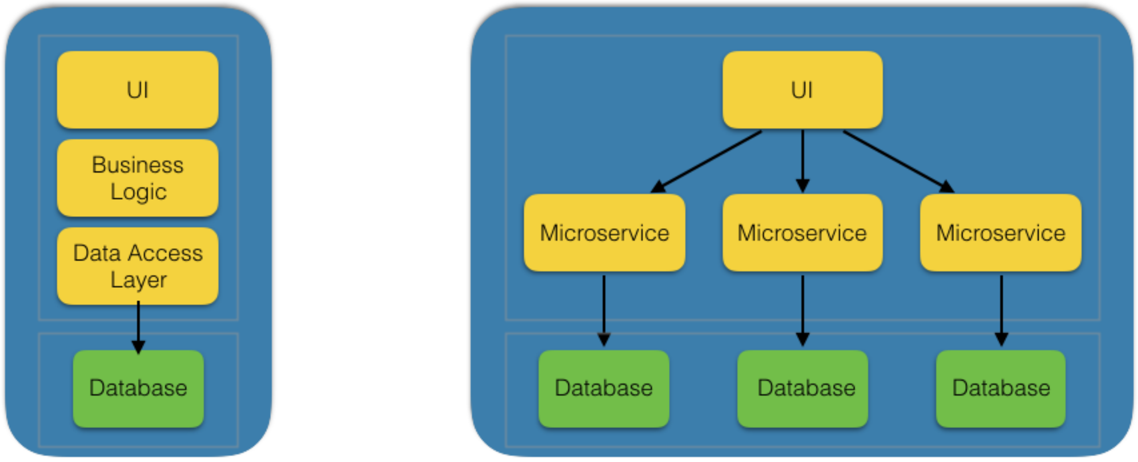


Figure 3.5: Monolithic Architecture and Micro-Service Architecture

Software built as micro-services is broken down into multiple component services, so that each can be independently maintained or updated without affecting the entire application. [22] compared MSA to the Monolithic Architectural style: A modification made to a section of an application following the Monolithic Architectural style may require building and deploying of the entire application, whereas in micro-services, developers might only need to change one component instead the entire application.

The code below is an example of a rudimentary micro-service by [32] to respond to a query written in Go [17]. The function of the code is to start a server on port 8080 on all interfaces. When a user connects to a corresponding URL in the format "http://localhost:8080/user_name," he or she will receive a welcome

```go

package main

import (
    "encoding/json"
    "fmt"
    "net/http"
)

// Resturns a string of text "Welcome, user_name!"
// to a connection through "http://localhost:8080/user_name".
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Welcome, \%!", r.URL.Path[1:])
}

// Returns a lengthy introduction of the service, and returns error code during a fault
// connection.
func about (w http.ResponseWriter, r *http.Request) {
    m := "Lengthy introduction of this service."

    // Returns an error code with logging when a fault connection or request is made.
    b, err := json.Marshal(m)

    if err != nil {
        panic(err)
    }

     w.Write(b)

// Direct all traffic coming into port 8080 to according handlers.
func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/about/", about)
    http.ListenAndServe(":8080", nil)
}
```

message as a response.

In conclusion, separating an application into MVC modules, and building each module to expose a suite of independent services following MSA pattern, can help build a distributed application that is loosely coupled, providing highly usable services.

3.3 RESTful Web Services

RESTful Web Services have been used extensively in mobile device environment. In this section, the REST Architecture and its use in mobile applications is reviewed.

REST (Representational State Transfer) is a lightweight architecture style for designing networked application proposed by Fielding [10]. RESTful Web Services are web applications built upon the REST architecture. According to the architecture, each available resource on the server is identified by a Uniform Resource Identifier (URI). A client of RESTful Web Services can request different operations (Create, Read, Update, and Delete) on resources through standard HTTP verbs(POST, GET, PUT, and DELETE). HTTP GET, for instance, is defined as a data-producing method that is intended to be used by a client application to retrieve a resource, to fetch data from a Web server, or to execute a query with the expectation that the Web server will look for and respond with a set of matching resources [31]. Christensen [7] states that RESTful Web Services are suitable for mobile device environments because them are easy to invoke, produce a discretely formatted response, can usually be easily parsed, and are less memory intensive.

Pautasso et al. [29] describes the advantages of the REST Architecture style for its lightweightness, and, because of the adoption of URIs and hyperlinks, resource discovery does not require registration to a central repository. REST is also easily scalable and the support for lightweight message formats can further optimize the performance of Web Services.

To conclude, the combination of RESTful Web Services with micro-services to expose specific resources of distributed modules would meet the requirements of this research, by having little overhead while providing flexibility and scalability.

3.4 HTTP and CoAP

HTTP and CoAP are both based on the REST model and can be used as communication protocols to expose RESTful Web Services in a mobile device cloud. In the following section, CoAP will be reviewed and compared with HTTP.

CoAP [35] is a RESTful web transfer protocol optimized for communication between resource-constrained networks and nodes. CoAP uses the User Datagram Protocol (UDP) as its transport protocol unlike HTTP which operates on top of the reliable Transmission Control Protocol (TCP) and can be too complex for constrained environments. CoAP is not a blind compressed version of HTTP, but a subset of REST common, with support of URI and HTTP verbs, that gear towards machine-to-machine applications, hence it is an effective protocol for micro-services hosted on mobile devices.

```
+----------------------+
|      Application     |
+----------------------+
+----------------------+
|  Requests/Responses  |
|----------------------|  CoAP
|       Messages       |
+----------------------+
+----------------------+
|          UDP         |
+----------------------+
```

Figure 3.6: Abstract Layering of CoAP [19]

According to [35], CoAP has the following main features:

- A web protocol fulfilling M2M requirements in constrained environments.
- UDP binding with optional reliability supporting unicast and multicast requests.
- Asynchronous message exchanges.

- Low header overhead and parsing complexity.
- URI and Content-type support which is similar to HTTP.
- Simple proxy and caching capabilities.
- A stateless HTTP mapping, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way, or for HTTP simple interfaces to be realized as an alternative to CoAP.
- Security binding to Datagram Transport Layer Security (DTLS).

Since the network environment in a mobile device cloud is not absolute, devices can be connected to the internet through Wi-Fi or connected to each other through an ad hoc network via low-range radio technologies. Ideally HTTP can be used with an unconstrained network and CoAP with a constrained network. However, some technologies, for example Bluetooth, does not rely on IP for addressing, and IP is essential in the existing implementations, except for [6]. A communication protocol should always be decided according to the underlying network environment, which is briefly introduced in the next section.

3.5 Network Technology

3.5.1 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is an emerging wireless technology developed by the Bluetooth Special Interest Group (SIG) for short-range communication. BLE has distinctive features not found in Bluetooth 4.0 [16]. BLE is ideal for applications requiring episodic or periodic transfer of small amounts of data. Therefore, Bluetooth Low Energy is especially well suited for sensors, actuators, and other small devices that require extremely low power consumption [36], for example, wearable health devices, home automation, and advertising. BLE has the following key features:

- Works well with high numbers of communication nodes with limited latency requirements.
- Very low power consumption.
- Robustness equal to Classic Bluetooth.
- Short wake-up and connection times.
- Good smartphone and tablet support.

3.5.2 Wi-Fi Direct

The IEEE 802.11 standard [37] has become one of the most common ways to access the Internet for over adecade. Wi-Fi Direct is a technology defined by the Wi-Fi Alliance, aimed at enhancing direct device-to-device communications in Wi-Fi. Traditional Wi-Fi requires a Wireless Access Point (WAP) in order for a device to connect to a wired network. Devices in the same Wi-Fi network can only communicate through the WAP. Since many of the smaller devices today support Wi-Fi, it has become increasingly necessary for peripheral devices to form networks without the presence of a WAP.

A major novelty of Wi-Fi Direct, according to [5], is that these roles are specified as dynamic, and hence, a Wi-Fi Direct device has to implement both the role of a client and the role of an AP (sometimes referred to as Soft-AP). One device that supports Wi-Fi Direct can take the role of an access point and connects to other devices that supports a Wi-Fi connection, forminf an ad hoc network. The possible downfall of the present implementations is that different platforms have their own implementations and may not be interconnectable.

To sum up, the network technologies, Wi-Fi, and BLE, and Wi-Fi Direct are all feasible choices for the connection of mobile devices inside a device cloud.

3.6 Summary

In summary, in the Related Work section, game engines and their applications in scientific and collaborative environments are reviewed, showing that game engines are applicable tools for working with other modules in a visualization role. After a brief review of MV* patterns and Mirco-service Architecture, the MVP pattern will be adopted to used as a guide for module partition. However, for convenience of expression, the module responsible for control logic will still be called the Controller. The RESTful Web Services is chosen because of its lightweightness. As for the communication protocol, both CoAP and HTTP are viable choices, but the messaging mechanism has to be implemented differently according to the network settings (for example, Wi-Fi or low range radio technologies, such as BLE and Wi-Fi Direct).

Chapter 4

Architecture

As modern mobile devices have been growing in screen resolution, CPU, battery life, and memory space, many constraints on application development no longer exist, especially for games and visualization applications that are computationally demanding because of constant frame updates. Since a stable frame rate and high resolution are guaranteed, a considerable number of mobile versions of AAA game titles have been developed. As a result, mobile platforms have become a standard for both casual and more "hardcore" games, and for visualization tools in general.

Moreover, the number of mobile devices an average person owns is growing rapidly, with the inevitable overlap of functionality, many devices are not put into frequent use, providing abundant idle resources.

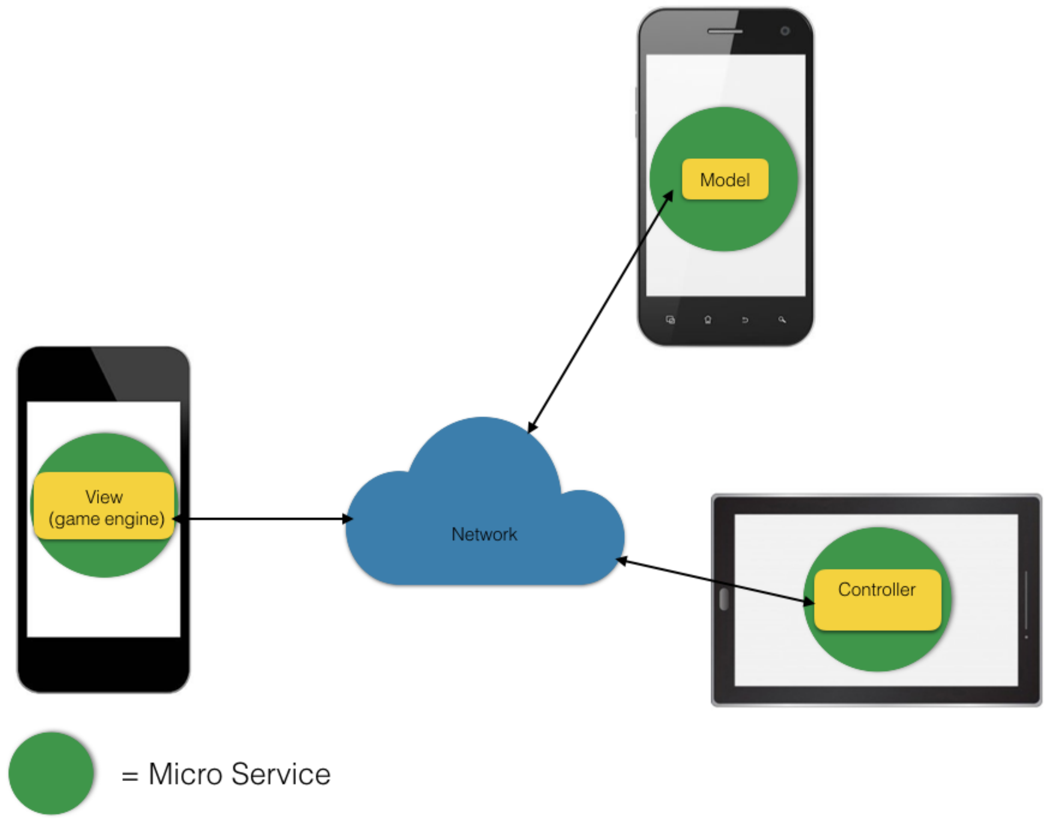


Figure 4.1: Overview of Proposed Architecture

The main objective of this research is to propose an architecture for distributing mobile games or visualization apps onto different mobile devices as modules, and working coordinately to make use of idle resources, such as computation, display, and data storage.

As is shown in the Related Work section, most of the previous research relies on the built-in multi-player networking solutions of chosen game engines to facilitate the communication of distributed game instances between different game players. This approach does not address the need for distribution and communication of modules for single-player games, which is the most common genre on mobile platforms. In addition, these solutions are well encapsulated and offer no freedom in the modularization of game instances, such that there is no way to facilitate the coordination of more finely grained game modules.

The proposed architecture encapsulates different business logics of a game or visualization application as modules, distributes them over several mobile devices, and exposes their functions as micro-services for each other to consume. The design also allows the devices to communicate, using low-overhead protocols over a wireless network, which is a common setting for both homes and offices. This architecture is flexible so that the number of participating mobile devices and players can vary, and by modifying the number and function of distributed services, it can cater to a different number of devices and both single- or multi-player scenarios.

The pattern of distributed MVC is adopted for implementing the distributed modules. Note that in this implementation each module does not strictly contain the code for only one role. As mobile applications, they each have their own UI and control logic to bind the UI to their domain data. This holds especially for the View module which is implemented using a game engine, that encapsulates complex controls, for the included render, sound, AI engines, and other features.

There are three main modules inside the architecture: the View, Controller and Model, as shown in Figure 4.1. The View is responsible for rendering the game screen, Controller gathers user input and feeds requests to the Model, and finally, the Model stores and synchronizes game data between the Controller and the View. Following the MVC design pattern, the Controller sends update requests to the Model, the Model updates data accordingly, and then the View updates its rendering of data by observing the Model.

When a user starts a Controller application, the Controller first checks if the other services are alive and accessible in the same network. If they are (their instances have been started on other devices), the user can start to manipulate game data as the Controller UI prompts. To hold the game state and data for one session of the game, one Model has to exit, but multiple instances of the View can be present.

The modules communicate with each other through a network that can be either Wi-Fi or WPAN (Wireless Personal Area Network). In the context of Wi-Fi communication, either CoAP or HTTP can be applied as the transfer protocol, with CoAP being a lightweight alternative to HTTP for use in constrained networks and for working on top of UDP. Both provide the HTTP methods PUT, GET, POST, and DELETE to map to the REST CRUD services. If the modules are connected in a WPAN via low-power radio technologies, such as BLE or Wi-Fi Direct, modules of a distributed application may not be able to communicate if they are deployed onto different platforms since there appears to be no multi-platform solution.

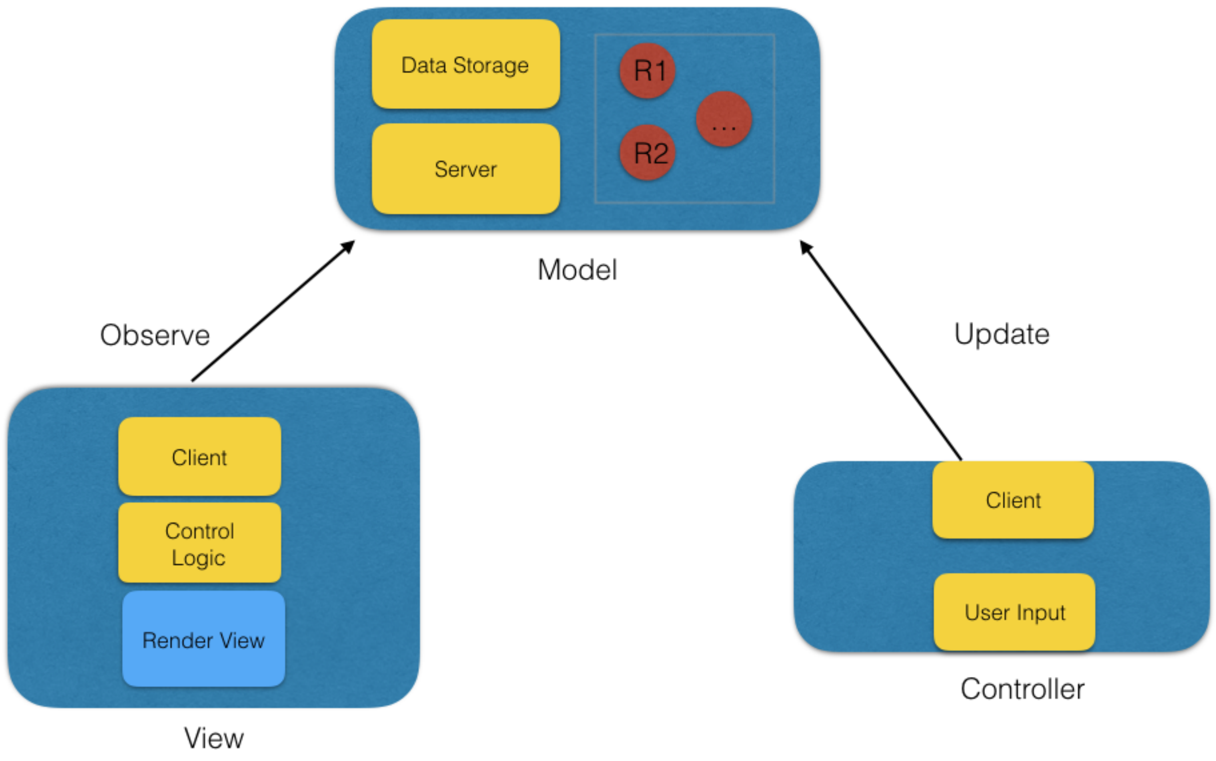


Figure 4.2: Overview of Distributed Components in the Architecture

For the purpose of demonstration of the proposed architecture, a primitive level of the game Sokoban is implemented as an example of the application. Sokoban is a type of transport puzzle that was first created in Japan. In the original game the player pushes boxes or crates around in a warehouse, trying to get them to storage locations. When implemented as a video game, it is played on a board of squares, where each square is a floor or a wall. Some floor squares contain boxes and others are marked as storage locations (i.e., "targets"). The player is confined to the board, and may move horizontally or vertically onto empty squares, but never through walls or boxes. The player can also move into a box, which pushes it into the square beyond. Boxes may not be pushed into other boxes or walls, and they cannot be pulled. The number of boxes is equal to the number of storage locations. The puzzle is solved when all boxes are at storage locations and the player is at the wining spot. The targets are colour coded and should be easy to identify from the regular board.

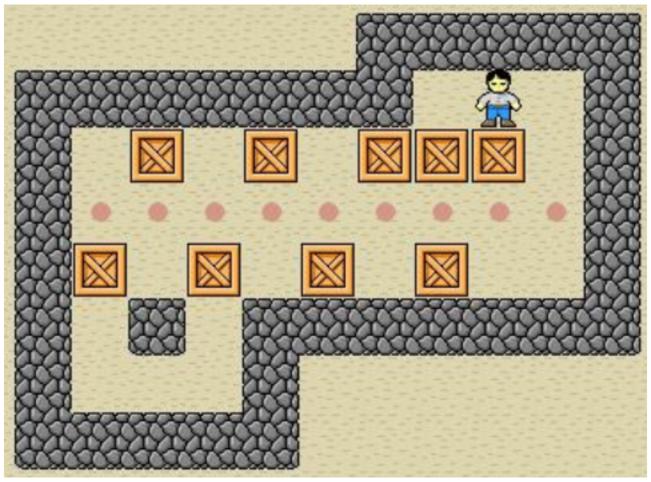


Figure 4.3: Level #7-2 From "Monster Sokoban" [27]

In the following section, the implementation of each moduleas well as communication setups are explained.

4.1 Implementation

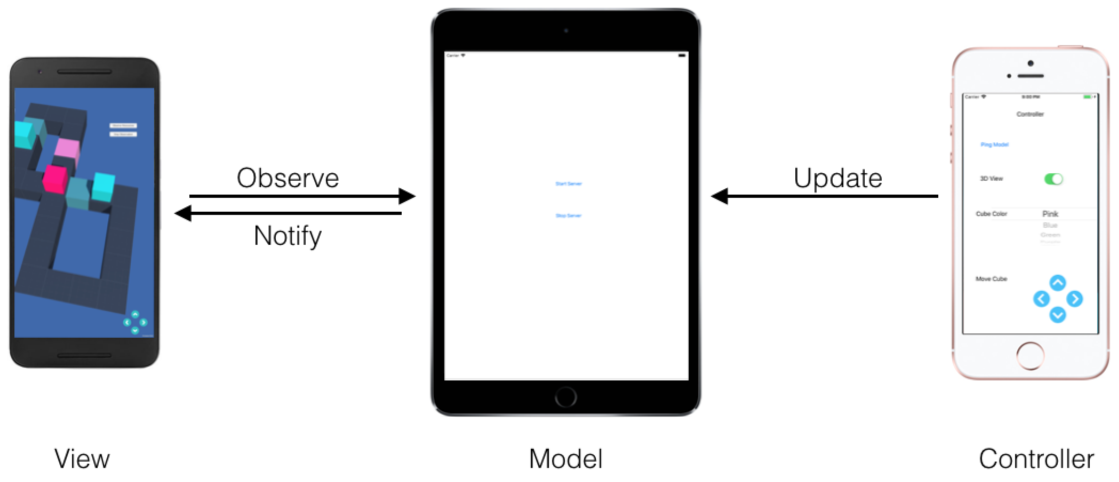
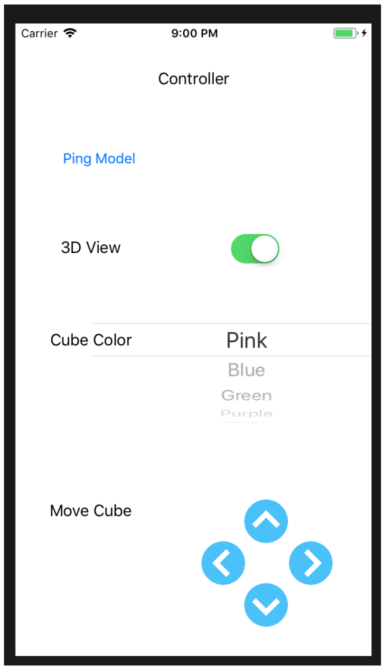
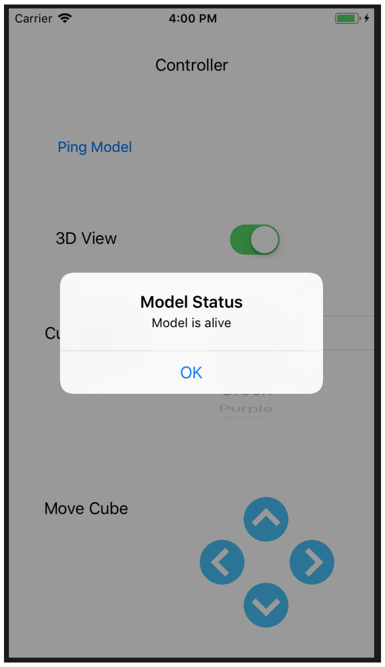


Figure 4.4: Implementation Setup

To demonstrate the proposed architecture, three mobile applications are created as three mobile applications with their functions exposed as services. The CoAP protocol is used as the communication protocol, and the whole system runs in a Wi-Fi environment. Figure 4.4 shows the basic setup.



(a) Screen of Controller App



(b) Alert for Model Status

Figure 4.5: Screen of Controller Apps

4.2 Controller

The Controller is written in C# and compiled into native iOS app, using the cross-platform tool Xamarin [46]. Xamarin chosen because of the following reasons:

1. Xamarin uses C# complemented with .Net framework to create apps for mobile platforms. Thus, source code can be reused over several platforms to speed up the development.

2. Xamarin does not require switching between the development environments; since it was acquired by Microsoft, all apps can be built inside Visual Studio which completely replaced Xamarin Studio. The cross-platform development tools are provided as a built-in part of the IDE.

3. Xamarin apps are built with standard, native user interface controls.

For the implementation of the CoAP library, as discussed in the Related Work section, the CoAP.NET library [34] is used in the development of all MVC modules. CoAP.NET is based on Californium [25], a CoAP framework in Java developed in ETH Zurich. The implementation was originally based on a previous C# CoAP implementation. However, it provided only basic functions and seemed to be out of maintenance. The implementation has migrated to the current active CoAP.NET.

The Controller module is responsible for sending user input requests to the Model module, that is, gathering user input for the game, encapsulating them into CRUD CoAP requests, and sending them to the Model module. For the convenience of gathering user input, a UI is provided. In this Controller app implementation, the UI interface demonstrates three attributes:

1. The camera angle of the view of the game View (default setting is “3d”), an entry for the configuration of the game.
2. A picker of colour for the player cube, can be a part of both game configuration and game play.
3. The player movement control, controls the movement of the player cube using the direction buttons, and is the main game control.

When the screen of the Controller app finishes loading, a new CoAP client is created, with methods including Pinging the Model server to find out its status (if started in the same network), and sending out updated resource representation as CoAP PUT requests to the Model module. After a user toggles a switch and selects a colour from the colour picker or touches any direction button, one of these methods gets called and performed by a Task object asynchronously on a thread pool thread from the main UI thread. A code snippet of the CoAP client is included below.

```

public class ControllerComponent
    {

        private CoapClient coapClient;

        public bool modelStatus = false;

    // Constructor
        public ControllerComponent()
        {
            client = new CoapClient(SERVER_URI);

        }

    // Ping the Model server to see if alive
        public bool PingModel(){

            bool status = client.Ping();

            return status;

        }

            // Sends async PUT request to uri path /camera_view to modify the camera view
    // resource with content format of JSON
        public void ChangeCameraView(string view){

            coapClient.UriPath = "camera_view";

            coapClient.PutAsync(view, MediaType.ApplicationJson, ResponseHandler,
                FailHandler);
        }

    // Sends async PUT request to uri path /player_colour to modify the play colour resource
    // resource with content format of JSON
        public void ChangePlayercolour(string colour){

            coapClient.UriPath = "player_colour";

            coapClient.PutAsync(colour, MediaType.ApplicationJson, ResponseHandler,
                FailHandler);

        }

    // Sends async PUT request to uri path /player_move to modify the play move resource
    // resource with content format of JSON
        public void MovePlayer(string direction){

            client.UriPath = "player_move";

            var response = coapClient.PutAsync(direction, MediaType.ApplicationJson);
        }


    }
```

4.3 Model

Same as the Controller module, the Model is implemented using Xamarin. The app includes a CoAP server that holds resources that are game play data. A resource is a conceptual mapping of a set of entities, which varies according to the genre of a game. For a turn-based board game, resources can be the board, game pieces, players, and moves. For an adventure RPG (Role-playing Game), where a player controls a main character to explore given maps, and advances the game story by interacting with items and NPCs (Non-player Character), resources can be the main character, character position, inventory, or game level, entities that exist at a given time over the course of a game. In this case, API is exposed, modifying the camera angle of view, the colour of the player cube, and player movement.

When the screen finishes loading, the user can choose to tap on the button “Start Server” to start a CoAP server. The server first registers resources and then adds endpoints for chosen IP addresses and ports

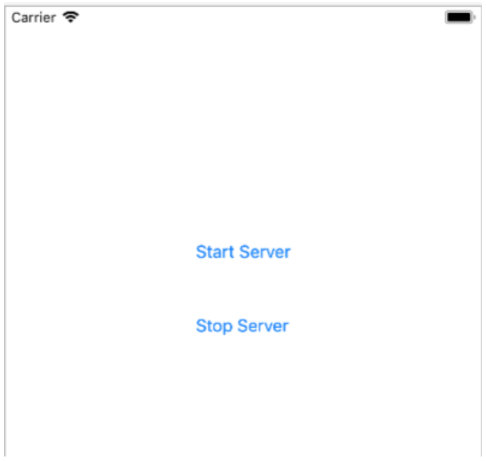


Figure 4.6: Screen of the Model App

(or binds to all available network interfaces if none is specified). A code snippet for both the server and player_move resource class is listed below.

```

public class ModelServer
    {
        private CoapServer server;

    // Default CoAP port
        const int COAP_PORT = 5683;

        public ModelServer()
        {
            server = new CoapServer();
        }

        public void StartServer(){

        // Adds server resources
        server.Add(new CameraViewResource("camera_view"));
            server.Add(new PlayercolourResource("player_colour"));
            server.Add(new PlayerMoveResource("player_move"));
            try
            {
                // Get endpoint for suitable IP4 address
                var host = Dns.GetHostEntry("");

                IPEndPoint ip4Ep = new IPEndPoint(host.AddressList[0], COAP_PORT);

                CoAPEndPoint unicast = new CoAPEndPoint(ip4Ep);

```

```
                    server.AddEndPoint(unicast);

                    server.Start();
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
        }
}
```

CoAP supports resource discovery, a mechanism whose the main function of such a discovery mechanism is to provide URIs for the resources hosted by the server, complemented by attributes about those resources and possible further link relations. The CoRE Link Format is carried as a payload and is assigned an Internet media type. A well-known relative URI "/.well-known/core" is defined as a default entry point for requesting the list of links about resources hosted by a server, and thus for performing CoRE Resource Discovery. As is shown in the code snippet for the player move resource, the resource description is set inside its constructor.

When a server receives a request, it passes the request to according resources. Then, the request handler inside each resource routes the request to corresponding handlers. If an observation request is received (a request with Observer Option set to 0), an observe manager that is bind to the server keeps track of the observe relationships which represents a relationship between a client and a resource on this server. The components called "observers" register at a specific, known provider called the "subject" that they are interested in being notified whenever the subject undergoes a change in state. The subject is responsible for administering its list of registered observers. If multiple subjects are of interest to an observer, the observer must register each separately. When a resource being observed is changed, the executor of this resource notifies a set of CoAP clients which have established an observe relation with this resource, that the state has changed by reprocessing the original request that established the relation.

In this implementation, all three resources, camera_view, player_colour, and player_move, are defined as observable resources. The observer pattern is a core mechanism of the proposed architecture, because the update of the View using resources in the Model is solely based on observation. So the View has to send an observe request to the Model once, and waits for responses for an update of resources.

```

    class PlayerMoveResource : Resource
    {
        private string _move;


        public PlayerMoveResource(String name)
            : base(name)
        {
                // Constructor for setting resource description for resource discovery
        // requeststo /.well-known/core
            Attributes.Title = "Player Moves";
            Attributes.AddResourceType("PlayerMove");

            // Sets the resource to be observable
            Observable = true;
       // Set initial value for the value of player_move
            _move = "left";
        }

    // Handler for GET request
        protected override void DoGet(CoapExchange exchange)
        {

            exchange.Respond(StatusCode.Content, _move, MediaType.ApplicationJson);
        }

    // Handler for PUT request
        protected override void DoPut(CoapExchange exchange)
        {
            _move = exchange.Request.PayloadString;
            exchange.Respond(StatusCode.Changed);
        // For observable resource, notify registered Observing clients for this resource
            Changed();
        }
    }
```

4.4 View

The View module is implemented as a mobile game instance, using the Unity game engine. The main function of the View module is to render the game screen in the proposed architecture. To acquire input of the game control (e.g., player movement) and change of game configuration (e.g., camera view, player colour), the View registers to observe resources in the Model. There is no need to constantly poll for an update of resources,

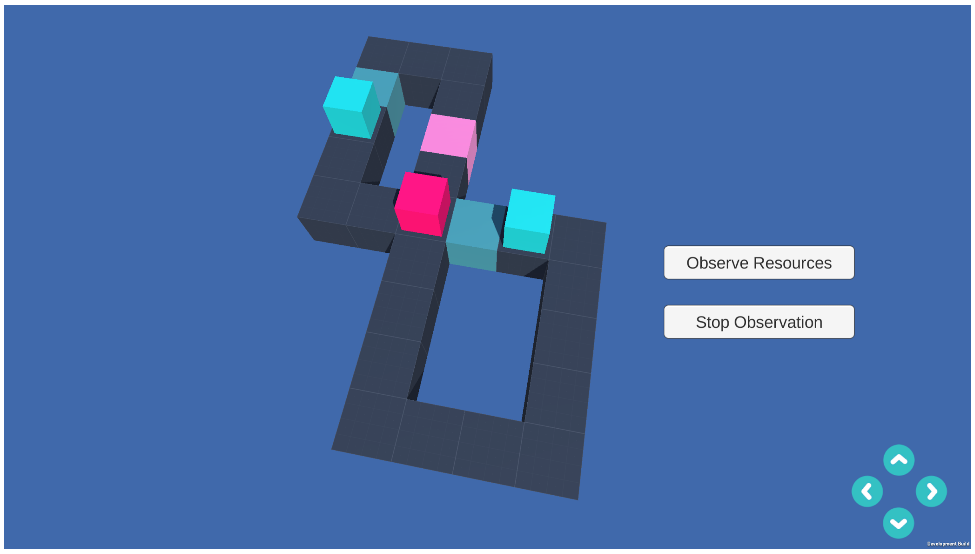


Figure 4.7: Screen of the View App in 3d View

since the Model server will send updates to the View.

In a number of game engines, game objects are tagged as separate entities. Each entity is a container of components, which implements actual functionality that capture properties of the object, define object movement and its interactions with the world. For instance, every game object in a game world would have a transform component that captures its world position, rotation, and scale. Movement of a game object can be achieved by modifying data in the transform component; in addition, with a rigid body component attached, a game object is subject to the control of the physics engine. The rigidbody component can receive forces and torque, making game object movement more realistic looking than through the transform component.

Resources shall be mapped to user configurable properties of a game object or GUI and performable behaviors and identified by URIs. For example: /player/physics/gravityScale and player/behaviors/jump. The View will monitor incoming updates, and perform function calls into the controller logic inside the View, which will then modify attributes of according game objects to complete the update of visual representation of resources.

Below is the code snippet for starting a CoAP client, sending a request for observing the player movement resource, and updating a player in the game scene using received data. Note that any UI update can only happen in the UI thread:

```
// Use this for initialization
void Start()
{
// Bind endpoints to CoAP client
    IPEndPoint localEp = new IPEndPoint(IPAddress.Any, 0);
    CoAPEndPoint ep = new CoAPEndPoint(localEp);

    client = new CoapClient(SERVER_URI){EndPoint = ep};
```

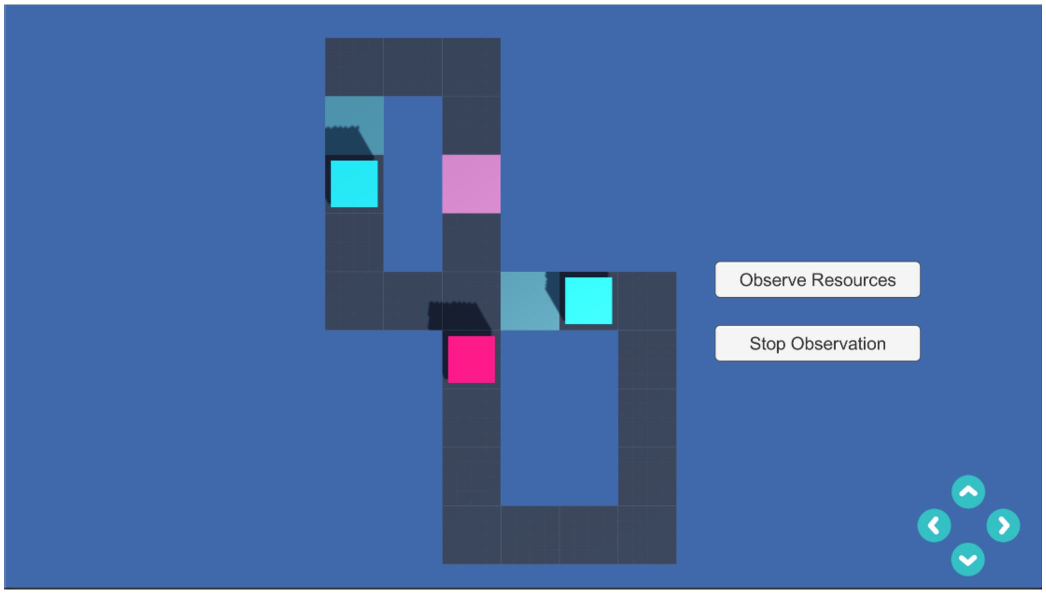


Figure 4.8: Screen of the View App in 2d View

```
        ep.Start();

        // Set requests to be Confirmable, so server will send
        // Acknowledgement responses
        client.UseCONs();

      // Get reference to Controller component to the player
        playerController = playerCube.GetComponent<Controller>();

        // Creates a TaskScheduler associated with the current UI thread
        UIschedular = TaskScheduler.FromCurrentSynchronizationContext();
    }

    public void ObserveResources(){
    // Set path to player movement resource
        client.UriPath = "/player_move";

        // Set a GET request with an observer option
        // Register Notify() as action to take for each notification
        client.ObserveAsync(MediaType.ApplicationJson, Notify);
    }

    void Notify(Response response){
    // Same as other mobile applications
    // any modification for UI must happen in main thread
    // queues an action to be invoked on the main game thread
        Task task = new Task(() => {
            playerController.MoveByController(response.PayloadString); });

        // Starts the Task, scheduling it for execution to the specified UIScheduler.
        task.Start(UIschedular);

        // Turn off auto-reconnection with server
        observeMove.Request.ObserveReconnect = false;
        }


    public void CancelObservation(){

    // Send a request to actively cancel observation
        Request request = new Request(Method.GET, true);
        request.MarkObserveCancel();
        client.Send(request);
    }
}
```

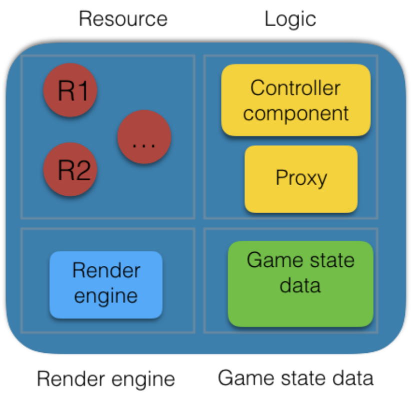


Figure 4.9: Architecture of the View Module

Since the View module is a game instance, its life cycle is essentially the same as a game, as shown in Figure 4.10.

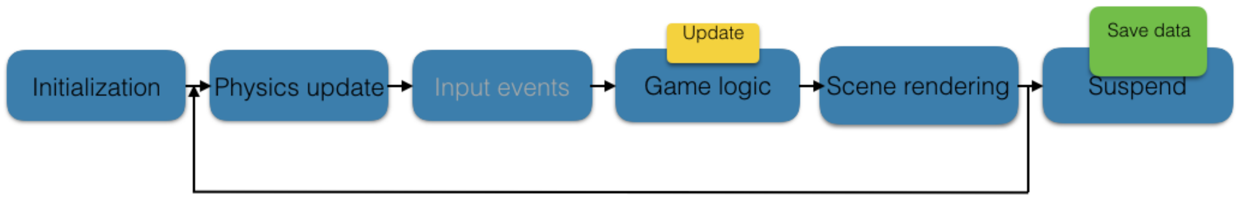


Figure 4.10: Life Cycle of the View Module

In this scenario of single-player games, only one View exists, but multiple Views can be present simultaneously. They can be set to all render the same set of game objects and game states, while using different representations according to the user configuration (e.g., one in 3d, and another in 2d), or can be sets to observe different set of resources, and thus render totally different contents.

4.5 Optional Services

4.5.1 Save Game Progress

An optional service for the View is to keep track of game state data locally. Data usually consists of values, including player health, wealth, and items in inventory. There are two common methods to store state data locally. One is to write variables to a file in binary or XML(eXtensible Markup Language) form. Note that binary data uses disk space. Most commercial game engines provide tools to help implement such features, for example, the PlayerPrefs API in Unity Engine. However, the main disadvantage of saving data as PlayerPrefs is that it does not come with support encryption, so manual encryption is needed in order to prevent players from modifying saved game state data. The other method is to store variables in a database, which provides the ability to query data structures dynamically. This differs from the previous method, where when a save file is loaded, all the game state data is loaded into the memory. Since the database method is more suitable for games require dynamic query with larger set of data with more complex data structure, and many databases support encryption.

In the context of the proposed architecture, both uploading the game state data file and posting the serialized object would be applicable.

4.5.2 Game Data Storage

Additional modules can be added to the architecture to act as game backends. A module can work as a game server and provide two services inside the device cloud. The first is hosting the game content for game instances to request and load into a running game; the other is to preserve posted game status data from the aforementioned Save Game Progress function and wait for further requests.

The contents to be loaded include numerous game assets, including models, textures, audio clips, entire game levels, and new items to be unlocked. It is a common strategy to load assets from a separate local file or server during game play to reduce initial installation time and space or to allow interchangeable game content. In the case where characters or objects can appear in uncertain scenes of the game but only infrequently (for example, an error message), it is also beneficial to make an asset available to a project without loading it as part of a game scene.

When new game models need to be added to the current game scene, the game instance running as the View will either load the model from included assets (which is pre-built), or download from the Storage

Server.

The Unity engine, it supports two ways of packaging contents to be loaded at runtime. The first one is to place the contents in a set directory in a game project, which allows content to be supplied in the main game file yet not be loaded until requested. The second is to create Asset Bundles out of the contents as an external collection of assets. These are files completely separate from the main game file. They will be saved into the file server on Storage Server for the game instances to request during runtime. The workflow of using Asset Bundle is shown below in Figure 4.11.

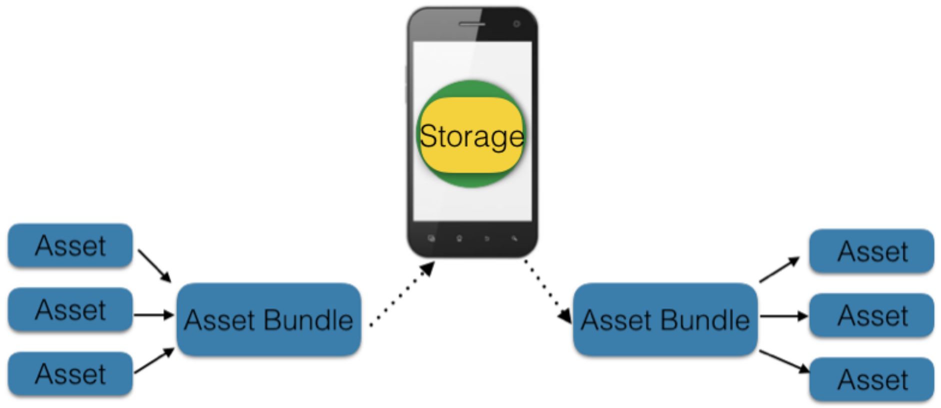


Figure 4.11: Asset Bundle Workflow

Using Asset Bundles calls for extra steps of uploading them to external storage and downloading them at run time from a current application from a script within a game scene as either non-caching or caching download. The latter will cache the downloaded Asset Bundles to a Cache folder in the local storage device running the View.

Game state data is posted from the View to the File Server when a play session ends, in order to preserve data between sessions, which can also be used to synchronize multiple Views to a same game state. Game state data will only be accessed by modules inside the mobile cloud at this stage. Thus no connection to an external data server will be considered in this research. Should an external server be needed, adding such support should be straightforward.

In the presence of an uncomplicated single-player game, game status will only be captured at certain locations with a small set of status. A simple file server will be sufficient to store data files, whereas for games that generate of a large amount of data dynamically, a database is needed to handle dynamic data query.

Chapter 5

Evaluation

In this chapter, the main goal is to evaluate the performance of the proposed architecture by testing the implemented prototype for the performance attributes shown in Table 5.1. The evaluation was conducted in a realistic network setting to decide its feasibility.

Evaluation Goal	Experimentation
1. Response Time and Throughput for Main Methods	(a) Measure Response Time and Throughput under different request sending frequency settings.
	(b) Measure Response Time and Throughput of GET and PUT requests under different payload sizes.
2. Resource Observation	Evaluate Response Time and Throughput for Subscription request and Round Trip Time for Notifications messages.
3. Scalability	Compare Response Time by adding an extra Controller and have the two Controllers sending GET and PUT requests to the Model on the same resources.

Table 5.1: Evaluation Goal

1. (a) Measure response time and throughput for key methods PUT and GET requests. The two main methods supported in the prototype for sending user control for READ and UPDATE data are under different package sending frequencies, with a fixed size payload. Evaluations for a set frequency is repeated 2000 times.

   (b) Measure response time and throughput of GET and PUT requests under different payload settings, to show the correlation between performance and payload size. Evaluations for a set payload is repeated 2000 times.

2. Measure server response time and throughput for a resource subscription request and round trip time for a server sending notification to the client. Evaluations for subscription is repeated 1000 times and 500 times for notification.

3. Compare the performance when adding an extra Controller component to the previous single Controller setting.

5.1 Evaluation Setup

The prototype to be evaluated has the same configuration as stated in the implementation, which consists of two CoAP Clients, one CoAP Server and the network infrastructure. The functions of the components are subjected to the aforementioned role of the Controller, the Model and the View, where the Controller captures user inputs, encapsulates and sends them as CoAP requests. The Model listens to these requests, updates corresponding resources which are representations of manipulatable objects in its game View. The View receives the updates by subscribing to its interested resources, and updates the game View. There were no other applications running on any devices, but there may be contain certain system services running in the background. The prototype runs in a dedicated wireless network to simulate an at-home environment.

The hardware setup is shown in Figure 5.1

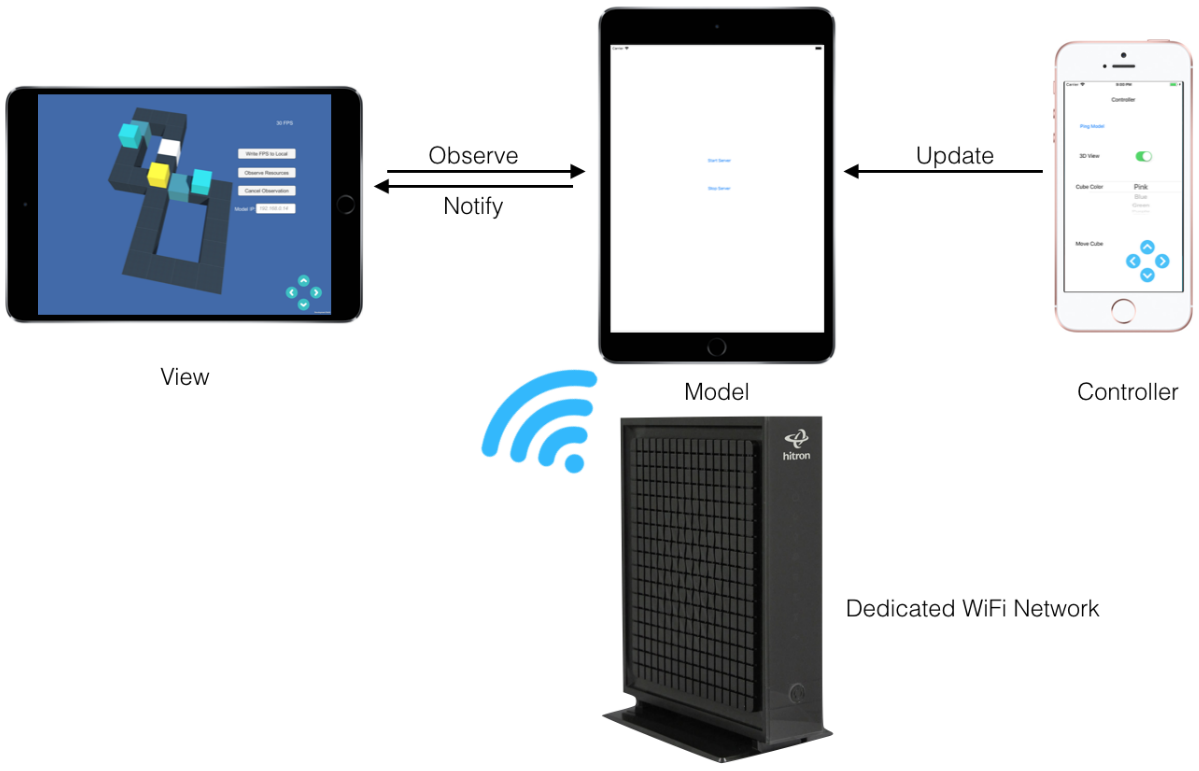


Figure 5.1: Hardware Setup of Evaluation

The implementation aforementioned is deployed over three iOS devices:

- iPhone 6s Plus (iOS 11.2.2) as the Model component;
- iPad Air (iOS 11.2.2) as the Controller component;
- iPad Air (iOS 11.2.2) as the View component;

- hitron CGNM_2250 wireless router; and
- iPhone 6s (iOS 11.2.2) as an additional Controller component.

All device are located within 5 meters of the range of the router, and the Wi-Fi signal appears to be strong on all devices.

5.2 Data Collection

The data collected is,

- The response time for GET and PUT requests in milliseconds;
- The response time for Subscription Request in milliseconds;
- The round Trip Time for Observation Notification messages in milliseconds; and
- The payload size in bytes.

In each experiment, data is captured using APIs that are included in CoAP.NET library, except for Notification Round Trip Time, which is calculated using packet time stamps captured by WireShark [39], an open source packet analyzer.

5.3 Result Analysis

The result is collected and plotted according to the three groups in the experiment. In the following figures, the horizontal axis represents either a request sending frequency that is derived from various fixed request sending intervals, or a payload size from 16 bytes to 1024 bytes.

In the first set of evaluations, the Controller component sent 2000 GET requests after every request sending intervals from 2 ms, 4 ms, 6 ms, 8 ms, 10 ms, 20 ms, ... to 100 ms, from which were derived frequencies of 10, 11, ... 250, 500 requests per second. The same pattern applies to the PUT evaluation below. Requests are all sent to the Model component at URI /player_move to READ current player movement, in order to receive a response with a string payload of a direction. Tn this case it is 16 bytes throughout the first set of evaluations. At an increasing frequency from 10 requests per second till 500 requests per second, the Model receives requests and sends current resource representation to the Controller. Since CoAP protocol is based on UDP and has no requirement for maintaining connections, a higher sending frequency can simulate scenarios where multiple Controllers are sending in requests in a multi-player setting. The average response time for each sending frequency is calculated, and the result is shown in Figure 5.2 and 5.4. Figure 5.4 depicts all 2000 response times for all frequencies to show the variance of data.

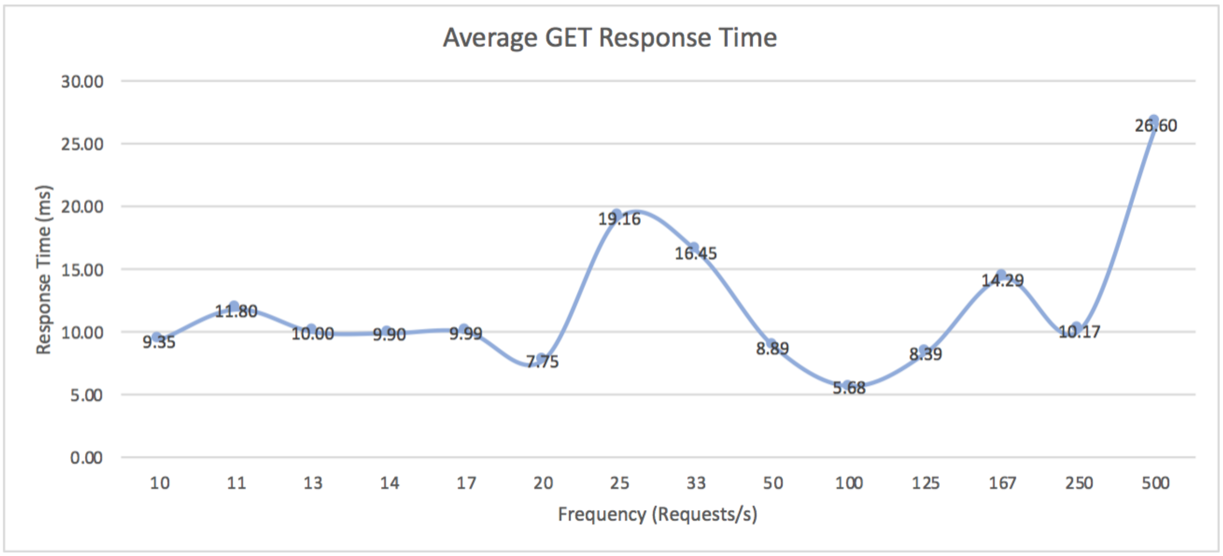


Figure 5.2: Average Response Time for GET Request under Various Frequencies

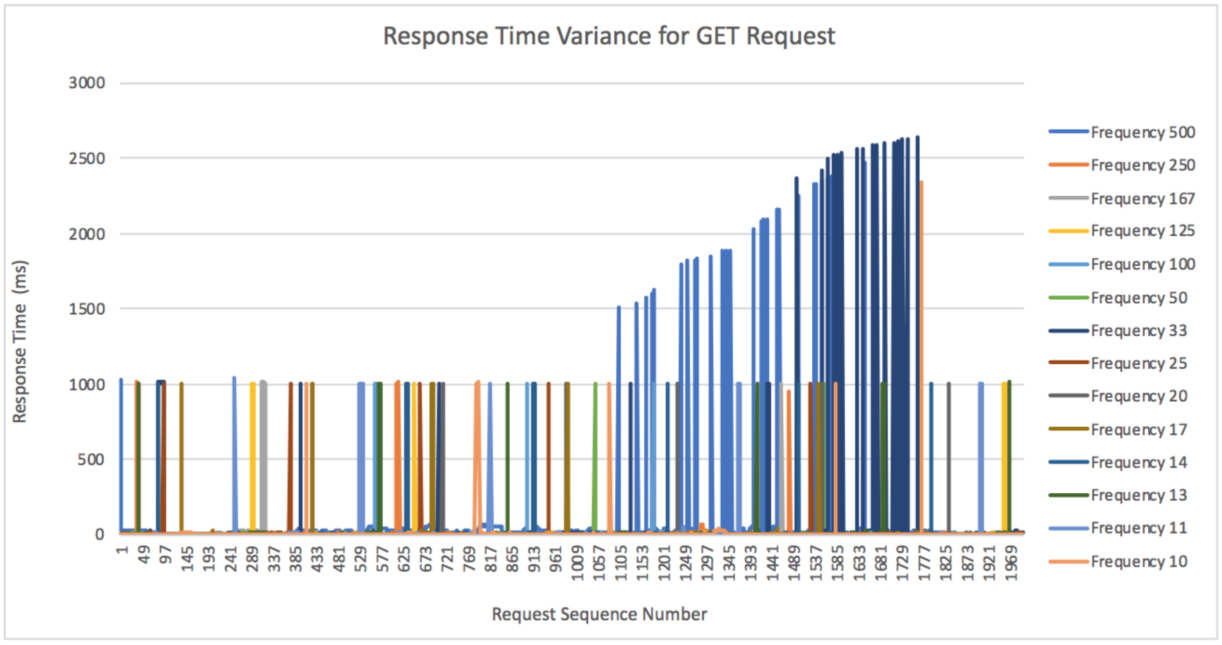


Figure 5.3: Response Time Variance for GET Request under Various Frequencies

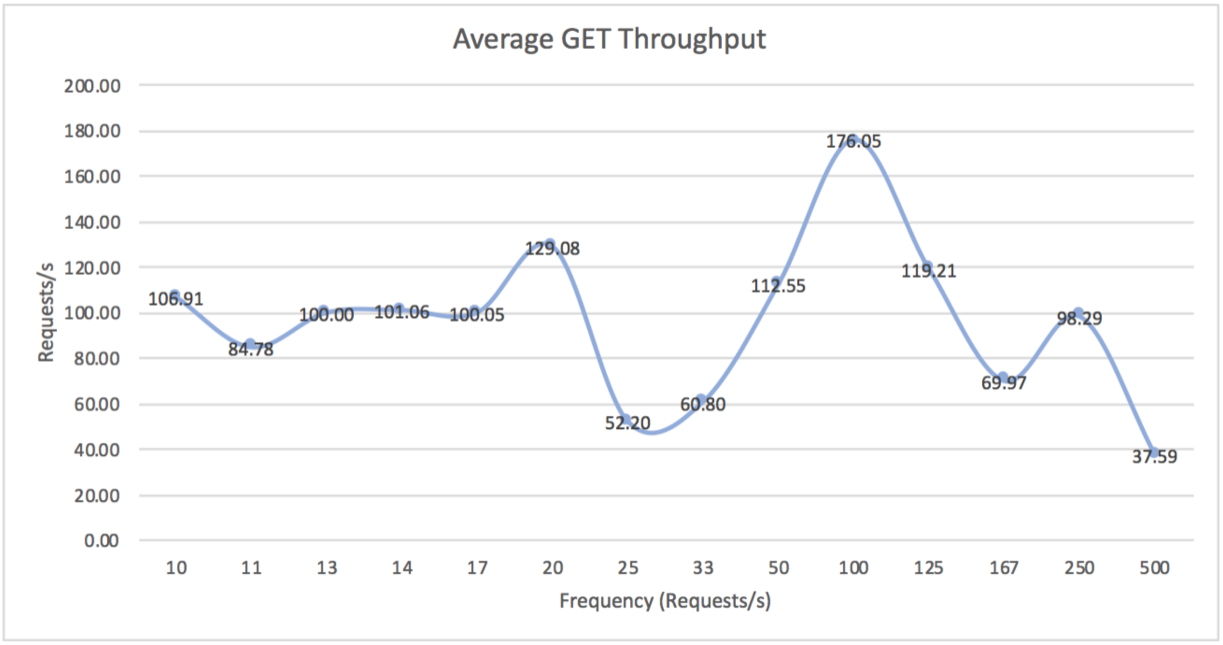


Figure 5.4: Average Throughput for GET Request under Various Frequencies

Figure 5.2 shows the response time for the GET Requests remains steady at 10 - 20 requests/s. When the sending frequency reaches 25 requests/s, a fluctuation starts, possibly because of the increasing requests the Model Server has to deal with concurrently. The average response time peaks at 500 requests/s.

Figure 5.3 shows the overall variance of the response time. The horizontal axis shows the sequence number of requests in 2000 evaluations. It is difficult to tell but most dots are located near the bottom hence the average value us 10 - 20. The raised blue part represents the frequencies 500 and 33, respectively. The initial time-out for this prototype is set to default, which is a random value between 2 to 3 s, and some of the rises might be a result of retransmission. In this implementation, the GET method is available. But hardly ever used, so the possible time out does not affect much of the overall performance.

For the throughput shown in Figure 5.4, it stays rather stable when a request sending frequency is low, and is less stable towards the higher frequency range.

In Figure 5.5 and 5.7, the result for PUT requests is shown. The evaluation setting is the same as GET requests. Similar to the results shown in GET evaluations, the average PUT response time was relatively steady when the sending frequency is low. Response time distinctly rises when the frequency is 167 - 500 requests/s, the same as GET. This could be due to increasing concurrent handling of requests on its Server. The peak happens at 100 - 125 requests/s. For higher performance, the PUT sending frequencies should be set lower than 100 request/s. Figure 5.6 shows the overall variance is better than the GET request, which is favourable because PUT is the mainly responsible in this prototype for updating game data.

Figure 5.7 shows the throughput is most stable at 14 - 50 requests/s, and decreases at 167 - 500 requests/s.

The following figures show the result from measuring response time and throughput when the payload size

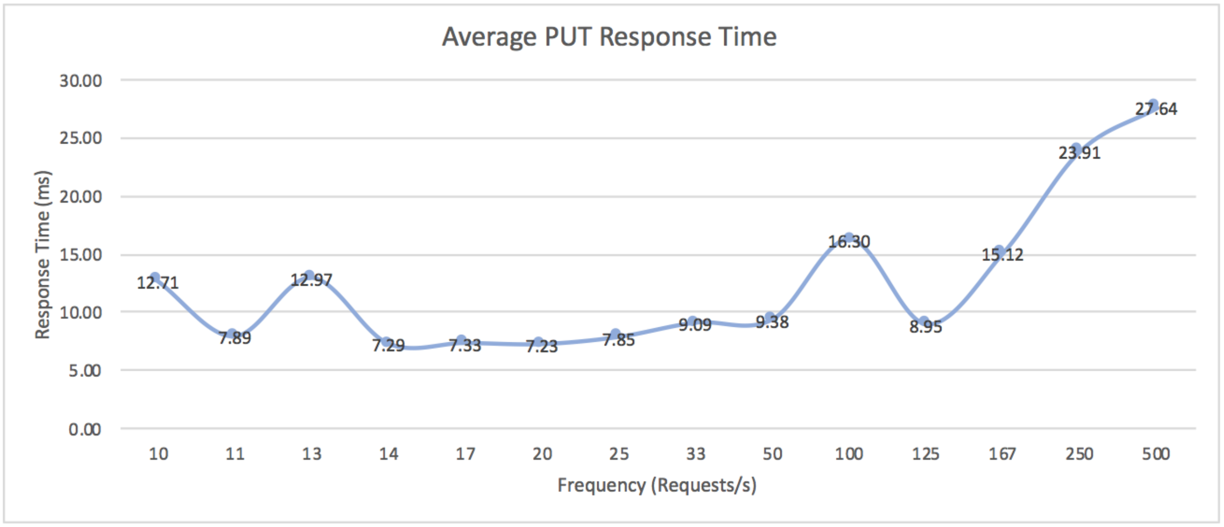


Figure 5.5: Average Response Time for PUT Requests under Various Frequencies

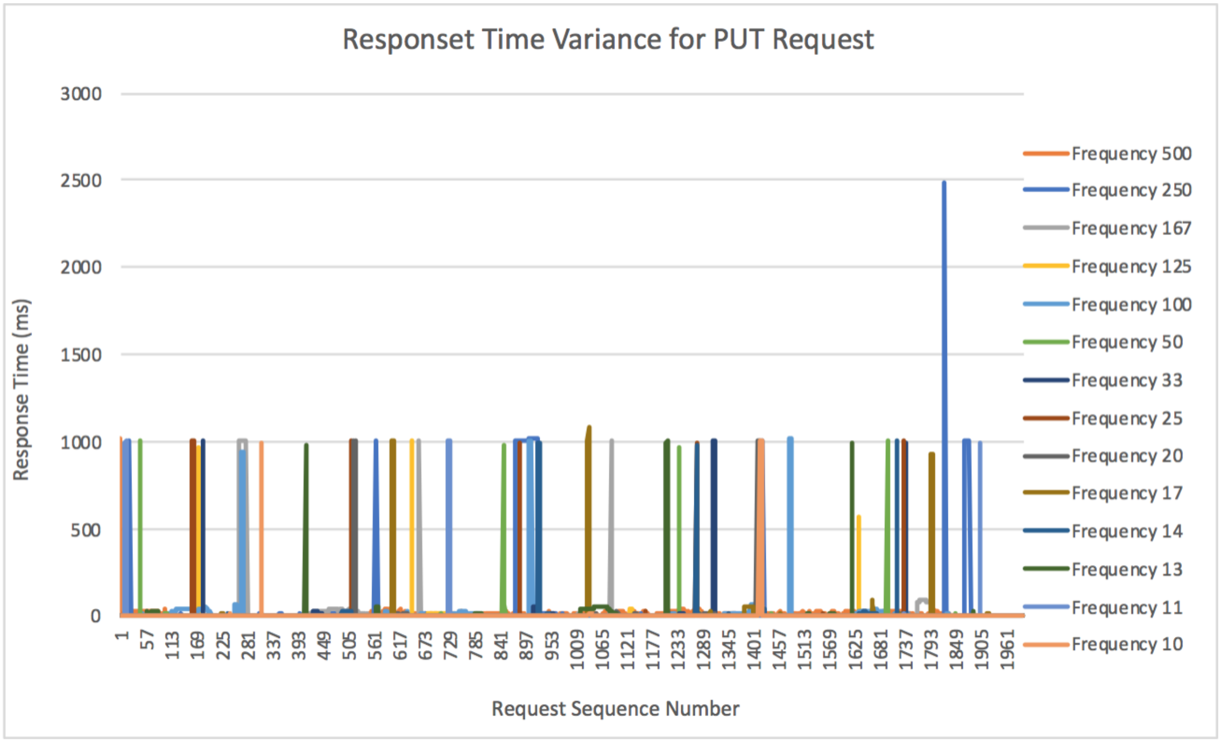


Figure 5.6: Average Response Time for PUT Requests under Various Frequencies

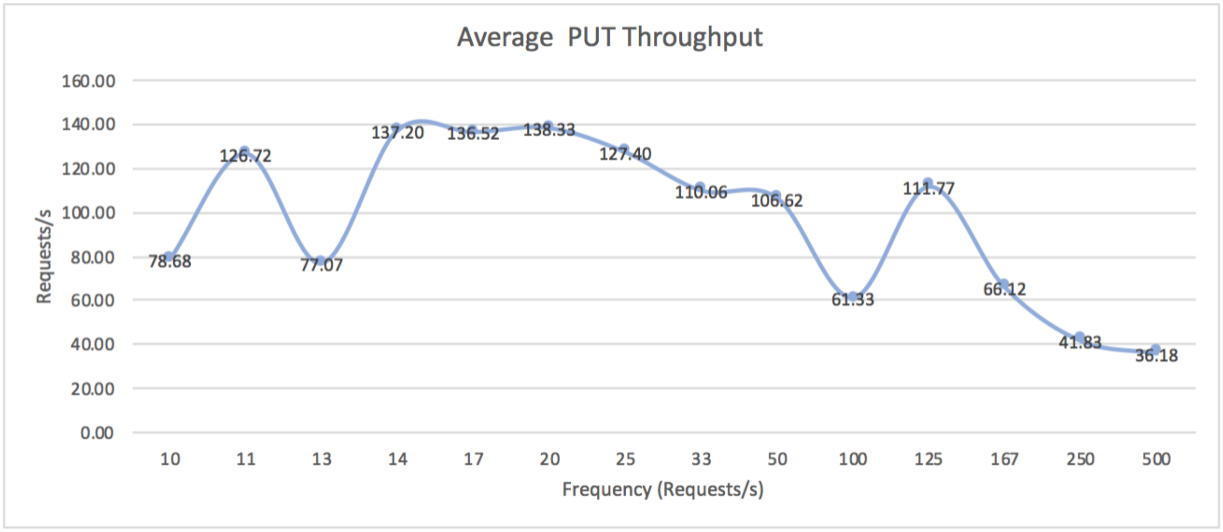


Figure 5.7: Average Throughput for PUT Requests under Various Frequencies

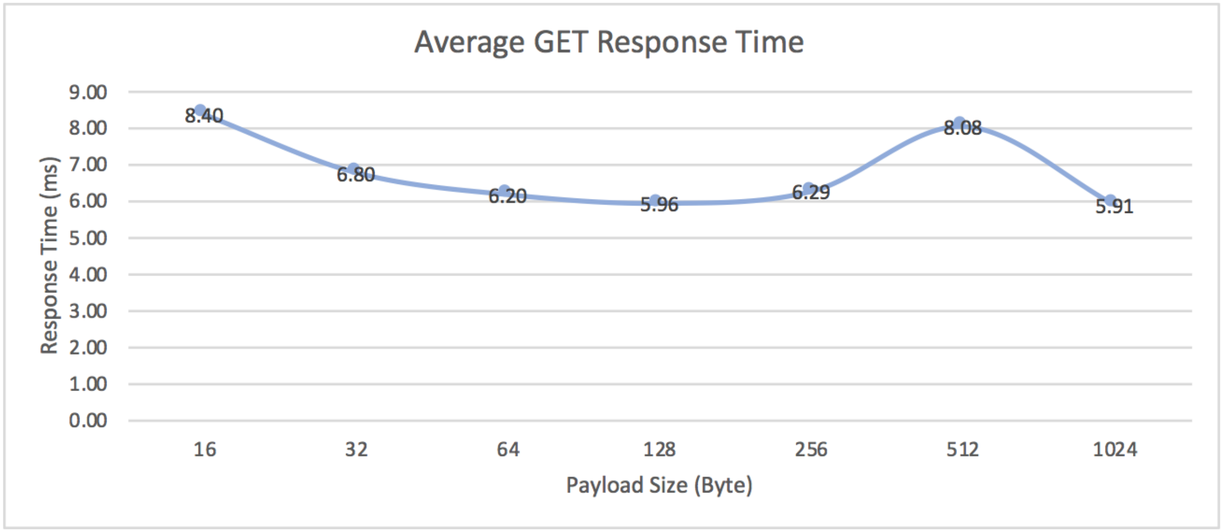


Figure 5.8: Average GET Response Time for Various Payload Size

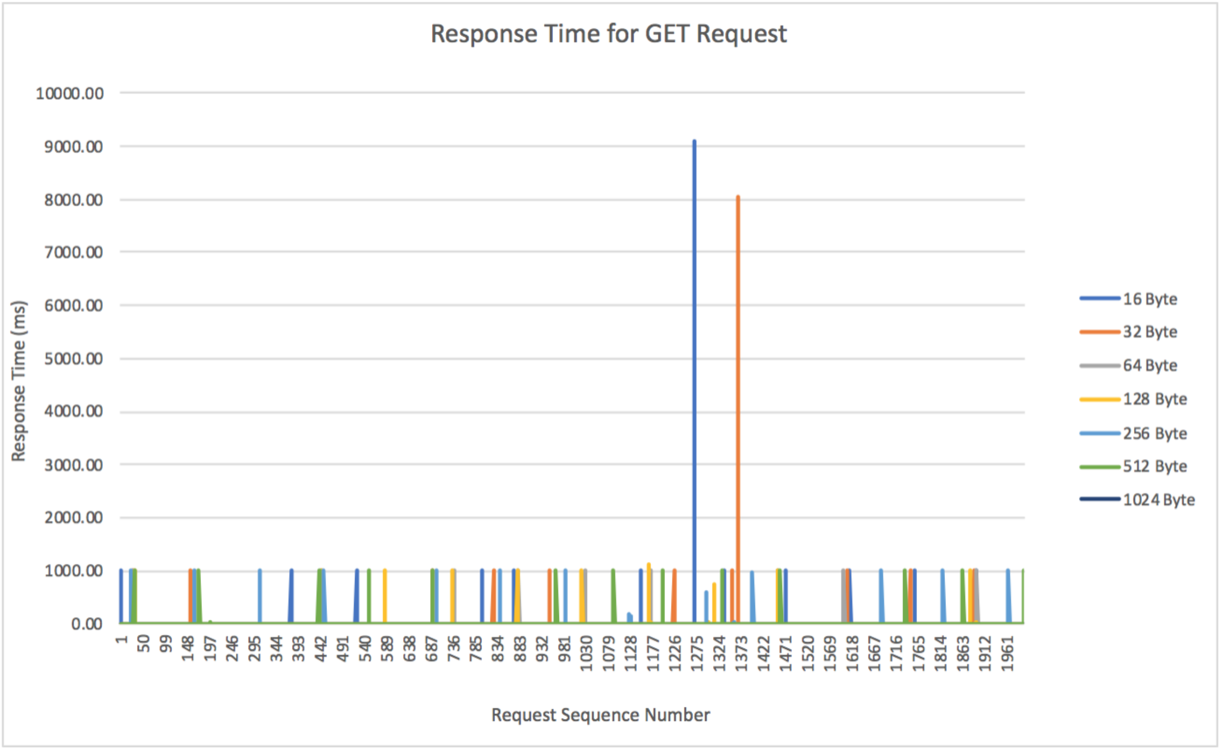


Figure 5.9: Response Time Variance for GET Requests for Various Payload Size

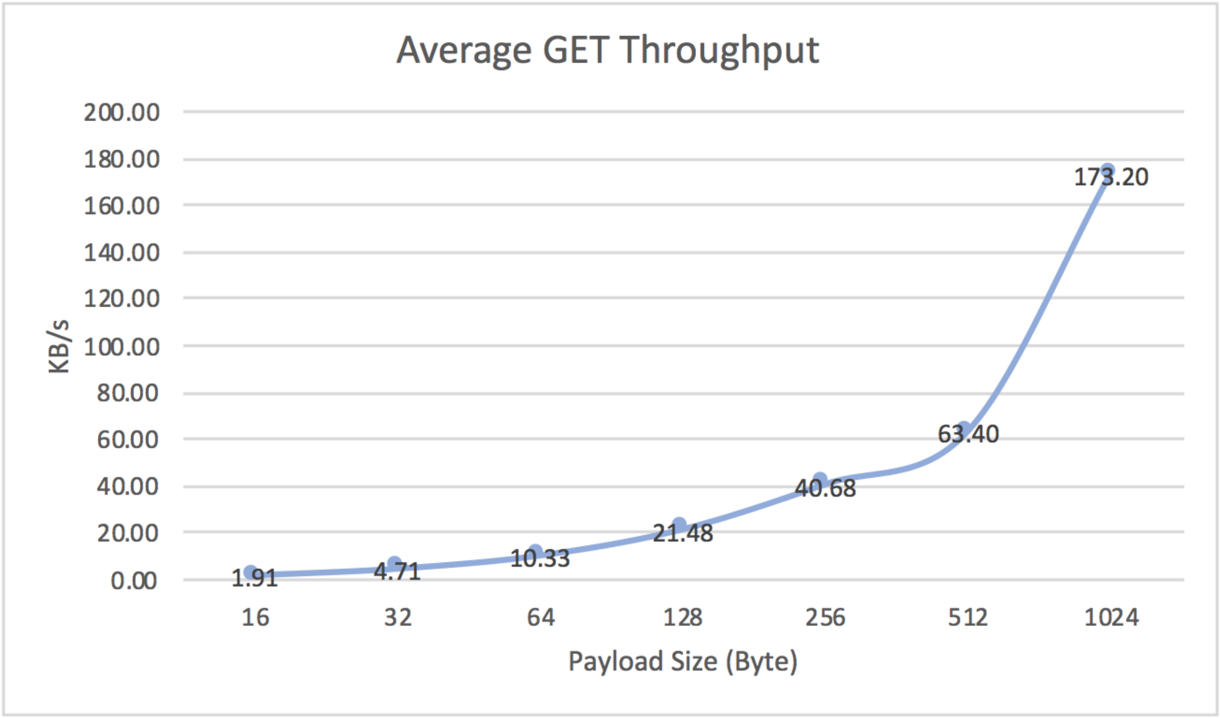


Figure 5.10: Average GET Throughput for Various Payload Size

changed from 16, 32, 64, 128, 512, to 1024 bytes (the upper limit for the payload size for CoAP messages). Evaluation for each payload size setting is repeated 2000 times.

The response time for GET requests in Figure 5.8 remains in the range of 5.91 ms to 8.4 ms with a rise at 512 bytes. Because the variance is not evident, the rise could be caused by network delay or OS activities. In a scenario where GET is used frequently, according to this result, it is better to avoid 16 and 512 bytes. In Figure 5.9, the obvious rise in response time happens at 16 and 32 bytes, which could be because of the Server warming up. In Figure 5.10, the slope from 16 to 256 bytes is roughly 1 whereas from 512 to 1204, the slope nearly triples.

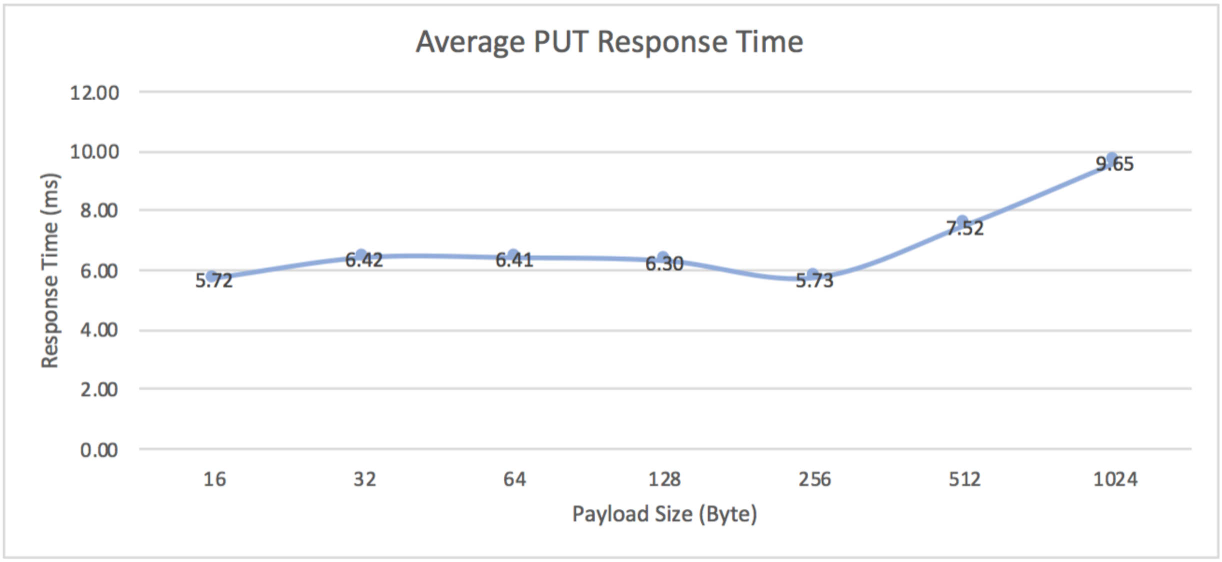


Figure 5.11: Average PUT Response Time for Various Payload Size

For the PUT request, in Figure 5.11, the response times peaks at 1024 bytes, and although the response time dips at 256 bytes, there is a general trend of response time increases as the payload sizes doubles. So, for better performance, payload sizes should be set to smaller values. Figure 5.12 the fluctuation was minimal and the payload size change does not affect response time in a obvious manner. Figure 5.13 shows the throughput nearly doubles as the payload size does.

In Figure 5.14, 5.15, and 5.16 show the result from the first part of the second evaluation, which is measuring response time and throughput resource subscription. Evaluation for each payload is repeated 1000 times. It is clear the response time peaks at 128 and 512 bytes. For this specific application, subscription requests are only sent a few times in the beginning, but in the scenario where there are multiple observers or subscription requests, these two payload sizes should be avoided. The variance graph 5.15 is rather uneventful, but there is an increase at 1000 ms, which may be a characteristic caused by a combination of network transfer and server configuration.

For evaluation of the round trip time of Notification, the minimal sending interval for notification is set to 50 ms. Because the CoAP.NET library implemented the basic congestion control mechanic defined in RFC

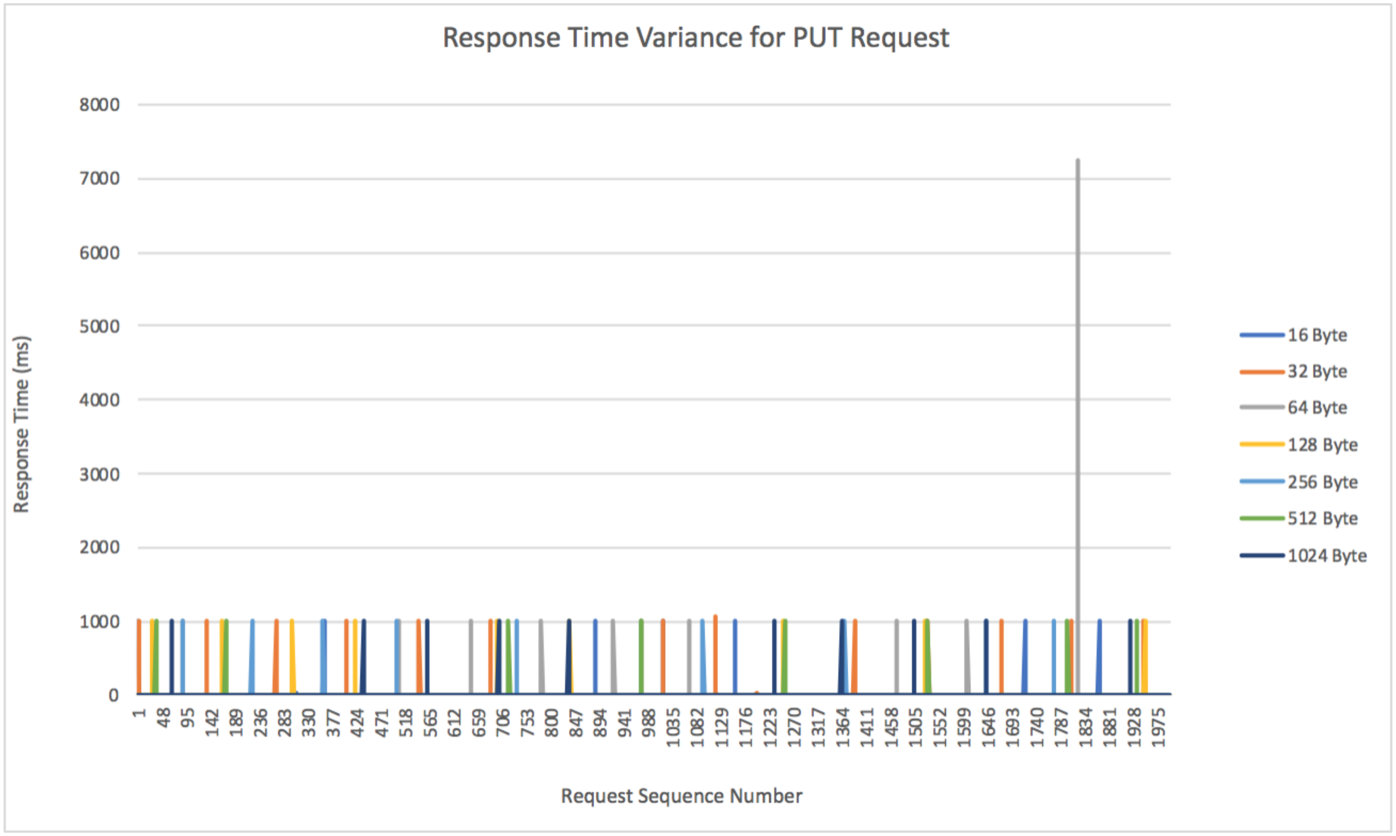


Figure 5.12: Response Time Variance for PUT Requests for Various Payload Size

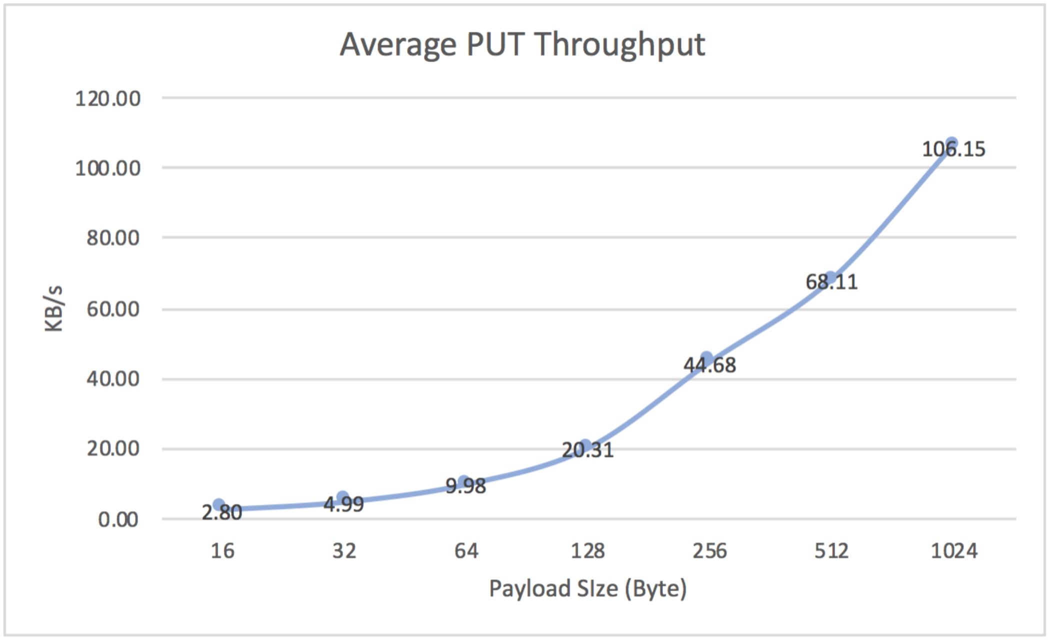


Figure 5.13: Average PUT Throughput for Various Payload Size

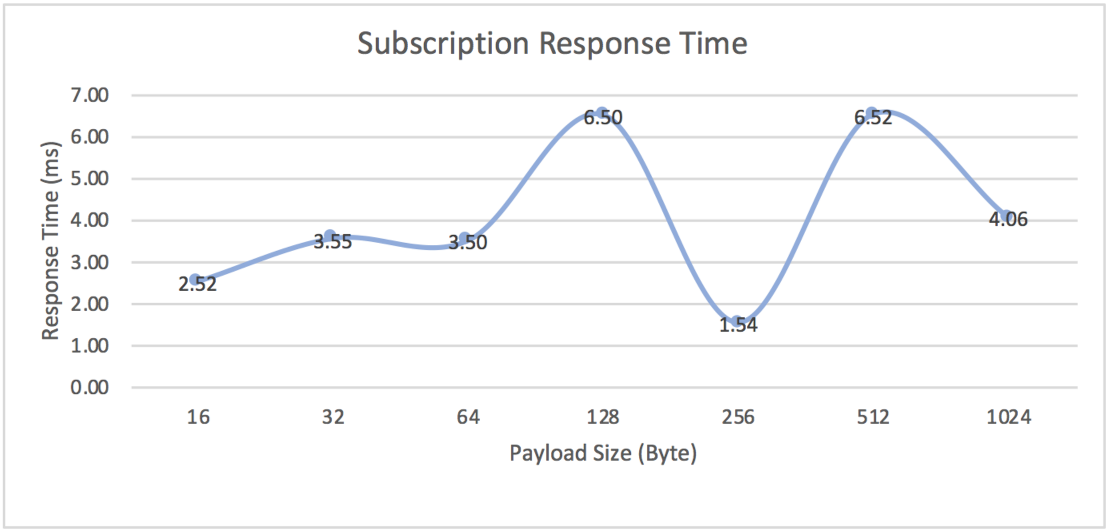


Figure 5.14: Average Response Time for Subscription Requests

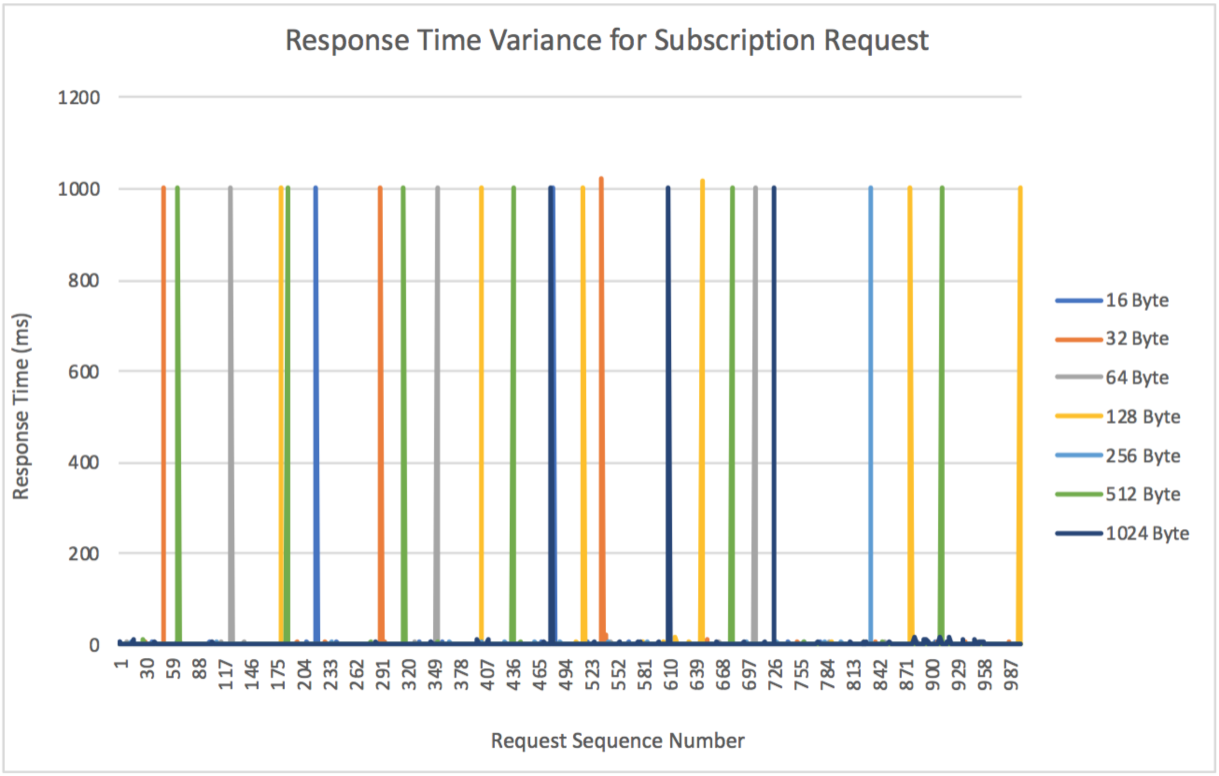


Figure 5.15: Response Time Variance for Subscription Requests

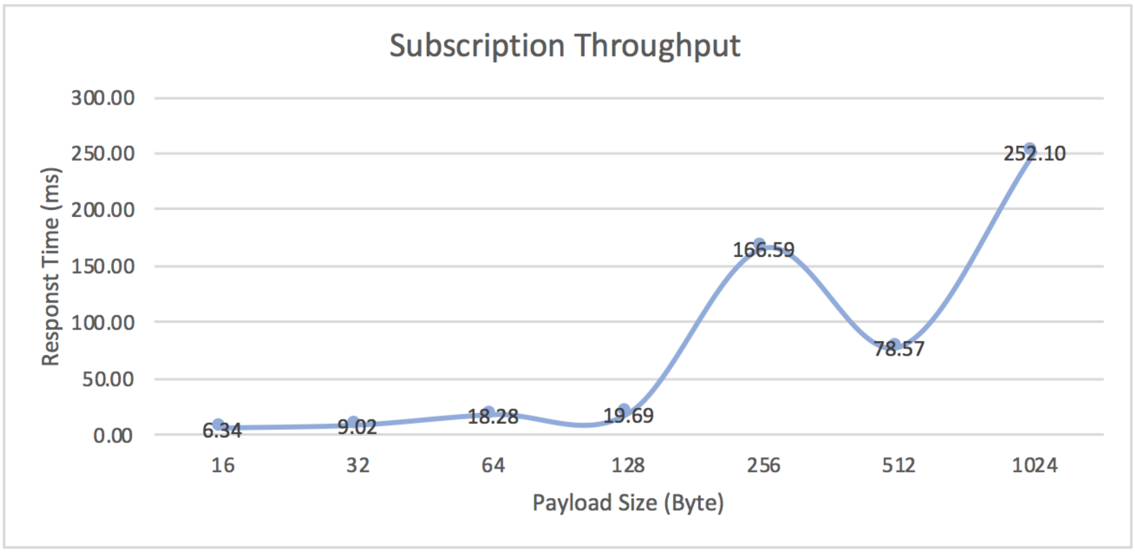


Figure 5.16: Average Throughput for Subscription Requests

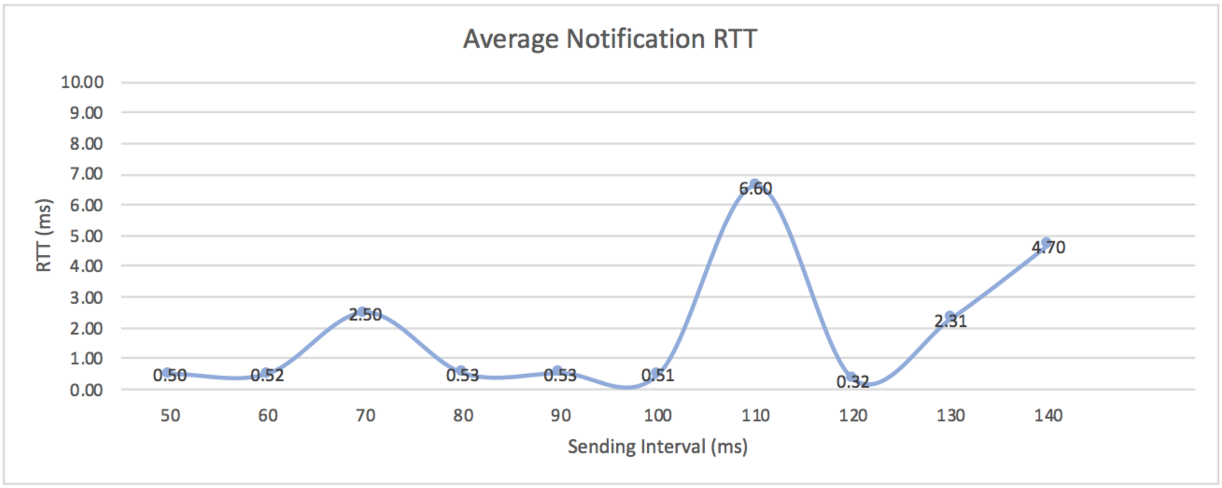


Figure 5.17: Average Round Trip Time for Notification

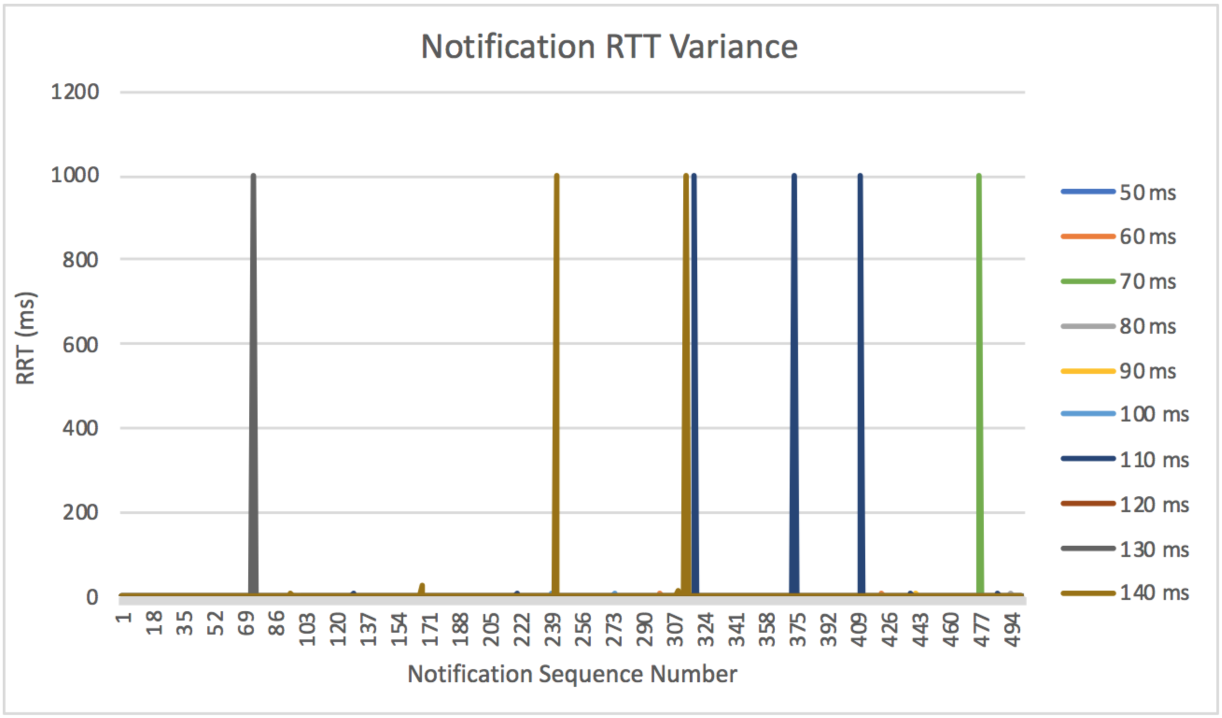


Figure 5.18: Round Trip Time Variance for Notification

7252 [35], where in order not to cause any congestion, a clients must strictly limit the number of simultaneous outstanding interactions. An outstanding interaction is either a CON for which an ACK has not yet been received, but is still expected or a request for which neither a response nor an Acknowledgment message has yet to be received. The default number set in the CoAP.NET library for outstanding interactions is 1. In this evaluation setting, in order for the Model component to receive Acknowledgements from the View component steadily, the sending frequency cannot go any higher, that is the minimal sending interval for notification is 50 ms. As is shown in Figure 5.17, the round trip time is in the range of 0.32 ms to 6.60 ms, and peaked around 110 ms sending intervals. However the overall value is quite small. For this application which depends heavily on observation in order to function, it is better to avoid of this resource updating interval.

As shown in Figure 5.18, sending notifications does not require handling on the Server side, but the round trip time of 1000 ms still stands out. It could indicate that network transfer has ignificant impact on this characteristic.

In order to evaluate scalability, an additional Controller is added to the current network. This Controller is set to send a GET and then PUT request to /player_move resource on the Model component, so the respective Controller behaviour is the same as in the first evaluation with the same request sending frequency.

In the Figure 5.19, the yellow line is from the GET response time in figure 5.2. In general, it shows the lowest response time for every tested request sending frequency except 13 requests/s. The blue line is drawn from the data gathered through an iPhone 6s, and the red line from an iPad Air. It appears at certain points that the two devices were in a race condition (at frequency 20 - 50 requests/s), whereas at other times they

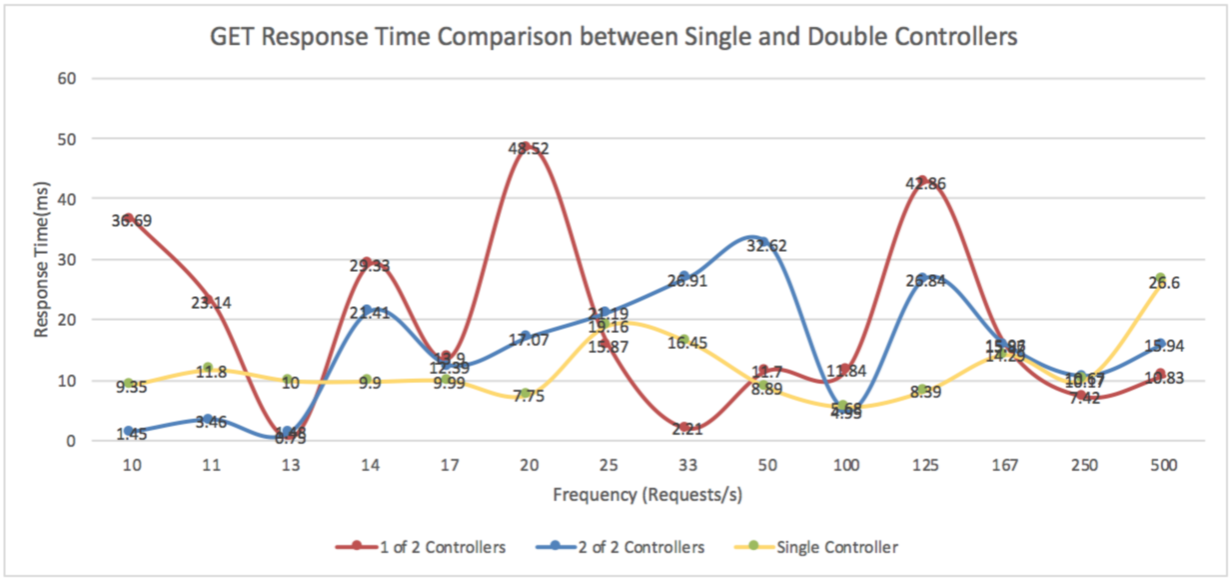


Figure 5.19: GET Response Time Comparison between Single and Two Controllers

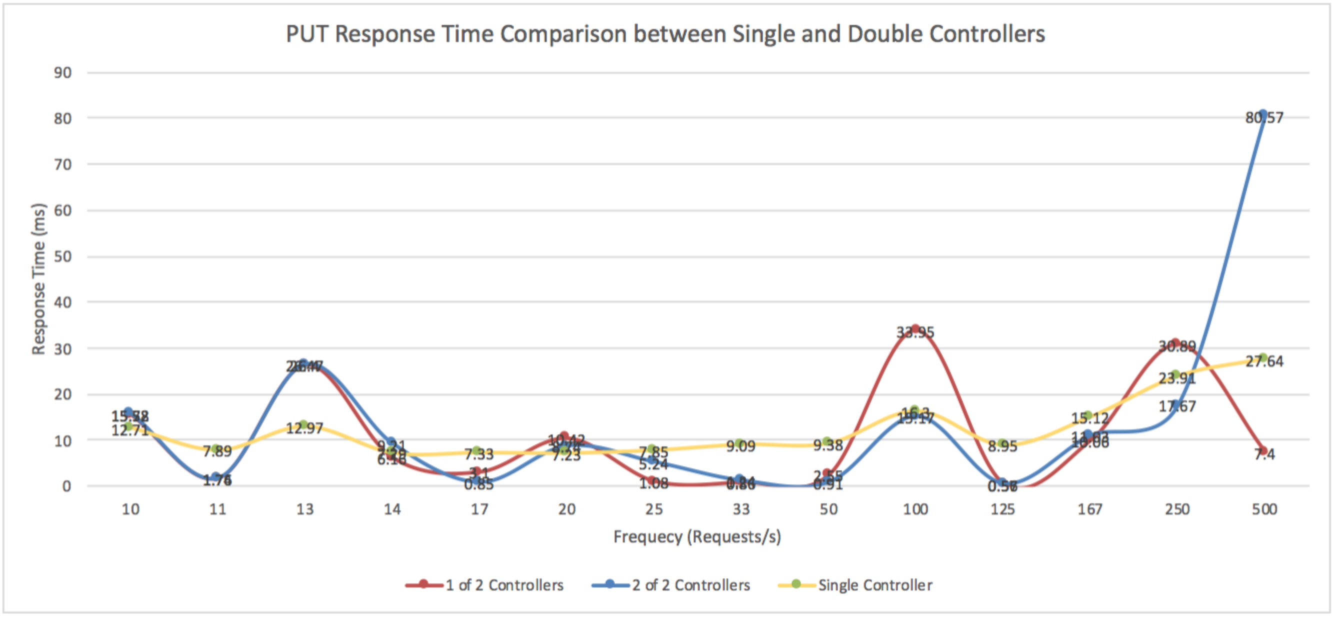


Figure 5.20: PUT Response Time Comparison between Single and Two Controllers

have similar trends (100 - 500 requests/s). Overall the maximum response time for the double-Controller case is significantly more than the single Controller scenario, and the minimum cases are also much lower. The differences are quite obvious, but again, this if for GET, so it has minimal impact on performance when running this prototype.

In the Figure 5.20, the response times have similar trend lines with the double-Controller case but they fluctuates more and their maximum response time can be 10 times as much as the Single Controller case. The 80.57 ms response time at 500 requests/s might be caused by a request timed out.

An additional point to note is that the Frame Per Second (FPS) rate is measured during notification evaluation, andit does not drop from 30 under any frequency settings, which is the default rendering loop for iOS applications by Unity.

5.4 Summary

In this chapter, the effectiveness of the implemented prototype is evaluated mainly under two settings: various request sending frequencies and different payload sizes. The aspects evaluated are response time and throughput for two main request methods: GET and PUT, as well as for a subscription request, and the round trip time for notification. In the final stage, an extra Controller was added to find out more about the scalability of the architecture.

For different request sending frequencies, lower frequencies generated a stabler response time for both GET and PUT, although the worst case scenario is better for PUT. As for payloads, subscriptions and notifications both have certain settings that need to be avoided, other results fall in a rather small range. As for the double-Controller case, PUT is better than GET, and lower frequencies are better than higher frequencies. In generala 100 ms is the limit for a user to feel that the system is reacting instantaneously, and 1 second is the limit for the user to notice the delay, but the flow of thought of a user is kept uninterrupted. Most of the performance falls is instantaneous with a few exceptions.

Chapter 6

Conclusion

Nowadays, individuals on average own enough devices to form a personal device cloud. In order to utilize fully the idle resources, several questions can be asked:

- How can components of an application be spread over multiple mobile devices?
- How can distributed components coordinate with each other?
- How effective are the distributed components working as one application?

To answer these questions, an architecture was proposed. The main concept of the architecture is to encapsulate different business logics of a game or visualization application as modules, distribute them over several mobile devices, and expose their functions as micro-services for each other to consume, while they communicate using the low-overhead protocol. This architecture is flexible, in that the number of participating mobile devices and players can vary, and by modifying the number and function of distributed services, the proposed architecture can cater to a different number of devices and both single- or multi-player scenarios. Furthermore, the pattern of a distributed MVC is adopted for implementing the distributed modules, the View, Controller and Model. The View is responsible for rendering the game screen, the Controller is for gathering user input and feed requests to the Model, and finally, the Model, is for storing and syncing game data between the Controller and the View.

In order to demonstrate such architecture, a prototype with one level that adopted the game mechanics of a simple Sokoban game was implemented. The components communicate with each other using CoAP protocol over a wireless network. To evaluate the effectiveness of the proposed architecture, evaluations of three categories were conducted. The main factors evaluated include a response time and throughput for key request methods and subscription requests under various request sending frequencies and payload sizes, round trip time for notification messages, and finally, scaling up with extra Controller components.

To conclude, the main contribution of the research is a proposed design to break up game applications and distribute them over a device cloud, and an evaluation of the usability of the design in a small, modest context with little overhead.

Chapter 7

Future Work

For future work, the following are some features that could be added to the prototype implemented in this research in order to advance it as a standalone application.

1. Additional Levels: The current prototype only contains one level. Adding more levels will add to its completeness as a game.

2. Save and Load: Functions same as optional services mentioned in the Architecture Chapter. Can then store game data locally or upload to a separate server.

3. Loading Resources at Runtime: In some situations, it is useful to make an asset available to a project without loading it in as part of a scene. For example, there may be a character or other object that can appear in any scene of the game but which will be used infrequently, for example, a high score alert. These assets may be stored in a separate local file or URL to reduce initial download time or to allow for interchangeable game content.

4. Network Infrastructure: For now the prototype is running on a wireless network. To branch out and incorporate wireless personal area network technologies, like BLE and Wi-Fi Direct, would add to its flexibility when Wi-Fi is not available.

As for the architecture proposed, more types of applications can be built to explore its possibilities.

- Education Application: An application showcasing the combination of the proposed architecture and education (e.g., as shown in Figure 1.5) preferably with collaborative features could be developed.

References

[1] Clark C Abt. *Serious games.* University Press of America, 1987.

[2] Autodesk. 3ds max. https://www.autodesk.ca/en/products/3ds-max/overview.

[3] Autodesk. Maya. https://www.autodesk.ca/en/products/maya.

[4] Chase Buckle. Digital consumers own 3.64 connected devices. https://blog.globalwebindex.net/chart-of-the-day/digital-consumers-own-3-64-connected-devices/.

[5] D. Camps-Mur, A. Garcia-Saavedra, and P. Serrano. Device-to-device communications with wi-fi direct: overview and experimentation. *IEEE Wireless Communications*, 20(3):96–104, June 2013.

[6] N. Chen, X. Li, and R. Deters. Collaboration amp; mobile cloud-computing: Using coap to enable resource-sharing between clouds of mobile devices. In *2015 IEEE Conference on Collaboration and Internet Computing (CIC)*, pages 119–124, Oct 2015.

[7] Jason H. Christensen. Using restful web-services and cloud computing to create next generation mobile applications. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 627–634, New York, NY, USA, 2009. ACM.

[8] J. Craighead. Distributed, game-based, intelligent tutoring systems - the next step in computer based training? In *2008 International Symposium on Collaborative Technologies and Systems*, pages 247–256, May 2008.

[9] Hoang T. Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 13(18):1587–1611, 2013.

[10] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000.

[11] Blender Foundation. Blender. https://www.blender.org/.

[12] Martin Fowler. GUI architectures. *MVP.[Online] http://martinfowler. com/eaaDev/uiArchs. html*, 2006.

[13] Martin Fowler and James Lewis. Microservices. *Viittattu*, 28:2015, 2014.

[14] Karl-Ingo Friese, Marc Herrlich, and Franz-Erich Wolter. Using game engines for visualization in scientific applications. In *New Frontiers for Entertainment Computing*, pages 11–22. Springer, 2008.

[15] MAXON Computer GmbH. Cinema 4d. https://www.maxon.net/en/products/cinema-4d/overview/.

[16] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[17] Google. The go programming language. https://golang.org/.

[18] Khronos Group. OpenGL ES - The Standard for Embedded Accelerated 3D Graphics. https://www.khronos.org/opengles/.

[19] K. Hartke. Rfc 7641 observing resources in the constrained application protocol (coap). https://tools.ietf.org/html/rfc7641, September 2015.

[20] Sang Ho. Na, jun-young park, eui–nam huh, personal cloud computing security framework. In *Proc. Service Computing Conference (APSSC) IEEE publication*, pages 671–675, 2010.

[21] Jantina Huizenga, Wilfried Admiraal, Sanne Akkerman, and Geert Ten Dam. Learning History by Playing a Mobile City Game. In *Young researchers furthering development of TEL research in Central and Eastern Europe*, Sofia, Bulgaria, 2007.

[22] Tom Huston. What is microservices architecture? https://smartbear.com/learn/api-design/what-are-microservices/.

[23] Aswin Indraprastha and Michihiko Shinozaki. The investigation on using unity3d game engine in urban design study. *Journal of ICT Research and Applications*, 3(1):1–18, 2009.

[24] D. Maggiorini, L. A. Ripamonti, E. Zanon, A. Bujari, and C. E. Palazzi. Smash: A distributed game engine architecture. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 196–201, June 2016.

[25] Matthias Kovatsch, Dominique Im Obersteg, and Daniel Pauli, ETH Zurich. Californium. https://www.eclipse.org/californium/.

[26] J. Moloney and L. Harvey. Visualization and 'auralization' of architectural design in a game engine based collaborative virtual environment. In *Proceedings. Eighth International Conference on Information Visualisation, 2004. IV 2004.*, pages 827–832, July 2004.

[27] Hirohiko Nakamiya. Monster sokoban. http://miya.s16.xrea.com/selection/bannin/.

[28] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015.

[29] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: Making the right architectural decision. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM.

[30] Pluralsight. Unity, source 2, unreal engine 4, or cryengine - which game engine should i choose? http://blog.digitaltutors.com/unity-udk-cryengine-game-engine-choose/, March 2015.

[31] Alex Rodriguez. Restful web services: The basics. *IBM developerWorks*, 2008.

[32] Kristopher Sandoval. Writing microservices in go. http://nordicapis.com/writing-microservices-in-go/.

[33] Tidalwave s.a.s. Beyond mvc: better ui design with pac, presentation model and dci. http://tidalwave.it/fabrizio/blog/beyond-mvc-pac-presentation-model-dci/, 2012.

[34] Jim Schaad. CoAP.NET - A CoAP framework in C#. https://github.com/Com-AugustCellars/CoAP-CSharp.

[35] Zach Shelby, Klaus Hartke, and Carsten Bormann. Rfc 7252 - the constrained application protocol (coap). https://tools.ietf.org/html/rfc7252, 2014.

[36] Bluetooth Special Interest Group (SIG). Bluetooth smart technology: Powering the internet of things. https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy.

[37] IEEE 802.11-2007 Standard. Wireless lan medium access control (mac) and physical layer (phy) specifications, 2007.

[38] A. Syromiatnikov and D. Weyns. A journey through the land of model-view-* design patterns. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 21–30, April 2014.

[39] CACE Technologies. Wireshark. https://www.wireshark.org/.

[40] Unity Technologies. Coroutines in unity. https://docs.unity3d.com/Manual/Coroutines.html.

[41] Unity Technologies. Introduction to components. https://docs.unity3d.com/Manual/Components.html.

[42] Unity Technologies. Unity. http://unity3d.com.

[43] Unity Technologies. Unity as the leading global game industry software. https://unity3d.com/public-relations/.

[44] Jeff Ward. What is a game engine? http://www.gamecareerguide.com/features/529/what-is-a-game.php.

[45] Burkhard C Wünsche, Blazej Kot, Andrew Gits, Robert Amor, and John Hosking. A framework for game engine based visualisations. In *in Proceedings of Image and Vision Computing New Zealand 2005, Nov. 2005.[Online]. Available: http://www. cs. auckland. ac. nz/ burkhard/Publications/IVCNZ05 WuenscheKotEtAl. pdf*. Citeseer, 2005.

[46] Xamarin. Xamarin. https://www.xamarin.com/.

[47] Xamarin and the Mono community. Monodevelop. http://www.monodevelop.com/.

[48] Hong Zhou, Huamin Qu, Yingcai Wu, and Ming-Yuen Chan. Volume visualization on mobile devices. In *14th Pacific conference on computer graphics and applications*, pages 76–84. Citeseer, 2006.

[49] Michael Zyda. From visual simulation to virtual reality to games. *Computer*, 38(9):25–32, 2005.