n-Screen Application Framework

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

By
Xiaobo Zhang

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
Canada
S7N 5C9

Abstract

Smartphones and tablets with advanced computing ability and connectivity have already become indispensable in our daily lives. As operating systems of these computer-like handheld devices are getting more mature and stable, many users want physically separated devices to interact with one another and with shared resources in real time. Those devices may have the same type of operating systems, such as sharing between android smartphone and tablets. However, sometimes the sharing occurs among different operating systems. A user may want to use a smartphone to control the menu while the image presentation is displaying on the Internet Protocol television (IPTV), as well as the audio on a personal computer. This scenario brings about the motivation of this thesis.

This thesis proposes an architecture that allows for sharing resources among many devices with separated screens at real-time. Compared with traditional mobile application framework, instead of the user experience on a specific device, the consistent user experience across multiple devices becomes the key concern. This research introduces a novel approach to implement the classical Model-View-Controller (MVC) framework in a distributed manner with a multi-layered distributed controller. To ensure consistent user experiences across multiple devices with different platforms, this research also adopts a channel-based Publish/Subscribe with effective server push state synchronization. The experiments evaluate the portability, message travelling latency improvement and bandwidth optimization. The results of those experiments prove the advantages of the n-Screen Application Framework (NSAF) both in portability that allows deployment on multiple devices from different manufacturers and performance improvement (both in latency and bandwidth consumption) while comparing with traditional data dissemination scenarios.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor, Professor Dr. Ralph Deters for the continuous support of my Master study and research, for his patience, motivation, enthusiasm, and immense knowledge throughout my entire three years of study. Besides my advisor, I would like to than the rest of my thesis committee: Prof. Dr. Julita Vassileva, Prof. Dr. Jim Greer and Prof. Dr. Daniel Chen for their valuable advices and insightful suggestions. My thanks also go to Ms. Jan Thompson and Ms. Gwen Lancaster, the former and current Graduate Correspondent at the Department of Computer Science, who has been very helpful throughout my study at University of Saskatchewan and very kind.

I would like to thank all of my friends for their unconditional love and selfless support. Last, but not least, I would like to thank my wife Jun for her understanding and love during the past few years. Her support and encouragement was in the end what made this thesis possible. My parents, Jingming and Huiqin, receive my deepest gratitude and love for their dedication and the many years of support of my studies both in China and Canada and that provided the foundation of all of my work.

Contents

List of Tables

List of Figures

List of Abbreviations

CCS	Cascading Style Sheets
LOF	List of Figures
LOT	List of Tables
GUI	Graphic User Interface
GUID	Global Unique Identifier
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JVM	Java Virtual Machine
MMVC	Message-based Model-View-Controller
MVC	Model-View-Controller
NSAF	n-Screen Application Framework
OSI	Open Systems Interconnection
OWD	One-Way Delay
P/S	Publish/Subscribe
RMI	Remote Method Invocation
ROA	Resource Oriented Architecture
RPC	Remote Procedure Call
RTT	Round Trip Time
SOA	Service Oriented Architecture
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WS	Web Services
WWW	World Wide Web

Chapter 1

Introduction & Problem Definition

1.1 Background and Motivation

Smartphones with advanced computing ability and connectivity have already become indispensable in our daily lives. The recent Canalys report [6] reveals that the smartphone market exceeded 100 million units sold in the fourth quarter of 2010. For media tablets, the Gartner research [17] expects that the worldwide sales will increase from 18 million in 2010 to 300 million in 2015.

As operating systems of these computer-like handheld devices are getting more mature and stable, many users are starting to have multiple computer-like devices at home or in office. A new issue emerges: "how do we combine multiple devices together in a single application with shared resources in real-time?". Those devices may have the same type of operating systems, such as sharing between android smartphone and tablets. However, sometimes the sharing occurs among different operating systems. Figure 1.1 is an example demonstrating a smartphone controlling the video list, meanwhile an IPTV displaying the video and a tablet showing the description of the video. A user may want to use a smartphone to control the menu while the image presentation is displaying on the IPTV, as well as the audio on a personal computer. This scenario brings about the motivation of this thesis.

1.2 Problem Definition

This thesis aims to build an architecture that allows sharing resources among many devices with separated screens in real-time. Additionally, as one device updates the resource, the system will propagate the new status among all other devices that are currently sharing the resource. In order to achieve this goal, a distributed system must be designed and implemented. To make this system simple and stable, and open to extension, I adopt the Model-View-Controller (MVC) pattern, a design pattern for user-facing software, as the architecture design pattern. The MVC pattern is a classic architecture design pattern, which separates the different elements of an application (the data model, the user action and the presentation) to provide a loose coupling between those elements. The central idea behind MVC is code re-usability and separation of concerns [2], that is, each component can be easily reused, modified or replaced. Nevertheless, the employment of MVC brings two key problems this thesis tries to solve:

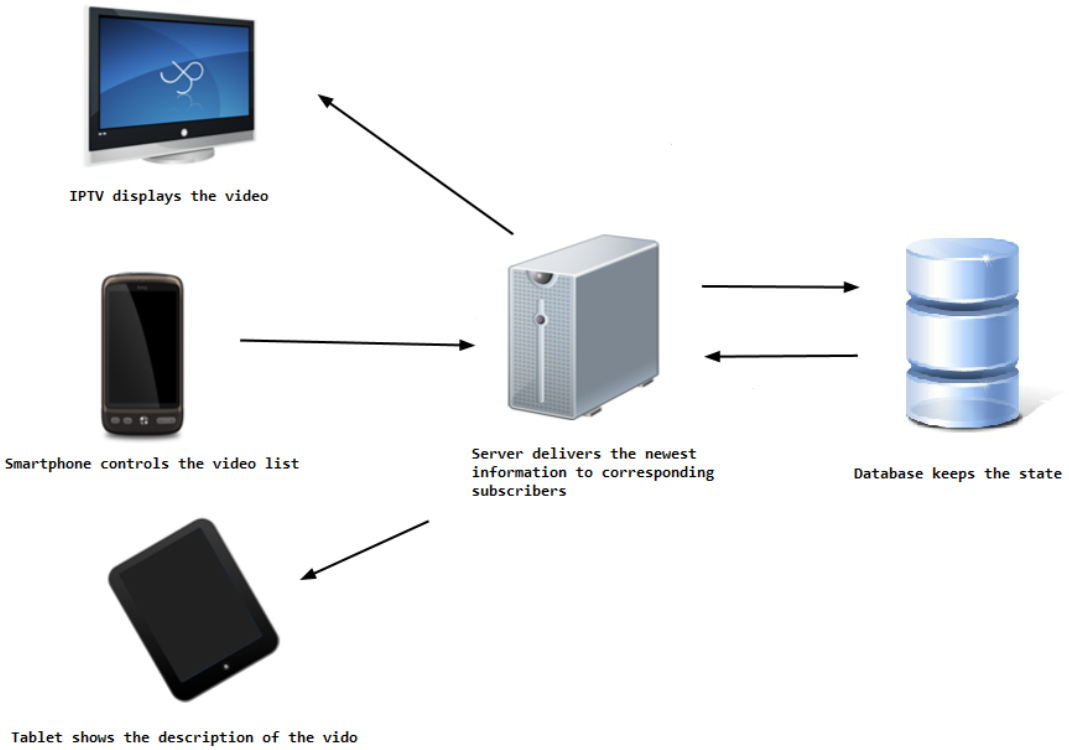


Figure 1.1: Example of n-Screen Application Framework (NSAF) in Daily Life

- How to distribute the MVC framework among multiple devices? (E.g. clients and server)
- How to effectively communicate between those distributed components and synchronize the application state and resources among them in real-time?

The key **challenges** are:

Challenge 1. Distributed MVC Architecture: The system is implemented in a distributed way. For building a distributed application with consistent data sharing and efficient data transmittion, it is a challenge to organize and distribute the resource data, application logic and functions, which are three basic components of classic MVC architecture pattern, to many physically separated devices.

Challenge 2. State Consistency: The application should not only share computation objects, but also be capable to provide minimum time latency user experience among different devices in real-time. To be more specific, when there is a state changing action on the screen of one device, it is necessary for the server to receive the action, analyze it, find out client-side devices which need to change their display, and propagate the changed information to those devices. This synchronization step should be finished as quickly as possible. Upon receiving the changed information, the devices should be able to update their display automatically without any user action (e.g., pressing a refresh button).

Challenge 3. Bandwidth: There are various types of client side devices (e.g., smartphones, tablets, personal computers, IPTV, etc.). Some of these devices are connected via wired network, whereas some

of them are connected via wireless or cell network and act as nomadic agent when the user carries the device. Devices connected via wireless or cell network have limited bandwidth. In order to consume the least amount of bandwidth, the chosen deliver scheme should be able to organize the messages propagated to client-side devices and reduce the waste in network bandwidth.

Challenge 4. Web Standard Compatibility: Unlike desktop computers, which are running with a few mainstream platforms (e.g. Windows and MacOS), the mobile devices are normally running with various platforms (such as Android, iOS, BlackBerry OS and etc.) or the same type of platform but in different versions (e.g. Android 2.2, Android 2.3, Android 3.0, and etc.). This fact raises the question: How to enable the application to support as many devices as possible with minimum effort?

1.3 N-Screen Application Framework (NSAF)

With the aim of overcoming the challenges listed in the last section, this work proposes an n-Screen Application Framework (NSAF) to properly leverage the physical advantages from different devices running with various platforms in a single application. The NSAF framework adopts the idea of classic MVC architecture design pattern, but in a distributed way. As shown in Figure 1.2, it treats the whole application over multiple devices including the network connection as a large, single and independent MVC framework. The detail design of NSAF, as well as how NSAF solves the challenges listed in last section, will be explained in Chapter 4.

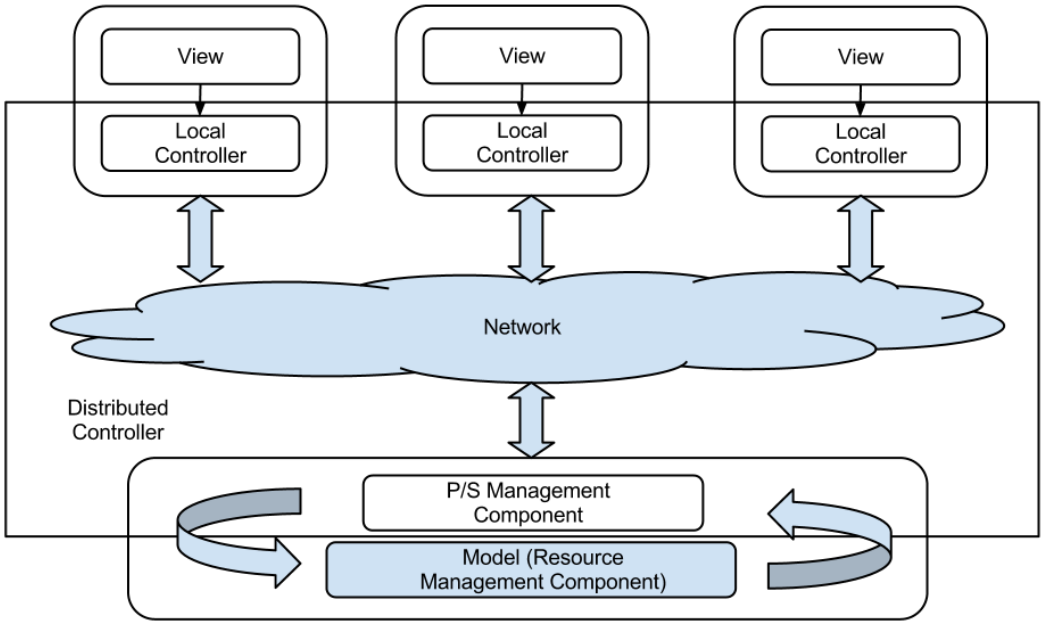


Figure 1.2: The MVC view of NSAF Architecture

1.4 Research Goals

The goal of this research is *to build a distributed MVC framework and synchronize the application state and resources among components in real-time.* This leads to the following research questions:

- How to distribute classic MVC framework among multiple devices?
- How to communicate among those distributed components and synchronize the application state and resources in real-time?
- How to enable the application to support as many devices as possible with minimum overhead?

This research will explore those questions as well as investigate the costs and benefits of different approaches to support real-time state synchronization among distributed MVC components.

1.5 Structure of Thesis

This thesis is organized as follows: Chapter 2 presents an overview of the relevant literature in the area of distributed model-view-controller pattern, web service and data dissemination technology. Chapter 3 describes the conceptual design and architecture of this n-Screen application. The experiment, result and evaluation are described and analysis in Chapter 4. Chapter 5 summarizes the thesis and gives an outline for some possible improvements and research ideas. The implementation of the proposed approach is presented in Appendix A.

Chapter 2

Literature Review

This chapter reviews related research works in the following fields: Distributed Model-View-Controller (MVC) architecture, mobile Web Service (WS) and state synchronization technologies.

2.1 Overview of MVC Approaches

Nowadays, there are many high-level patterns and principles available for software development. These systems are often referred as architecture styles or architecture patterns. A well-defined architectural style normally suggests a vocabulary of component and connector types, a topology of interconnection, and a control flow strategy [30]. It provides the software developer a clear abstract framework of the system under development, provokes appropriate questions during the early design during the software construction process, as well as giving proper answers to those questions directly [8].

The MVC architecture pattern is an ideal solution to build a system that is capable to organize views on different devices and connect them with shared data, as well as to provide an elegant, simple and stable architecture. As mentioned in the previous chapter, since different devices communicate to each other over the network, certain changes are needed to reform the classic MVC framework, which usually works on a single computer, to a distributed one. The MVC, first described by Trygve Reenskaug in 1979 [34], is an essential design pattern for interactive applications since it provides independence to each component in the software.

2.1.1 Classic MVC Architecture

The classic MVC pattern separates the different elements of the application (the modeling of the external world, the action from the user, and the presentation to the user), while providing a loose coupling between those elements. Therefore, each component can easily be reused, modified or replaced. For example, a design change in the application's Graphic User Interface (GUI) does not require a corresponding change in the data model. In classic MVC pattern, the communication across elements is implemented by method calls. If two components are needed to be tightly linked (e.g., View - Controller link in most cases), instance variables are usually used to maintain this tight coupling. On the other hand, if the component (e.g., Model in most cases) needs to be independent and loosely linked to the MVC Triad, special register mechanisms, such as

Observer pattern, are usually adopted to record the dependencies. The classic MVC framework includes the following components 2.1:

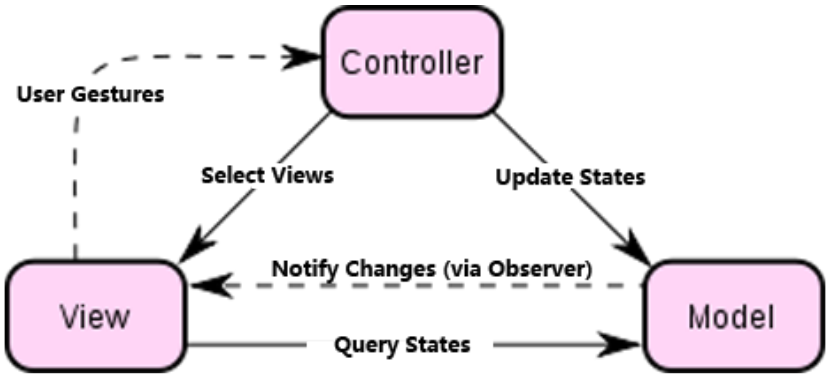


Figure 2.1: Model-View-Controller Concept

- **Model** represents the data of the application domain and manages the rules that govern access to and updates of this data. It encapsulates the application state, responds state query request (usually those requests are from the view), as well as takes actions and changes states (usually from the controller). When the state is changed, the model notifies views of the change. According to Reenskaug's report[34], the model should be a passive holder and manipulated by the view and the controller. Unlike the view and the controller, which usually keeps an instance of the other component inside of its body, the model should be unaware of the existence of either the view or the controller and of its participation in an MVC triad. This isolation is not an artifact of the simplicity of the model, but of the fact that the model changes only at the behest of one of the other members of the triad [34]. Since it is an important component in the MVC Triad, all models cannot be completely isolated and passive. For example, if a model is changed without request from view or controller, it should be able to notify the corresponding view to update the representation. In this case, the dependency between model and view should be registered to keep the consistency between them.

- **View** manages the display of information, renders the content of models and allows controllers to select different views. Even with the same models, different views can specify different ways to present the content by highlighting certain attributes of the model and suppressing the others. It is thus acting as a presentation filter [34]. It is usually the view's responsibility to maintain the consistency with changes in the model. This consistency can be achieved by either a push mechanism, where the view registers itself with the model as the Observer pattern [14] for change notifications, or a pull model, where the view calls the model and requests the newest state when it is needed.

- **Controller** interprets the interactions from the user, often via a registered handler or callback, into actions to be performed by the model. In a stand-alone desktop application, the user interaction could be a mouse click or a keyboard button press; in a mobile application, that could be an incoming call

or a screen tap; in a web application, that could be a GET or a POST HTTP request. The controller translates those actions and informs the model to be updated as appropriate. Also, based on the user interaction and model updates, the controller could respond by selecting an appropriate view.

The goal of MVC is to increase the re-usability, flexibility, and maintainability of the code by decoupling models (data) and views (representation). There are two principal separations in the MVC patterns [12]: separating the presentation from the model and separating the controller from the view. The separation of models and views components allows different views to use the same model and increases the model's re-usability. It also enables the possibility to allow different users to share and collaborate simultaneously on the data model. The separation of views and controllers makes it easier to alter or add new presentations later on. Also, compared with traditional software architectures, MVC architecture can relatively easily accommodate major functional changes. To support a new client demand, developers can simply write new views and control logic, as well as connect them into the existing architecture.

The decoupling feature of MVC also provides a clear view and structure of software that helps developers to understand its architecture; however, it also introduces some extra implementation and new levels of indirection that increases the complexity of the solution [26].

2.1.2 Derivatives of MVC

There are a few derivatives of MVC software pattern, among which Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM) are targeted at modern UI design and development. Those derivatives usually keep the separation concept of the MVC pattern; however, some changes are applied in order to fit and support some of the contemporary technologies.

Model-View-Presenter (MVP)

As some UI-creation technologies (such as Windows Forms) become more and more powerful, the UI layer is able to handle more than it usually does. Some logics that belong in other layers of the application in traditional software patterns become part of UI layer's responsibility. MVP is a UI design pattern that is trying to improve the separation of concerns in presentation logic by introducing presenter object. It was first introduced in the 1990s at Taligent [31], which was a company producing an object-oriented operating system named Taligent. Later Microsoft began incorporating MVP into their documentation in the .NET framework [25]. The MVP patterns contain the following components:

- **Model:** As in the MVC pattern, the model component defines the data to be displayed. It is either a domain model that represents the application state content, or a data access layer that represents the application state content.
- **View:** The view component is responsible for displaying data from the model component, as well as route user commands (mouse click, button press and etc.) to the presenter component.

- **Presenter:** The presenter component retrieves data from model, subscribes model events and formats models to display in the view component.

Passive View versus Supervising Controller Since the view component is the display of the model component, when the model updates, the view component also needs to be updated to reflect the changes. In this situation, there are two types of MVP variants: Passive View and Supervising Controller [25]. In Passive View, since the Presenter component contains the UI business logic for view component, the view is considered to be as passive as possible (it contains almost zero logic) and forwards all interaction operations to the Presenter component. On the other hand, in Supervising Controller, for some simple data binding, the view can interact directly with the Model component without the intervention from the presenter component. The presenter updates the view only when there are some complex logic transactions involving.

Difference between MVP and MVC Although there are many similarities between MVC and MVP, MVP is an advanced form for MVC, several variations make them to two different design patterns. Unlike in MVC, where the controller handles user gestures and determines which view component is displayed in response to any events, the MVP view component handles the user gestures and routes those user events to corresponding presenter component to act upon the model. Also, in MVP, the view component is required to be as passive as possible and must delegate the presenter component. Otherwise, it will never get called. All interactions between model and view components must pass through presenter. However, in MVC, the relation between the M-V-C components is usually triangular. The view can query the model component directly. Finally, since presenter contains the UI logic of the view components and it talks to view component through interfaces, MVP allows unit testing by mocking of view component. The MVC has limited support to unit testing since the view component has some intelligence and is not completely passive.

Model-View-ViewModel (MVVM)

Model-View-ViewModel is another architectural design pattern that is largely based on MVC pattern. It targets at modern UI development platforms that support event driven programming. As in MVC and MVP pattern, MVVM is designed to facilitate the separation between the development of UI (View component) and the development of business logic (Model component).

In 2004, Martin Fowler published an article to describe a new pattern named Presentation Model(PM) [11]. The essence of this model is a Presentation Model that represents all the data and behaviour of the UI window, but without any of the control to render that UI on the screen. The view component simply projects the state of this Presentation Model. The Presentation Model makes all of the decisions needed for presentation and this leaves the view component to be as simple as possible. Later in 2005, John Gossman described the Model-View-ViewModel(MVVM) pattern on his blog [18], which is considered as identical to PM or a specialization of the more general PM patterns [36]. There are three elements included in MVVM pattern:

- **Model:** As in the MVC pattern, the model component represents the application state.

- **View:** As in the MVC pattern, the view component manages the display information.

- **ViewModel:** The ViewModel is considered a model of the view which means it is an abstraction of the view component. It serves as a mediator between the view and the model component, which updates changes from the model to the view and routes the view event to the model. However, unlike the Presenter component in MVP, a ViewModel does not need a reference to a view component [36]. In order to sense the state change, the view component binds to properties on a ViewModel. If property values in the ViewModel changed, they are automatically propagated to the view component via data binding technologies. When a user event happens in the view component (such as a button click), it is routed to the ViewModel and a command on the ViewModel executes to perform the request action.

MVVM is a very loosely coupled design. The relationship between MVVM is as follows: The view component does not keep a reference to the model component. The model component does not know both the view component and the ViewModel component at all. The ViewModel is the mediator between view and model component; however, it is unaware of the view component since it is the View's duty to connect to the ViewModel component.

2.1.3 Distributed MVC

As mentioned in the previous chapter, a key issue that we need to solve is resource sharing and synchronization among different devices. Because of this key issue, the whole system needs to be built in a distributed manner. The architecture styles reviewed in the previous section focus more directly on traditional application domain, while rarely discussing distributed implementation concerns which will largely influence the software architecture. The choice of how to implement the system in a distributed way will ultimately impact the system's performance and functionality.

Like the meaning of its words, components (M-V-C) in distributed MVC may be deployed on separated computer-like devices over the network. As a result, some traditional approaches of classic MVC, such as method-based communication between components, cannot be directly translated to the connector between distributed components [32]. Consequently, a new connector that combines different components and passes information among them must be designed. This issue is the primary concern of transforming classic MVC into a distributed way. It makes the design of the distributed MVC architecture more sophisticated. In order to link the distributed components together and provide an efficient communication mechanism, multiple technologies must be used. The content of the following paragraphs examines concepts about the distributed MVC system and the technologies that can be used to implement it.

Distributed System

In order to provide minimum time latency user experiences among different devices over the network, we need to build a loosely coupled Distributed MVC system that each component can be deployed on different physical devices. To achieve this goal, we have to face some fundamental challenges for distributed systems:

- **Unreliable Network Connection.** Distributed systems that have a central server to store information data need the server data to be shared among different client-side devices over the network, which means the data has to be transferred from one device to another across networks. This step is considered less reliable compared to different processes running on a single device and communicating via shared memory or messages. A distributed system has to be prepared to deal with a much larger set of transportation problems such as sudden loss of connectivity and inconsistent bandwidth.

- **Platforms are different.** As mentioned earlier, one main goal of this work is to provide seamless user experience among different devices, which needs to be able to transfer data between devices that are using different programming languages, different operating platforms, and even different data formats.

- **Minimize the cost of changing.** Since a distributed system operates on many different devices, the application updates are frequent and inevitable. Changes in one system may affect the others. In order to keep the service consistency among many devices, we must minimize the cost of changing - minimize the dependencies from one system to the other. That is, if one system changes, only minor changes or even no change is needed to other systems. Therefore, the system must follow the loose coupling design idea, which means each of its components has, or makes use of, little or entirely no knowledge of the existence of the other components.

Over time, some technologies have been developed to overcome the above challenges [20].

- **Shared Files:** The idea is similar to the shared memory mechanism among different processes on a single system. One application writes a file and the other reads it later. Applications need to agree with the policy of reading and writing the file and the format of the file. This is a relative slow solution compared with other technologies since it requires file transfer over the network.

- **Shared Database:** In this scenario, applications across the network can access the same single database. The problem is that no data can be directly exchanged between applications since there is no duplicate data storage and all applications can only have access to the data by connecting to the shared database. Security is also an issue since the single database is vital to the system.

- **Remote Procedure Call:** One application exposes some of its function calls to the network so it can be accessed remotely by other applications over the network. This procedure is usually a real-time and synchronous mechanism.

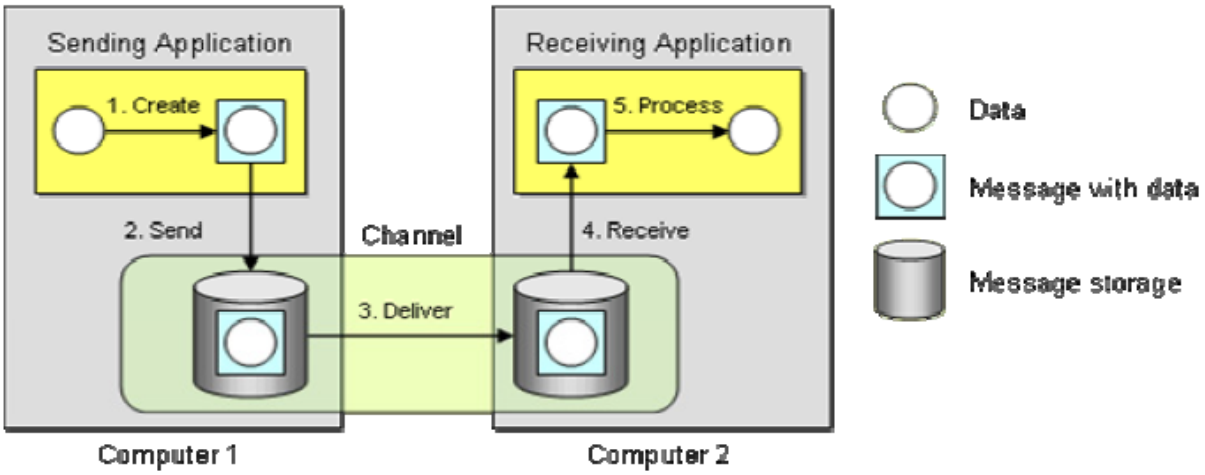


Figure 2.2: Message Transmission in Message System [20]

- **Messaging System:** When one application publishes a message to a shared message channel (queue), others can read the message from the channel at a later time. Those applications must register the same channel and agree on the format of the message.

Since I am talking about distributed MVC, I primarily focus on messaging system since it is an ideal technology to allow different components to quickly, safely and asynchronously communicate to each other.

Messaging System

Since components are in different devices, they need to use specific technology to communicate with each other in order to synchronize their states. There are several solutions that have been proposed in the literature and most seek to leverage the power of HTTP protocol [21]. A messaging system that can provide messaging functionalities (carrying messages from one component to another over the network) must be deployed. As shown in Figure 2.2, a message is transmitted in five steps: create, send, deliver, receive, and process. The sender application invoking messaging system needs to pass the data to the messaging system. The messaging system is then responsible for packaging the data into corresponding message format, moving the message from sender's computer to the receiver's computer, unpackaging the data and passing it to the receiver system.

According to Hohpe's book [20], there are two important concepts about a messaging system: (1) Send and forget: once the sender application sends the message to the message channel, it can continue with other work while the messaging system handles the transmission in the background. The sender application does not need to worry whether the transmission will be successful or not. (2) Store and forward: When the sender sends the message to the channel, the messaging system stores the message on the sender's computer and forwards it from the sender's computer to the receiver's, and then stores the message in the receiver's computer. To implement this concept, the messaging system should always carry a queue to send/receive

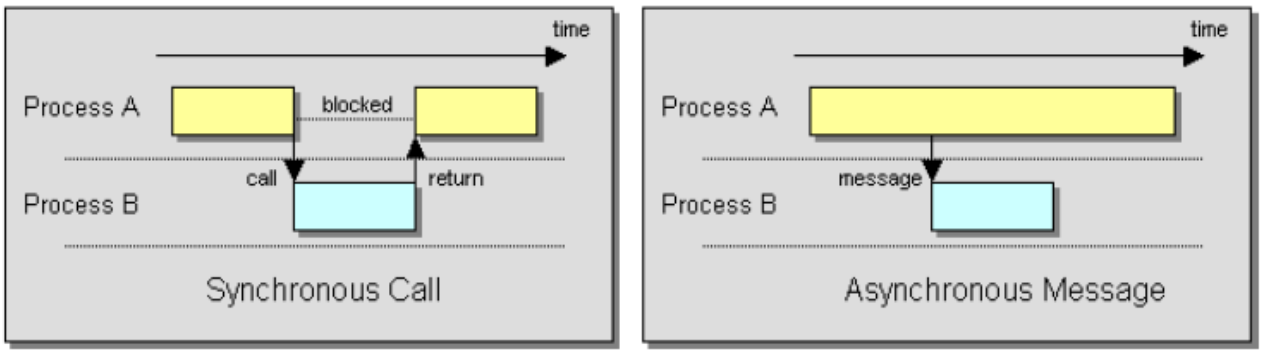


Figure 2.3: Synchronous and Asynchronous Call Semantics [20]

and store messages.

Compared with other technologies, the messaging system has the following advantages:

- **Asynchronous Communication:** As mentioned in the "send and forget"concept, once the sender sends the message to a messaging system, it does not have to wait for the messaging system to deliver the message. Once the data is sent to the messaging system, the sender is then free to perform other works. This is called Asynchronous Communication. On the other hand, if the sender needs to pause and wait for the acknowledgement or result back from the receiver, it is a Synchronous Communication. Figure 2.3 illustrates these two communication services. Since end users avoid continuously waiting for message transpiration over the network, asynchronous communication provides the possibility of seamless user experience, which is an important goal of this research.

- **Reliability:** Since it uses a "store and forward"mechanism, a messaging system provides more reliability in information delivery than RPC. Data is packaged as a message. The sender keeps a copy of the message during the sending. When the receiver receives the message, the receiver also stores the message. This process is assumed to be reliable compared with sending the message directly from the sender to the receiver without any storing action, since the network or the receiver may not be running properly. If the message is not properly delivered, the messaging system can resend the message until it succeeds.

- **Throttling:** Messaging system uses a queue to store incoming messages on the receiver side. In addition, asynchronous communication enables the receiver to control the rate at which it consumes the requests. This feature avoids the overload on the receiver side which is a very common problem for RPC with too many calls on a single receiver at the same time. Although this throttling can cause an adverse effect that the response may be not immediate and on time, the asynchronous communication can minimize this effect since the sender is not blocked waiting for the response.

- **Portability:** When multiple devices are communicated via remote communication, these devices are likely to use different systems, platforms, and languages, which are developed by diverse teams. A

messaging system can be a universal translator between those devices that work on independent systems and languages, yet allows them to communicate through the same messaging protocol.

Message-based Distributed MVC

As discussed above, a message-based system is a suitable and reliable mechanism for a distributed system to substitute classic method-based communications between components. It can provide loose coupling among components in the distributed system. The store and forwarding feature also avoids negative effect from fluctuating network connection.

Since communication patterns in Message-based Distributed MVC is different from the Method-based Classic MVC, the complexity of the system is decided by the inter-operating relationship between components, or, to be more specific, the relationship between model and view. A few researchers mentioned that the interactive patterns of model and views are one-to-one, one-to-many, and many-to-many [33]. Message-based Distributed MVC can be classified according to these interactive patterns and complexity of components. Xiaohong Qiu, Shrideep Pallickara, and Ahmet Uyar [33] propose Single Model Multiple View (SMMV) and Multiple Model Multiple View (MMVC) patterns. In Hamid Mcheick's and Yan Qi's work [24], they have Simple Message-based MVC (SM-MVC) and Complicated Message-based MVC (CM-MVC). Both SMMV and SM-MVC describe the architecture that has only one model but has more than one view. This pattern is widely used in client/server web applications with different types of clients accessing the same server. Different clients have their own ways to interpret data on the server. On the other hand, both the CM-MVC and MMVC have more than one Model that requires a more complicated connector. There is a minor difference between those two classification concepts: CM-MVC structure includes the possibility of many models utilizing one same view for presentation. However MMVC [24] only discusses architectures that contain more than one view and more than one model.

Figure 2.4 shows an approach to MMVC which utilizes Web Service(WS)[33] as the message API over the network. In this approach, MMVC is a service-oriented architecture (SOA) that separates the distributed system into two parts: the model (application logic) and the view (visual component). Because of WS, the model component naturally becomes an online service and the view represents the user interface to manipulate the service. In this example, the system also employs a double-linked multiple-stage pipeline model [33] that refines the system into small grained stages with bidirectional messages exchanging as the communication mechanism between two adjacent components. The benefit of this arrangement is it provides a well modularized structure with many decomposition possibilities.

Figure 2.5 shows another interesting approach in Adrian Hornsby's research [21] for MMVC using Extensible Messaging and Presence Protocol (XMPP). In this approach, there is a Virtual Model element that resides on the UI side. This Virtual Model element enables support for different clients who require different parts of the model. It is a subset of the server model element. The communication is accomplished via XMPP protocol, which is an open-standard, easily understandable, decentralized, extensible, and secure communica-

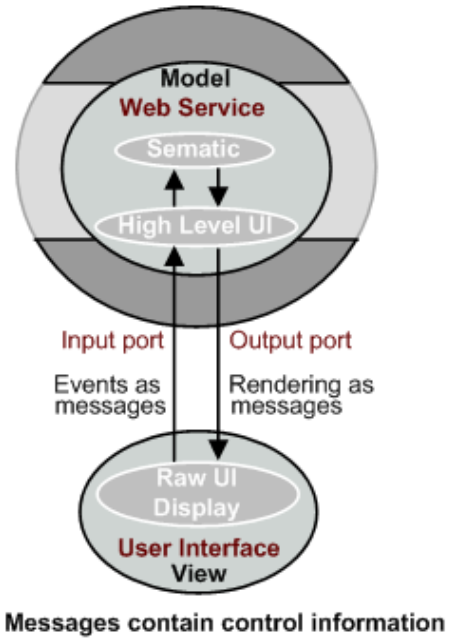


Figure 2.4: MMVC Model [33]

tion protocol for message-oriented middleware based on XML. The Internet Engineering Task Force (IETF) has formalized the XMPP as an approved instant messaging and presence technology. XMPP has been widely accepted by the software industry, from simple instant messaging software, such as Skype, Google Talk, Windows Live Messenger, to complex cloud computing systems. There are two Observer patterns [14] on the client side. One is used to monitor the incoming change of model component and the other is used for monitoring the local view component. The Observer pattern is a classical software design pattern containing two types of components: subjects and observers. A subject object maintains a list of its dependents which are observers. The subject notifies observers about its state changes automatically, usually by calling one of their methods. Observer design pattern is usually used in the classical MVC[14]. The benefit of the MMVC approach is that it divides a complex distributed MVC system into some smaller local MVC systems with real-time suitable message exchange protocol. This feature exemplifies the principle of software modularity and re-usability.

Publish-Subscribe Architecture Pattern

There are a few different communication mechanisms for distributed MVC: the classic method-based communication, a request/response scheme that can be either method-based or message-based and a message-based publish/subscribe pattern with a message transmission broker between view and model component.

The Publish/Subscribe (P/S) pattern expands based on Observer pattern [14] by adding the notion of a message channel for exchanging information via messages [20]. In P/S messaging system, each message subscriber needs to be notified of a particular message once. One subscriber should not notice the same

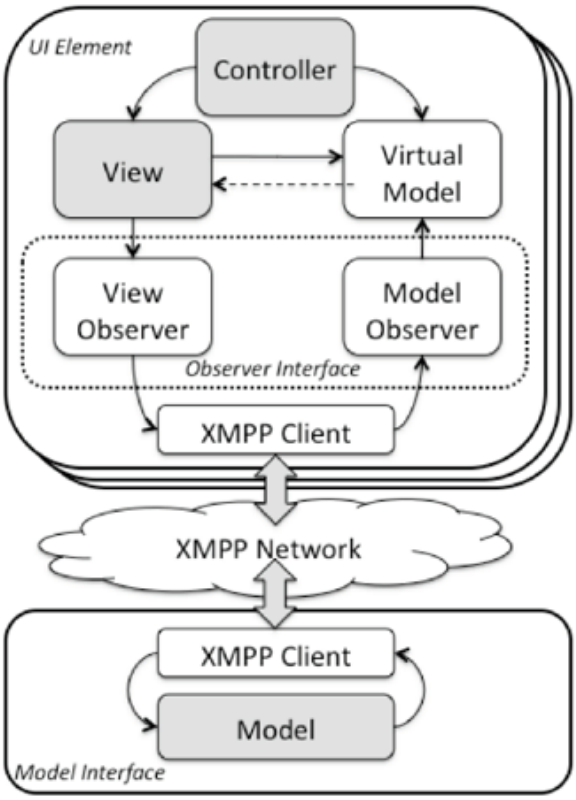


Figure 2.5: XMPP message based MVC model [21]

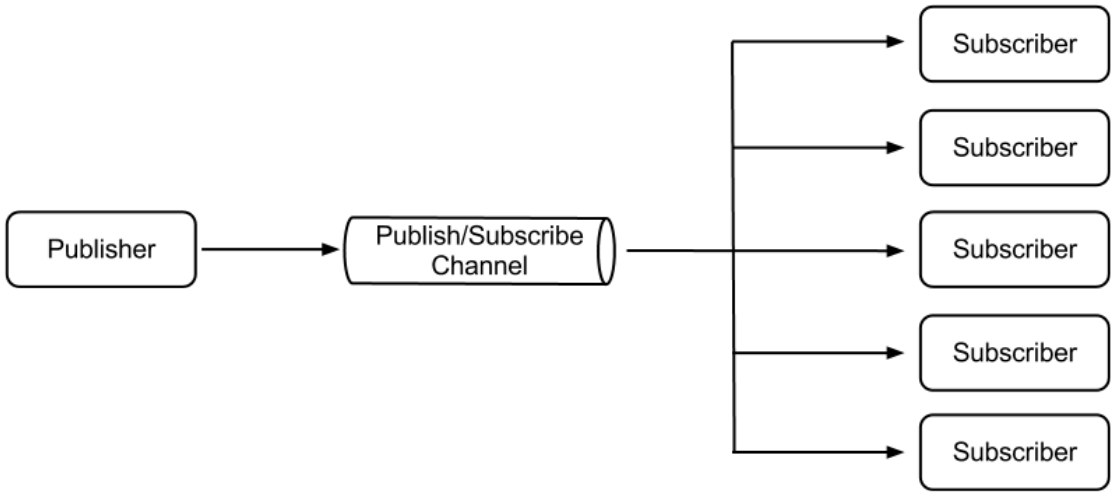


Figure 2.6: Publish Subscribe Pattern

message more than once. The message is not considered as successfully delivered until all the corresponding subscribers have received the message. Once all of the subscribers have received the message, the message is considered as consumed and should be deleted from the message channel. Figure 2.6 shows the passing scheme of P/S Pattern.

The P/S channel contains two parts: one input channel for the publisher and multiple output channels, one for each subscriber. The channel can be implemented as a queue structure. A message is considered to be consumed if it pops out of the queue. When a message is published into the channel, the P/S channel delivers a copy of the message to each of the downstream output channel. Each output channel can only have one subscriber, which is only allowed to consume a message once. In this way, each message must be delivered to any subscriber once and when the message reaches the subscriber, its copy is immediately disappeared in the channel.

According to Mcheick and Qi's work [24], a qualified distributed connector needs to link distributed components by using some formatted information according to the dependency (relationship) between components. In order to provide loose coupling among distributed components, the connector must be able to transport information, encode information into agreeable format messages, as well as describe the dependency between messages. The Publish-Subscribe Architecture Pattern is a very suitable messaging pattern for designing such connector. It is a very loosely coupled architecture. The publisher simply pushes messages into the channel and does not need to know the subscribers as long as this P/S messaging system is between them. On the other side, subscribers also do not need to know who the publisher is. They only receive the messages from the channel they have subscribed in the proxy. Also, since P/S is a design pattern of messaging system, it has all the advantages the messaging system has, such as portability, throttling, reliability and asynchronous communication. Data sent to the messaging system can be encoded with certain message format and delivered to corresponding subscribers.

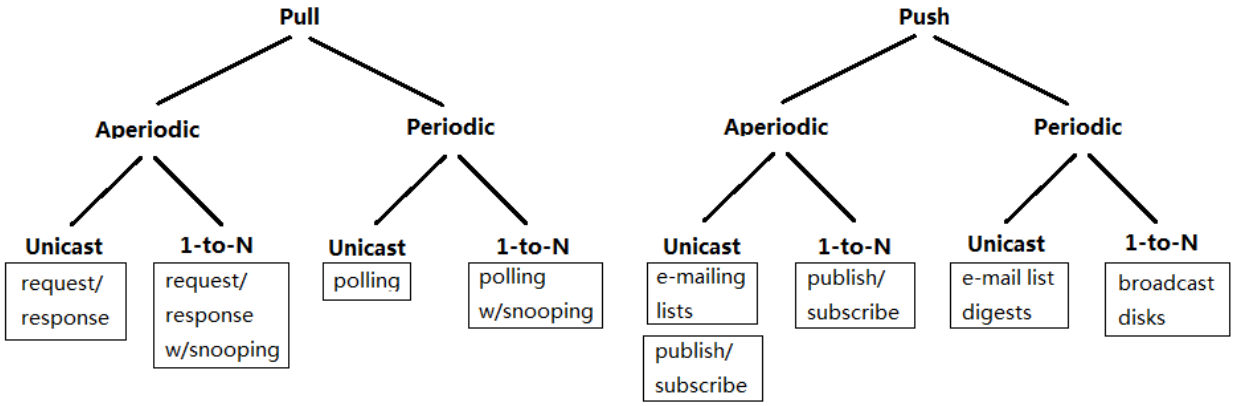


Figure 2.7: Category of Information Dissemination [13]

In conclusion, a distributed architecture receives considerable benefit from P/S messaging pattern; however, it cannot be directly deployed in distributed system without a proper transportation mechanism. The next section introduces a few data dissemination mechanism and protocols. A bidirectional communication protocol named WebSocket is used for the Publisher/Subscriber messaging system.

2.2 Information Dissemination

The architecture of software is like the upper level of a building. To support those upper level designs of the system, as well as provide state synchronization among the distributed components, some information dissemination mechanisms must be taken into consideration. The objective of information dissemination is to deliver data content from the information producer to interested data consumers. Even though there are various implementations of information dissemination, there are always three fundamental functions that need to be presented: making new content available, updating existing content, and revoking obsolete content [27].

2.2.1 Fundamental Properties

There are several characteristics that can be used to categorize information dissemination and compare the data delivery mechanisms [13]: client pull vs. server push, aperiodic vs. periodic, and uni-cast vs. 1-to-N. Although there are some other interesting characteristics that might need to be taken into account, such as fault-tolerant, ordering guarantees, etc., these three characteristics are fundamental components to make intelligent choices about delivery mechanisms for specific situations [13]. Figure 2.7 shows these characteristics and several common examples for each. In these three characteristics, client pull vs. server push is the one that has been discussed the most. In the following part, we will see the other two characteristics first, and then proceed to the client pull vs. server push.

Aperiodic vs. Periodic

Both push and pull approaches can use either periodic or aperiodic communication. In periodic communication, the information provider and consumer exchange information according to some pre-arranged schedule. This schedule can be a fixed pre-defined time interval, or be generated by some degree of randomness [5]. In contrast, aperiodic communication is event driven. Neither the data provider nor the consumer can pre-define the times at which they exchange information. This event may be a client side data request caused by user action for client pull or a server side data update for server push.

Unicast vs. 1-to-N

In unicast communication, data items are sent from a single data holder to another machine. On the other hand, 1-to-N communication allows multiple machines receive the data from one data source.

Client pull vs. Server push

Client pull vs. Server push is a basic characteristic to distinguish information dissemination mechanisms. Historically, web applications need a bidirectional communication between a client and a server. The general idea of client-server web application is that the content updates on the client-side application have always been driven by client request. This method is called client pull. However, this technology results in a variety of problems:

- The client pull cannot achieve real-time data update and event notification, because the client-server conversation is always initialized by the client-side application. Although the developer can set up the client to try to pull information from the server more frequently with a shorter time interval, this approach leads to problems that it consumes more resources, such as network bandwidth, CPU power, mobile device battery life, and etc.

- For web applications that need bidirectional communication between the client and the server, such as P2P software, real-time communication applications and Internet games, client pull forces the server to use a number of different underlying TCP connections on each client: one connection for sending the information to others, and the other for incoming messages.

Push technology is an essential part of modern web applications. In contrast with pull technology where the receiver or client initializes the connection and requests the resource, the request of a given transaction is initiated by the service publisher or central server. Push technology provides significant reduction in data latency. Once a data update event occurs at the server side, the message can flow from the server to the client without sending a request from the client side for each date update.

2.2.2 W3C Compliant Information Dissemination Approaches

AJAX

AJAX [16] is a combination of several existing web development technologies used on the client-side to create asynchronous web applications. Those web development methods include:

- standards-based presentation using XHTML and CSS;
- dynamic display and interaction using the Document Object Model (DOM);
- data interchange and manipulation using XML and XSLT;
- asynchronous data retrieval using XMLHttpRequest;
- and JavaScript binding everything together.

XMLHttpRequest (XHR) is an API available in most modern web browser scripting engines such as JavaScript and VBScript. It can be used to send HTTP requests directly to a web server and load the server response data back into the JavaScript. This feature allows it to establish an independent communication channel in the background between a web client and server [7]. The response data can then be directly used to alter the DOM of the currently active document in a browser window without loading or refreshing the web page.

HTTP Pull

The simplest approach of achieving high data accuracy and data freshness is to periodically request incremental updates in the background. This procedure is usually done by polling the server during every user-definable time interval known as Time to Refresh (TTR).

In order to provide the newest updated data on time, there is a trade-off between the time interval and data latency. If the client checks the updated data too frequently, it will cost more network bandwidth, as well as possibly cause unnecessary message handling on both client and server sides. On the other hand, if the client side application checks the update less frequently thereby decreasing the cost of bandwidth, the data might not be updated on time.

In order to provide the newest updated data on time, there is a trade-off between the time interval and data latency. If the client checks the update data too frequently, it will cost more network bandwidth, as well as possibly cause unnecessary message handling on both client and server sides. On the other hand, if the client side application checks the update less frequently thereby decreasing the cost of bandwidth, the data might not be updated on time.

HTTP Streaming

HTTP Streaming is also known as HTTP server push. The idea was first proposed in 1992 by Netscape [29]. It solves the issue of periodic pulling by streaming server data in the response of a long-lived HTTP connection. Normal HTTP pull opens a connection between client and server, sends a response from server, and immediately closes the connection by client once it receives the response. But in HTTP streaming, the client fetches the data but leaves the connection open by running a long loop. According to Bozdag and Mesbah and van Deursen's research [7], there are two forms of HTTP streaming, Page and Service Streaming.

Page Streaming This method allows the browser discovering server changes almost immediately. It keeps the connection and runs a long loop. The server script uses event registration or some other technique to detect any state changes. As soon as a state change occurs, it streams the new data and flushes it. After the browser receives the updated data and renews the user-interface, the connection remains open.

There are three drawbacks in this approach. First, it overuses system memory because the JavaScript keeps accumulating, and the browser must retain all of those data in its page model, which will increase the size of the model tremendously with numerous updates. In order to avoid hard drive swapping, the user might have to refresh the page. Second, long-lived connections will inevitably fail, and the developer needs to have a recovery plan. Third, it is going to be a heavy load burden on the server side since the server needs to deal with lots of simultaneous connections.

Service Streaming In order to solve the problems in page streaming, service streaming relies on XMLHttpRequest Call. Instead of the initial page load, an XMLHttpRequest connection is long-lived in the background. This characteristic brings more flexibility regarding the length and frequency of connections. The page will be loaded normally (one time), and streaming can be performed with a predefined lifetime [7]. The responseText property of XMLHttpRequest always contains the content that has been flushed out of the server. As a result, the client side can run a periodic check to see if its contents have been changed.

Long Polling

Long polling, also known as Asynchronous-Polling, is another approach to traditional polling technology emulating push. It is a mixture of purse server push and client pull. The client requests the content from a server with a normal poll. Unlike the traditional way that the server sends back an empty response immediately when there is no data update, the request will be kept open for a set period of time (e.g. 45 seconds). If an update occurs within that period, a response containing the updated data will be sent to the client and the client reconnects. If no update occurs within that period, a timeout event occurs, the response to terminate the open request will be sent to the client from server and the client needs to reconnect asynchronously. One disadvantage of long polling is that if the update occurs very frequent, it does not provide any substantial performance improvements over traditional polling.

WebSocket

We have already seen several approaches (e.g. Long polling, page streaming) solving the problem of getting the real time information updates. However, there are still some issues remaining. Traditional polling and pushing technology result in a few problems:

- The server needs to use at least two different underlying TCP connections for each client: one for receiving messages from client, and one for sending back the response.
- The client has to maintain a mapping from the outgoing connections to the incoming connection to track replies.

A simpler solution would be using a single bidirectional TCP connection for the traffic. The WebSocket protocol provides that capability. Defined in the Communications section of the HTML5 specification, WebSocket represents a new evolution of web-based real-time communication. It is a full-duplex, bidirectional communications channel that operates through a single Transmission Control Protocol (TCP) socket over the web. The WebSocket API is being standardized by the W3C, and the WebSocket protocol is being standardized by the IETF.

Relations with HTTP protocol Currently, many web based bidirectional communication technologies leverage the HTTP protocol as a transport layer to benefit from existing infrastructure (proxies, filtering, authentication and etc.). Since HTTP was not initially designed for bidirectional communication, those technologies were implemented as trade-offs between efficiency and reliability. The WebSocket protocol is designed to supplant those existing HTTP based bidirectional communication technologies, but also keep the benefit from it. It attempts to address the goals of existing bidirectional HTTP technologies in the context of the existing HTTP infrastructure [9]. It uses HTTP ports 80 and 443 as well as to support HTTP proxies and intermediaries. However, as mentioned in RFC 6455 [9], WebSocket is an independent TCP-based protocol and does not have to be bound to HTTP. It does not closely match standard HTTP traffic in order to induce unusual loads on some components. Indeed, its only relationship to HTTP is that the handshake utilizes the existing HTTP header structure and can be interpreted by HTTP servers as an Upgrade request.

WebSocket Handshake The WebSocket protocol contains two parts: a handshake, and then the data transfer. To establish a WebSocket connection, the client sends a WebSocket handshake request, and the server sends a WebSocket handshake response.

A typical RFC 6455 handshake request from the client looks as follows:

```
GET / HTTP/1.1
Upgrade: websocket
Connection: Upgrade
```

```
Host: localhost:5150
Origin: http://localhost
Sec-WebSocket-Key: qfpSNmDK3K8BVtF1hhjmhA==
Sec-WebSocket-Version: 13
```

This opening handshake is a request initialized by Google Chrome 17 to a WebSocket server implementation. It is designed for being compatible with HTTP-based server side. The benefit of this approach is a single port can be used by both HTTP based client and WebSocket based client. And this is seen as the only relationship between HTTP protocol and WebSocket protocol. An HTTP server interprets this as an HTTP upgrade request. According to RFC 6455 [9], a typical client side handshake consists of the following parts:

- An HTTP/1.1 or higher version GET request
- A "Upgrade" header field containing the value "websocket" which is a case insensitive ASCII value
- A "Connection" header field containing the value "upgrade" which is also a case insensitive ASCII value
- A "Host" header field containing the server's authority
- A "Sec-WebSocket-Key" field containing a base64-encoded value which will be used later to generate the "Sec-WebSocket-Accept" field value. This field will let the client know that the response is corresponding response.
- A "Sec-WebSocket-Version" field with a value 13 for RFC 6455
- Optionally, an "Origin" header field containing the origin address. A client sending with this field should be interpreted as a browser based client. On the other hand, a client sending without this field should not be interpreted as a browser based client.
- There are also other optional fields, such as "Sec-WebSocket-Extension" which provide a list of extension the client would like to speak, "Sec-WebSocket-Protocol" field which lists all protocols the client would like to talk, and some other cookie fields, and authentication field. Unknown fields will be ignored by the server.

If the server chooses to accept the connection, the corresponding handshake response from the server looks as follows:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: G6RjYkDEC916Mt9iuSrdlaQgq/s=
```

It contains:

- A status line with a 101 response code, usually looks like "HTTP/1.1 101 Switching Protocols"
- A "Upgrade" field with case insensitive value "websocket"
- A "Connection" field with case insensitive value "upgrade"
- A "Sec-WebSocket-Accept" field that the value is generated based on the value of "Sec-WebSocket-Key" in the request. The algorithm to generate this value will be discussed in the implementation chapter.

Once the handshake is successfully completed, the data transfer phase starts. At this point, both the client and server can start to send (or receive) data. Unlike traditional HTTP connection, WebSocket uses a two-way communication channel where each side can, independently from the other, send data at will.

WebSocket has been widely supported by mainstream browsers [4]. Chrome and Firefox are supporting the latest specification (RFC6455) of the WebSocket protocol. Internet Explorer version 9 does not support WebSocket; however, in the next version 10, it will support the latest RFC6455. There are other browsers supporting an older version of the WebSocket protocol, like Safari, Opera, iOS Safari, Opera Mobile and BlackBerry OS 7 Browser [3]. Nevertheless, those browsers disable the WebSocket by default due to security issues with the older protocol. The Android Browser does not support WebSocket. The Table 2.1 shows current support of WebSocket on different browsers.

Browser	Versions supporting WebSockets
Internet Explorer	10.0
Firefox	8.0, 9.0, 10.0, 11.0, 12.0
Google Chrome	16.0, 17.0, 18.0
Safari	5.0[1],5.1[1], 6.0[1]
Opera	11.6[1], 12.0[1]
iOS Safari	4.2[1], 4.3[1], 5.0[1]
Opera Mini	None
Opera Mobile	11.0[1], 11.1[1], 11.5[1], 12.0[1]
Android Browser	None

Table 2.1: Compatibility Table of WebSocket [4]

[1]Partial support. It refers to the websockets implementation using an older version of the protocol and/or the implementation being disabled by default (due to security issues with the older protocol). Microsoft is currently experimenting with the technology.

2.3 Web Services

Previous sections focus primarily on sharing information on time among different devices. However, in order to ensure devices from different users will not interfere each other and each device from the same user will always get the content they need, a session and user management component needs to be introduced. Since this management also contains remote communication between devices and the management server, their interaction does not need real time state synchronization as much as during the resource sharing stage, in order to remotely manipulate the session and user profile/preference data among different devices running with different platforms, Web Service (WS) is an ideal mechanism for updating/deleting server-side session data model.

WS is a method of communication between two devices over the network. It can be regarded as a web application programming interface that can be accessed on line, and executed on a remote system hosting the requested services. Commonly, the communications between clients and servers in Web service are transported over HTTP allowing for easier communication through proxies and firewalls than old remote communication technologies, such as Remote Method Invocation (RMI).

The use of WS can also provide other advantages. Unlike desktop computers, mobile devices usually suffer the restriction of battery life and limitations of hardware performance. Since today most mobile platforms support network connection, it is an ideal solution to take parts of a mobile platform-side function to the server and publish it as WS APIs to allow mobile devices to connect.

2.3.1 Overview of Web Service

From W3C Web Service Architecture (WSA) [37] specification, the purpose of WSA is to provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. There is no specific rule to define how Web Services should be implemented and combined. Any software system designed to support interoperable machine-to-machine interaction over a network is a Web Service. In a Web Service, it usually has an interface described in a machine-understandable format (e.g. WDSL). Machines from network can interact with the Web Service in a manner prescribed by its description.

2.3.2 Models of Web Service

A model is used to explain and encapsulate a significant theme within the WSA. According to W3C WSA [37], there are four models for WSA. They are:

- **Message Oriented Model (MOM)** focuses on aspects related to messages exchanging among agents. Those key aspects include the structure of messages, the relationship between sender and receiver agents and the mechanisms to deliver messages.

- **Service Oriented Model (SOM)** focuses on aspects of service and action. The term "service" used here is the notion of one agent performing actions for other agents. Clearly, services cannot be adequately realized without some means of messaging. SOM builds on MOM, but unlike MOM focuses on messages, its primary purpose is to explicate the relationships between an agent and the services it provides and requests.

- **Resource Oriented Model (ROM)** focuses on resources representation. This model comes from the idea of Web Architecture. It concentrates on issues such as the policies applied to the resource, the ownership of the resources, the category and hierarchy of the resources and so on. The popular RESTful Web Service is an example of ROM.

- **Policy Model (PM)** focuses on constraints on the behavior of agents and services to provide secure service with good quality. Policies are about resources and applied to agents that may attempt to access resources.

2.3.3 Types of Web Service

Web service communications are platform independent and the development language can be various. There are two merging camps of implementing Web services: REST-compliant Web services and arbitrary Web services.

REST-compliant Web Service

A REST-compliant Web Service is a type of Web Service whose primary purpose is to manipulate XML representations of Web resources using a uniform set of stateless operations. It first appeared in Roy Thomas Fielding's Doctoral Thesis "Architectural Styles and the Design of Network-based Software Architectures"[10]. Compared with the other types of Web Services, it is a relatively newer type of Web Services and quickly become popular. REST stands for "Representational State Transfer"and is a Resource Oriented Model Web Service, which focuses on resource representation. It defines a uniform set of architectural principles. A complete implementation of a RESTful Web Service application should follow four basic design rules:

- **Resource identified through URI.** Because RESTful WS is Resource Oriented Model (ROM) WS, resource is the fundamental concept. In RESTful WS, resource is identified through Uniform Resource Identifier(URI) and organized by certain policies. The service holder provides a set of resources that are regarded as targets of interactions with clients. The individual resources should be exposed as directory structure like URIs. This type of structure makes it easy for developers to use and understand the resource and the relationship among resources.

- **Use standard methods explicitly provide uniform interfaces.** In order to simplify and decouple the architecture, the service provider should define uniform interface between clients and servers. In

an HTTP based RESTful Web Service, the standard HTTP methods are explicitly used as uniform interfaces. The service must at least support four major HTTP methods (GET, POST, PUT and DELETE) and maps those method to Create, Read, Update, and Delete (CRUD) operations. The one-to-one mapping is usually defined as in Table 3.1.

HTTP Methods	RESTful Web Service Interface
POST	To create a new resource on the server
GET	To obtain a resource from the server
PUT	To change the state of a resource or to update it
DELETE	To delete a resource

Table 2.2: RESTful Web Service HTTP Method Mapping Example

- **Statelessness.** In stateful client-server communication, the server needs to store the client context, which means the more clients connect to the server, the greater the burden the server needs to carry. In RESTful Web Service, the client-server communication is stateless. Requests sent from the client sides contain all the necessary information to fulfill the request. The server-side does not need additional information from other messages that were sent previously. This feature makes the server more reliable in the face of partial network failures as well as further enhancing their scalability.

- **Cache.** In RESTful architecture, a cache is used for improving network efficiency. The data within a response should be implicitly or explicitly labeled as cacheable or non-cacheable. If the response data is cacheable, then the client side is able to reuse that data for later request. This is called a Client-Cache-Stateless-Server (C$SS) style [10]. The advantage of adding cache components is that they have the potential to partially or completely eliminate some interactions, improving efficiency and user-perceived performance [10].

- **Layered system.** A layered system is organized hierarchically with each layer providing services to the layer above it and using services of the layer below it. [15] Each layer is only allowed to "see"the immediate layer which they are interacting. This feature may improve system scalability by enabling load balancing, as well as providing shared caches at each layer. It also can enforce the security policies among layers.

- **Message format.** The format of the data entity-body that the server and client-side application exchange could be XML, JSON, both, or XHTML. This message format makes the message to be simple, easy to use, and human-readable. In addition, client-side application can be written in different languages running on different platform.

What follows is an example of querying Saskatoon weather by using RESTful Web Services. At the beginning, the request sends a simple HTTP request to get the current weather of Saskatoon:

```
GET /currentweather/saskatoon HTTP/1.1
Host: myrestservice.com
Accept: text/xml
Accept-Charset: utf-8
```

Since the resource is represented by URI and the interface of RESTful WS utilizing existing HTTP methods, it does not need extra message to carry information. The only thing the client side needs to do is to send a standard GET request to the URI http://www.myrestwebservice.com/weather/Saskatoon.

```
<?xml version="1.0" encoding-"UTF-8"?>
<s:Quote xmlns:s="http://myrestservice.com/weather-service">
    <s:CityName>saskatoon</s:CityName>
    <s:Temperature>12</s:Temperature>
    <s:Measurement>c</s:Measurement>
    <s:Humidity>86</s:Humidity>
</s:Quote>
```

The upper message could be a subset of a response message sent back from the server. We can see that it is human readable and only needs a few XML mark-ups. It is standard XML format message transported by well-known W3C HTTP protocol. The infrastructure of RESTful Web Services is already pervasive.

Arbitrary Web Service

Unlike RESTful Web service using a uniform set of operations, arbitrary Web service exposes an arbitrary set of operations. Before SOAP 1.2, the term arbitrary Web services primarily means SOAP based Web services [39] [28]. Since SOAP 1.2, the SOAP Web Service can be used in a manner consistent with RESTful Web services. We call it Arbitrary Web Service here.

Before SOAP 1.2, SOAP stands for "Simple Object Access Protocol". It is a lightweight protocol specification that defines a message architecture and message format based on XML format. A service is provided by exchanging those structured information through a network-accessible endpoint. Those exchanges usually rely on a variety of underlying network application layer protocols, such as Remote Procedure Call (RPC), Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP) or other appropriate messaging protocols. The SOAP document defines a top level XML element called envelop, which in turn consists of a required Body element and an optional Header element. The structure of a SOAP message is shown in Figure 2.8:

The Body element carries the payload of the message. The optional Header element can contain additional information (such as addressing information for routing the message) not directly related to the message

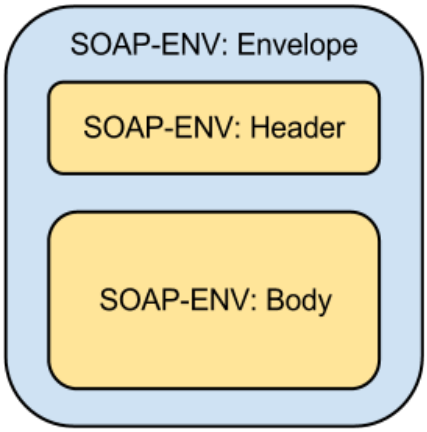


Figure 2.8: Structure of SOAP Message

content. Since the message is described in XML format, by using the XML parser, two endpoints (the client and the server) of the SOAP service can marshall and unmarshall the message content and route it to each other.

The design goals of SOAP are simplicity and extensibility. To meet these goals, SOAP provides a simple messaging framework whose core functionality is to provide extensibility. From the messaging framework, SOAP attempts to omit features that are often found in distributed systems such as "reliability", "security", "correlation", "routing", and "Message Exchange Patterns" (MEPs) [38].

In order to process SOAP messages, SOAP uses a distributed processing model and assumes the SOAP message is sent from an initial sender to an ultimate receiver via none or numbers of intermediaries. As mentioned above, SOAP attempts to utilize many MEPs which are often found in distributed systems. Consequently, the message can be exchanged via different patterns such as one-way connection, request/response interactions, peer-to-peer conversations and so on. SOAP Web Service is stateless, which means when the receiver processes a SOAP message, it only needs to consider this single message. The processing model does not need to maintain any state or perform any correlation or coordination between messages.

The following shows a SOAP request message and its corresponding responds which query the weather in Saskatoon. The request message example is:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/1999/XMLSchema">
    <SOAP-ENV:Body>
```

```
        <nsl:getWeatherByCity
         xmlns:nsl="urn:MySoapServices">
            <paraml xsi:type="xsd:string">saskatoon</paraml>
        </nsl:getWeatherByCity>
    <SOAP-ENV:Body>
</SOAP-ENV:Body>
```

When the WS server receives the request message, it gets the result and returns a message as following:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/1999/XMLSchema">
    <SOAP-ENV:Body>
        <nsl:getWeatherByCityResponse
         xmlns:nsl="urn:MySoapServices"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <return xsi:type="xsd:int>12</return>
        </nsl:getWeatherByCityResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Comparison between RESTful Web Service and Arbitrary Web Service (SOAP)

From the above information, we would find that SOAP based Web Services is more concentrated on various interfaces offered by the server. On the other hand, RESTful Web Service relies on various resources. As a result, SOAP is usually considered as Service Oriented Web Service since it provides different services (verb) for user to consume, however, REST is considered as Resource Oriented Web Service (noun) because it primarily focuses on different resources with uniform interfaces. In the SOAP based Web Service environment, the consumer has to contact some sorts of business registry service, such as Universal Description Discovery and Integration (UDDI), to find possible Web Service and then use WSDL to pass request information to the server. In the RESTful Web Service environment, the service consumer would need to know the location of the resource (e.g. URIs) and then use standard methods (e.g. HTTP methods: GET, PUT, POST, DELETE, HEAD) to manipulate or query the data.

2.3.4 Uses of Web service

According to WSA specification [37], Web services are most appropriate for the following applications:

- Distributed systems with remote operations over the network where reliability and speed cannot be guaranteed;
- Where there is no ability to manage deployment so that all requesters and providers are upgraded at once;
- Where components of the distributed system run on different platforms and vendor products;
- Where an existing application needs to be exposed for use over a network, and can be wrapped as a Web Service.

The service steps are:

1. The requester and provider entities become aware of the other party by using a discovery service. This step can be initiated by either the requester agent or provider.
2. The requester and provider entities make the agreement between the semantics and service description. This step does not mean both entities need to communicate with each other. It simply means that both sides must have the same or compatible understandings of the semantics and service description.
3. The service description and semantics need to embody in both the requester agent and the provider agent as appropriate.
4. The requester and provider entities start to exchange SOAP messages on behalf of their owners.

2.4 Summary of Literature Review

This chapter reviews a few possible techniques for solving problems described in the previous chapter. Table 3.2 categorizes and lists all the reviewed research based the area they can be applied on.

In summary, the current research gives possible solutions for these questions mentioned in the last chapter:

- **How to distribute the software architecture, the resource data, the application logic and functions among different devices?**

  As discussed in the first section of this chapter, distributed MVC is a suitable software architecture pattern for this requirement because it separates different logic elements of the application, and each element can be implemented and maintained individually to provide loosely coupling.

- **How to make the application open for adding new features in the future?**

  The distributed MVC also provides a suitable solution to increase the code re-usability and extensibility, which is a problem mentioned in the problem definition chapter. The decoupling among data model, information representation and application controller makes it relatively easy to replace or extend one part of the application without significantly change other components.

List of Research Solutions	
Distributed MVC Architecture	- Classical MVC architecture and its derivatives [11] [14] [18] [25] [31] [34] [36] - Challenges of Distributed System and possible solutions[20] - Messaging System for delivering information among components[21] - Researches on Messaging based Distributed MVC Architecture[21] [24] [33] - Publish-Subscribe Architecture for providing one-to-many delivery[14] [20] [24]
State Synchronization	- Different categories of data delivery mechanisms[13] - W3C Compliant Information Dissemination Approaches [7] [9] [16] [29] - WebSocket for full-duplex and bidirectional communications [3] [4] [9]
Web Services	- Definition of Web Service [37] - Meta-models of Web Service (MOM, SOM, ROM, POM) [37] - REST-compliant Web Service to manipulate XML representations of Web resources using a uniform set of stateless operations [10] [15]

Table 2.3: List of research solutions

- **How to distribute classic MVC framework among multiple devices?**

  Working with different devices across network in a distributed system, certain changes are needed to reform the classic MVC pattern. The primary change is that in classic MVC, component communicates with each other via method calls, however, in a distributed system, it could easily overload the call receiver if many callers call it at the same time. Messaging systems usually do not have this limitation because they queues messages into a channel and read and process them one by one on the receiver side. Also, since the messaging system provides storage and forward mechanisms, the message delivery can be considered as more reliable than RPC [20]. As a result, the messaging system is a better solution in this situation compared with other approaches.

- **How to communicate between those distributed components and synchronize the application state and resources among them at real-time?**

  In the second section of this chapter, a few information dissemination technologies were discussed. Historically, for a bidirectional communication between a client and a server, it is usually the client side application's duty to initialize the connection and query about the updates. This approach is called a client pull. However, there is a trade-off between efficiency and real time information dissemination. If the client queries the information frequently, there are too many query messages sent from the client to the server thereby wasting the bandwidth and other resources. If the client queries the information less frequently, the information update cannot be delivered to the client side on time. On the other hand, the server push approach makes a real time update and does not waste the bandwidth on querying

messages. Several contemporary server push approaches are introduced in this section and we mainly focus on the WebSocket push technology because compared with other approaches such as long polling, this is a real server push technology that once the WebSocket connection is initialized, the server and client can send information freely to each other without any request.

- **How to distribute resources on different devices and represent the resource decently on screens of those devices with different resolution, as well as manage and distribute those views into different devices that have different screen size and resolution?**

  Resources can be passed by dynamically generated XML, JSON or XHTML format messages in WS. RESTful WS allows the system to trim the message according to the device's hardware configuration and user preferences. Therefore, the P/S middleware only passes the required information to devices and those devices will present the resource in their local view component according to those messages. Different devices will have different display based on their local view component, device configuration and user preferences.

Overall, this chapter reviews several contemporary technologies that we need to solve problems in Chapter 2. However, there are still some **open questions** remaining, namely:

- How to design a complete distributed MVC architecture for multiple devices sharing and synchronizing resources over the network?
- How to enable the application to support as many devices as possible with minimum effort?
- How to achieve personalized service mashup for mobile clients?
- How to manage devices in different groups to ensure them won't interrupt each other?
- How to implement the 1-to-many Publish-Subscribe pattern with the WebSocket push technology?

In the next chapter, I will start to introduce how to utilize technologies that we discussed in this chapter to design our architecture and solve the problems listed above.

Chapter 3

Design and Architecture

In last chapter, a few contemporary technologies, such as MVC architecture pattern, state synchronizations, Web Services, are reviewed. Among these technologies, the Distributed MVC architecture design pattern is discussed as a feasible solution for building an elegant, simple and stable distributed architecture; the server push technology provides an effective communication mechanism among multiple devices; the RESTful Web Service is a simple Web Service design model for representing different types of resources in distributed systems. In this chapter, these technologies are used to propose a possible solution for the problems described in the problem definition chapter.

3.1 Overview

As mentioned in Chapter 2, the goal of the n-Screen Application Framework (NSAF) is to provide a seamless user experience when the user tries to share and consume resources on one or many devices. When the number of connected devices is larger than one, the graphical representations of the sharing resource may be displayed separately on multiple screens. In the following section, a real world example is introduced to give a better illustration of the NSAF's work flow.

3.1.1 A Real World Example

This example shows a user uses NSAF to share and consume a set of multimedia resources on three different types of devices: a Smart TV, a tablet and a smartphone. In this example, we use four different scenarios to explain the run-time mechanism:

Scenario 1 As shown in Figure 3.1, the user first logs into the system on the Smart TV (e.g. Google TV). In this step, the TV sends the login information to the server and the server checks the authentication. If the authentication is passed, the server reads the user resource configuration file, which stores a list of resources that the user has been subscribed. Figure 3.2 shows an example of the configuration file written in XML format.

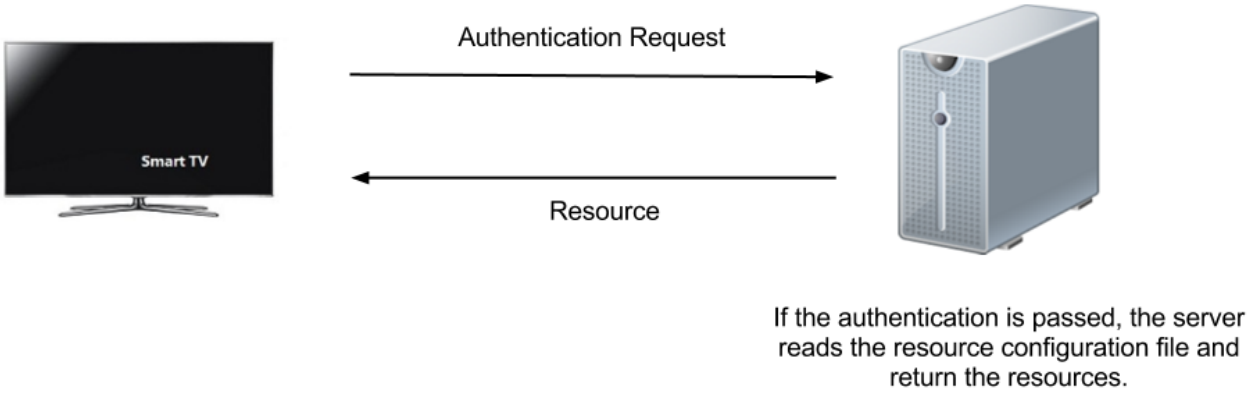


Figure 3.1: Scenario 1: the user logs into the system on a Smart TV

```
<?xml version="1.0"?>
- <components>
  - <component>
      <id>c1</id>
      <type>video</type>
      <mandatory>1</mandatory>
      <link>rtsp://www.example.com/video.mp4</link>
    </component>
  - <component>
      <id>c2</id>
      <type>text</type>
      <mandatory>1</mandatory>
      <text>This is the description about the video.</text>
    </component>
  - <component>
      <id>c3</id>
      <type>list</type>
      <mandatory>0</mandatory>
    - <list>
      - <item>
          <id>item1</id>
          <link>http://www.example.com/item1</link>
        </item>
      - <item>
          <id>item2</id>
          <link>http://www.example.com/item2</link>
        </item>
      </list>
    </component>
  - <component>
      <id>c4</id>
      <type>mp3</type>
      <mandatory>1</mandatory>
      <link>http://www.example.com/music.mp3</link>
    </component>
</components>
```

Figure 3.2: An Example of Configuration File

In the above configuration file, each resource is represented as a component. The component c1 is a video resource contains a link; the component c2 is a text that describes the video; the component c3 is a list that each item in the list contains a web link; the component c4 is an mp3 resource. Each component element must contain an id element, a type element and a mandatory element. The id element is used to identify the component. The type element indicates the type of the resource component. For the mandatory component, if the value is 1, it means the component must be displayed at least on one device from the current connecting group; if the value is 0, it means the component can be removed from the display due to optimization. Based on different types of the component element, other elements may be included to ensure the client side application can display the component correctly.

Since the Smart TV is the only device that currently logs in under this user's name, all these four components are displayed on the Smart TV. The server pushes the components information to the Smart TV and the TV displays them according to the client-side application.

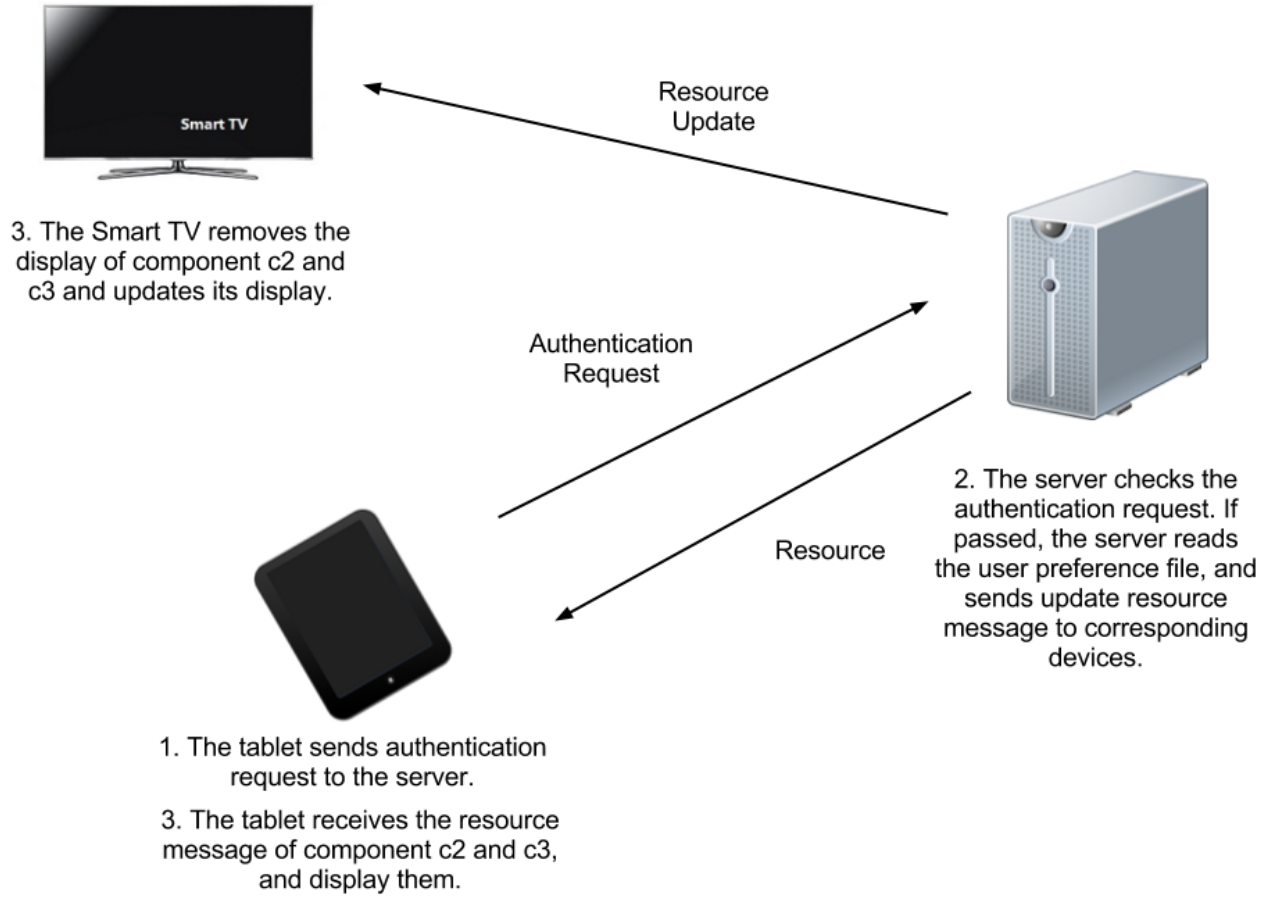


Figure 3.3: Scenario 2: the user continues to log into the system on a tablet

Scenario 2 In Figure 3.3, after using the Smart TV for a while, the user decides to sit on the sofa and to use a tablet to view the description of the video. He launches the NSAF client-side application installed on

the tablet and enters his password to log in. The client-side application passes his authentication information to the server and the server checks his authentication. Once the authentication passed, the server checks if the request device is the only device that currently logs in under this user's name. Since the user has already logged with the Smart TV, the server realizes that there are more than one devices are currently logging in the system under the user's name. The server then reads a user preference file to manage resources among multiple screens. Figure 3.4 is an example of this user's preference file.

```
<?xml version="1.0"?>
- <preferences>
  - <preference>
      <id>p1</id>
      <name>Default</name>
    - <components>
      - <component>
          <id>c1</id>
          <device>tv</device>
        </component>
      - <component>
          <id>c2</id>
          <device>tablet</device>
        </component>
      - <component>
          <id>c3</id>
          <device>tablet</device>
        </component>
      - <component>
          <id>c4</id>
          <device>tv</device>
        </component>
      </components>
    </preference>
  - <preference>
      <id>p2</id>
      <name>Customize 1</name>
      <component> ... ... </component>
    </preference>
  - <preference>
      <id>p3</id>
      <name>Customize 2</name>
      <component> ... ... </component>
    </preference>
</preferences>
```

Figure 3.4: An Example of Preference File

The user preference file saves the user preferences of the display arrangement of the resource components among devices. In this example, before the user logs into the system on the tablet, all components are displayed on the Smart TV. After he logs in with his tablet, the server reads his preference file and checks

the name of preference he is using. Assuming he is using the default preference, the system finds that the preferred display device of component c2 and c3 is the tablet, and the tablet is currently connected to the server. Then the server pushes an update message to the Smart TV to request the TV to delete the representation of these two components and pushes the resource information of these two components to the tablet. The client-side application on the tablet receives the resource information and displays them.

Scenario 3 The user decides to move to another room and his smartphone is the only device he would like to carry. He wants to listen the mp3 resource on this smartphone, but no other changes are needed for the rest components. As the first two scenario, the user logs into the system on his smartphone. Since there is no component that has the smartphone as its preferred device, no display changes are made after the login. Then the user selects the mp3 resource which is currently displayed on the Smart TV, and set it to be displayed on the smartphone. Then the client-side application on the Smart TV sends an update request message to the server. The server receives the message and sends the updated resource information to both the Smart TV and smartphone. Now, the mp3 resources is moved from the Smart TV to the smartphone. This updating process is shown in Figure 3.5. When the user moves to another room, even though he cannot watch other resources, he can still listen to the mp3 resource on his smartphone. After this change, if the user chooses to save the current preference, when he logs in next time, all resources are going to be displayed as current configuration.

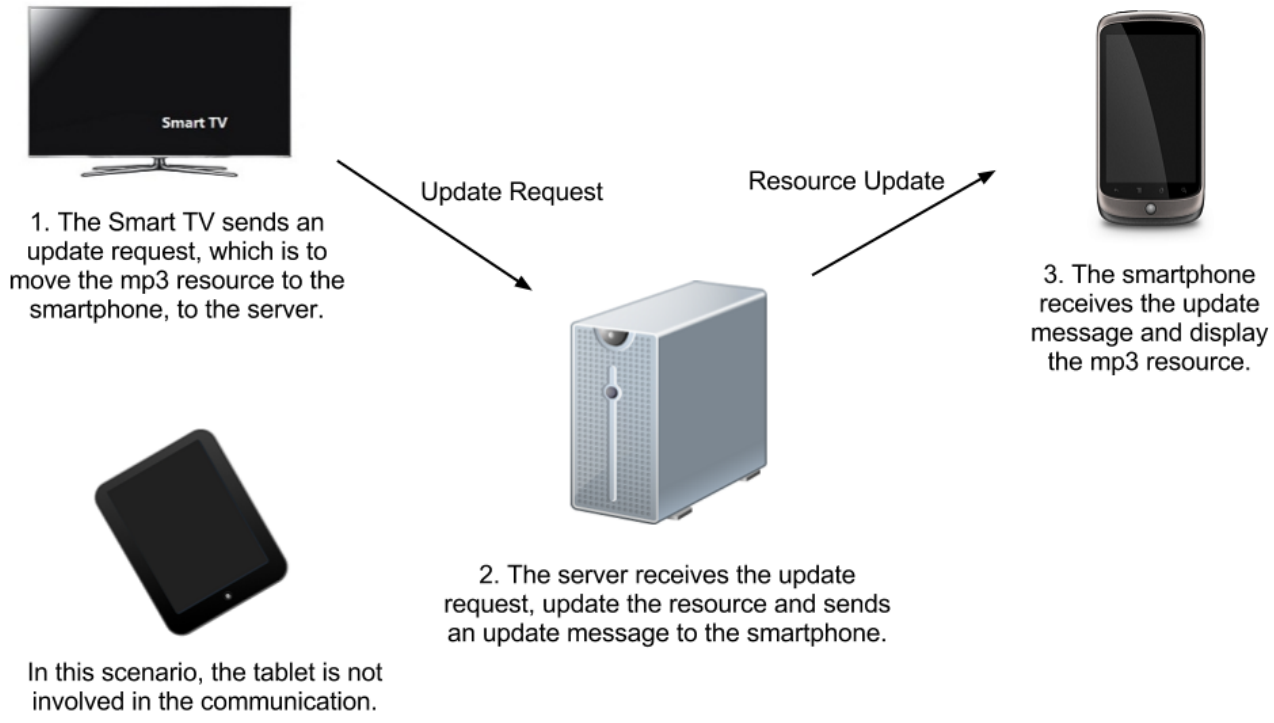


Figure 3.5: Scenario 3: the user moves a component from the Smart TV to his smartphone

Scenario 4 The user walks back to the room and he does not need to use the smartphone to display the

mp3 anymore. The user can use the log out function in the smartphone-side application. The application sends an update request to the server and closes the connection. The server receives the request, moves the resources displayed on the smartphone to one of the connected devices and sends an update message to that device. This scenario is shown in Figure 3.6.

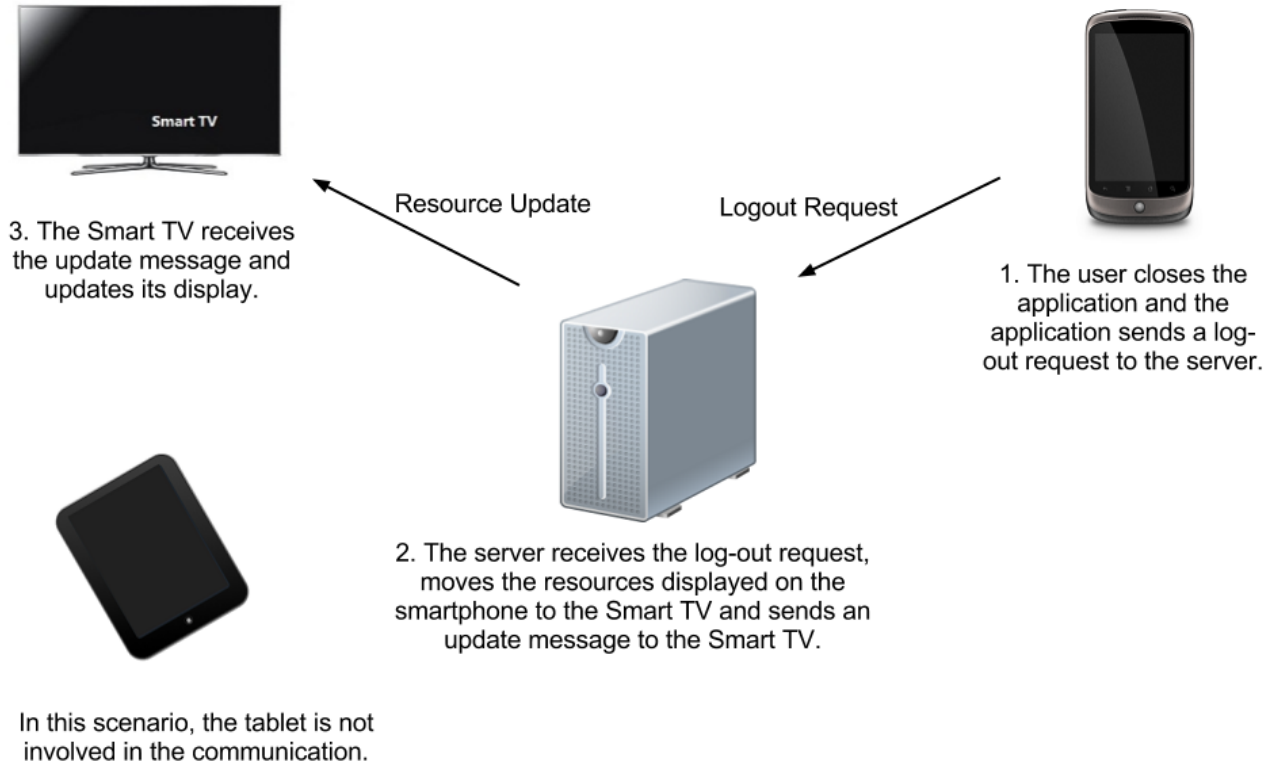


Figure 3.6: Scenario 4: the user closes the application on the smartphone

3.1.2 Overview of NSAF Architecture

The NSAF contains two parts, the front-end application over different platforms and the back-end application (server service) running with RESTful Web Service and the database.

The NSAF front-end application handles the user interaction and provides the visualization. Since the font-end application might be running on different types of devices, it could be written in different ways. It can be a mobile application running in the Android platform, a desktop application written by Java or C#, or even a web application running on different browsers. They provide visualization of the data model to the end user. According to the users' preference, even same model might have different visualization.

The NSAF back-end application contains two parts, a RESTful WS implementation which provides the service API for front-end applications to create session, query and update resources, as well as an information dissemination part to propagate information to corresponding subscribers.

The Figure 3.7 shows an overview of RESTful WS side service interaction. The client first needs to

request the resource through published service API. According to the HTTP method the client is using, the server process various actions, such as creating new session, generating new resource, deleting existing resource, querying existing session information and so on. By interpreting the HTTP message the client sends, the server generates different types of response. An Android Java client side application gets an XML or JSON representation of the resource. A browser-based application gets a HTML page as response. In those responses, a channel information is attached. Once the client registers to that channel, any information published to that channel will also be pushed to the registered clients.

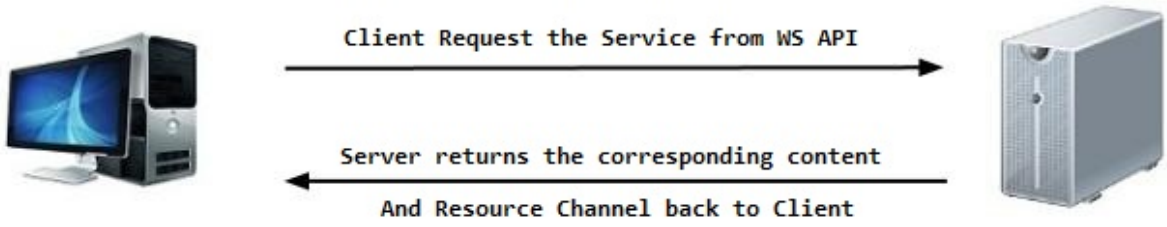


Figure 3.7: Service Request Overview

In the information dissemination phase, the actions between front-end application and the back-end application are shown in Figure 3.8 and Figure 3.9. In Figure 3.8, a user action might cause the client application sending an update request to the server via the channel. As shown in Figure 3.9, the server updates the resource according to the request and then pushes the new state to other devices that are sharing the same session at the same time. When other client side devices receive the state synchronization message, they automatically update their representation without user request, which provides the seamless user experience.

3.2 Design Requirements

In order to achieve the goal of NSAF, the system needs to meet certain requirements. The requirements are derived from the literature review and the goal of the NSAF.

Modularity In order to increase the reusability, flexibility, and maintainability of the code, the NSAF must separate the presentation from data model and the controller from the presentation. As discussed in the distributed MVC section in the literature review, the NSAF will be implemented in a distributed MVC pattern.

Protocol. To reduce the bandwidth cost of the NSAF, the system must support not only client pulling, but also server push mechanism. Based on the literature review, HTTP-based web application benefits from an integration of a push protocol in both time and bandwidth consuming.

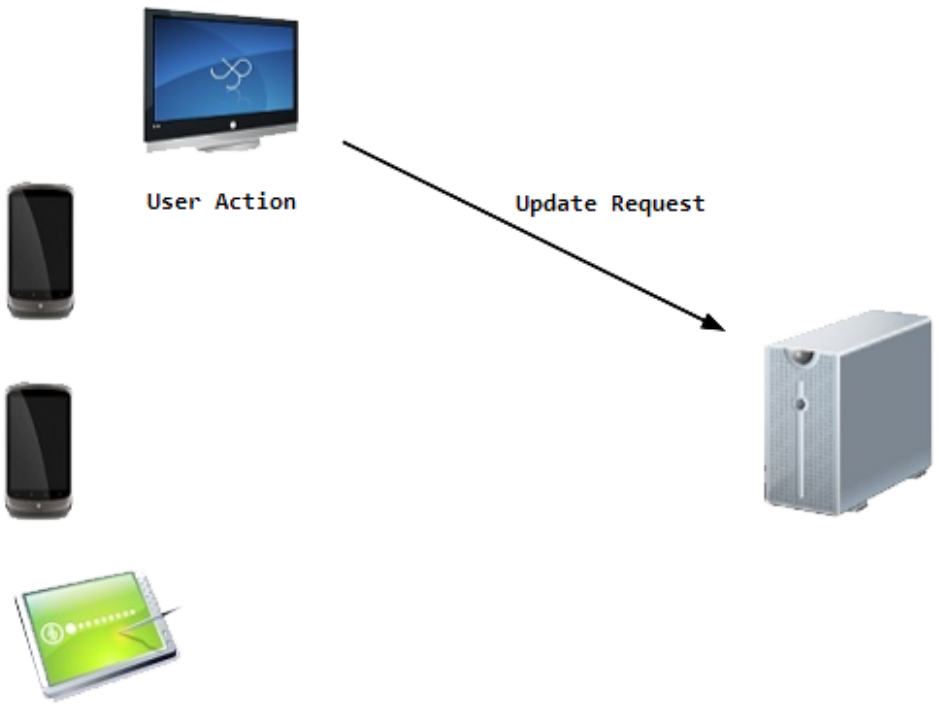


Figure 3.8: Update Request

Consistency. Once a data update event occurs on the server side, this change of information must be disseminated to all the corresponding subscriber client in time.

Delivery Scheme. NSAF system must support 1-to-N delivery scheme which is better organized than a broadcasting mechanism, but not a unicast dialog.

Network Performance. The NSAF must outperform traditional polling mechanism in terms of latency and amount of a transferred data.

Thin Client. Since the performance capability of front-end device varies, the NSAF front-end application must only handle the user interaction, receive notification and provide the visualization. No complex application logic should be implemented in the client side.

Asynchronous Communication In order to provide seamless user experience, the information transmission between server and clients is asynchronous. The sender (may be a client when a client side application sends a update request or the server when the server disseminate the information to subscriber clients) does not have to wait for the receiver to receive and process the information. The end user should not be aware of and wait for network data transmission.

Well Organized Information. The server keeps the whole information as a tree of XML document. However, only necessary data are chopped off and used to generate the message which delivered to corre-

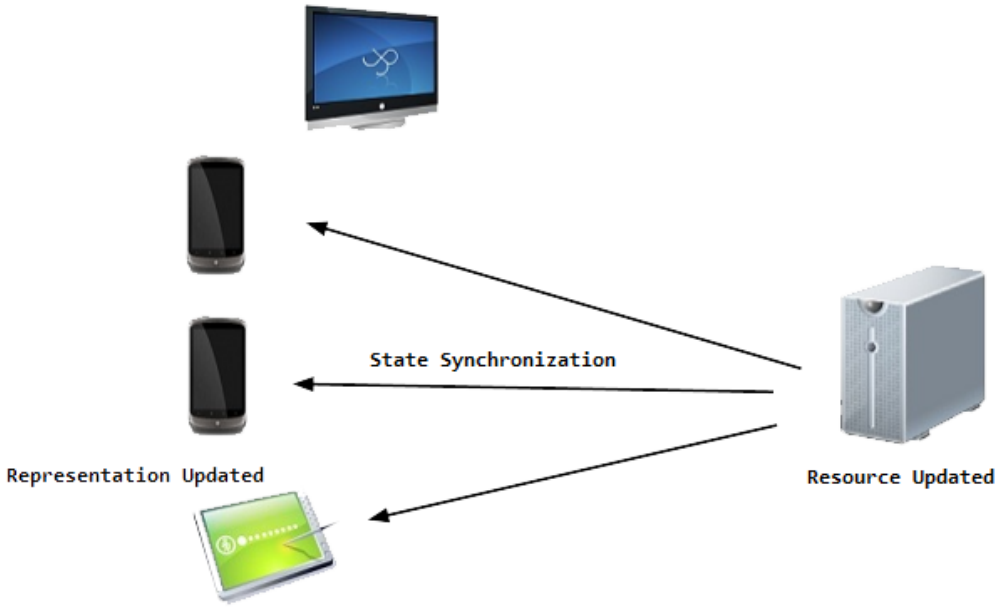


Figure 3.9: State Synchronization

sponding subscriber client. This approach reduces the bandwidth cost.

Portability. Software portability requires the usability of the same software in various platforms. The NSAF front-end application is implemented in different languages according to the working system, such as Android smartphones, desktop computer, web browsers, and etc. This portability enables NSAF to be deployed on different platforms. The application development should not require any complex library to ease the adoption of the system.

Web Service. Based on the literature review, parts of the system should be implemented by RESTful Web Services that provide a standard means of interoperating between client-side applications running on different platforms.

The Service Contract. There should be an agreement that specifies how clients and services may interact when the client is trying to subscribe the service. The WS APIs should be designed with an emphasis on when the information should be exchanged between client and server, and what information should be carried during the exchange.

Encapsulation. The service API should generally hide implementation details. The client side application should not know the detail implementation of the server side. Encapsulation prevents the client tightly coupled to the server.

Network Failures Tolerance. Since mobile devices connect to the internet via a wireless network (e.g. 802.11g) and the device holder often moves physically while he or she is using the device, the bandwidth could be fluctuated severely, and the connection could be intermittently lost. Consequently, the client side NSAF needs to work with an inconsistent transmission rate and a sudden loss of connection.

3.3 Platform Independent

To evaluate the use of NSAF among different devices, the NSAF client-side application is implemented into two styles: web browser-based application which requires support of WebSocket and standalone application which runs on Android phones (version 2.3), Android tablets (version 3.1), and personal computers which installs Java Virtual Machine (JVM). For Java based application, in order to provide simplicity and portability, part of the functions are implemented as Java library and use common Java libraries that can be supported both on mobile and stationary computers. The browser-based application is implemented by popular technologies, such as HTML5, CSS3, Javascript and jQuery which is runnable among most devices. The server-side application is implemented by Java to provide portability.

3.4 NSAF Architecture

The NSAF contains two major parts: a RESTful WS for session management to manage user profile and preference and coordinate devices through sessions, and data dissemination component handling push content to session channel subscribers after a session channel established and subscribed by different devices. The system can be viewed from different aspects. I describe the architecture in the following sections from two aspects: a layered view which describe each component and their interactions in detail and a distributed MVC view which gives an overview of the NSAF system.

3.4.1 Layer View of NSAF Architecture

The NSAF system contains a set of components. And as shown in Figure 3.10, those components can be logically divided into the following layers:

The Transportation Layer As in the Open Systems Interconnection (OSI) model, it contains TCP/IP and UDP protocols for message transmissions. Its primary responsibility is to create connections between different devices. This layer is used by the upper layer to build protocols with specific objective.

The Protocol Layer is the transportation protocol layer. It contains network protocols that have been used by the NSAF systems, such as WebSocket which utilizes the interface from transportation layer to provide a Push feature.

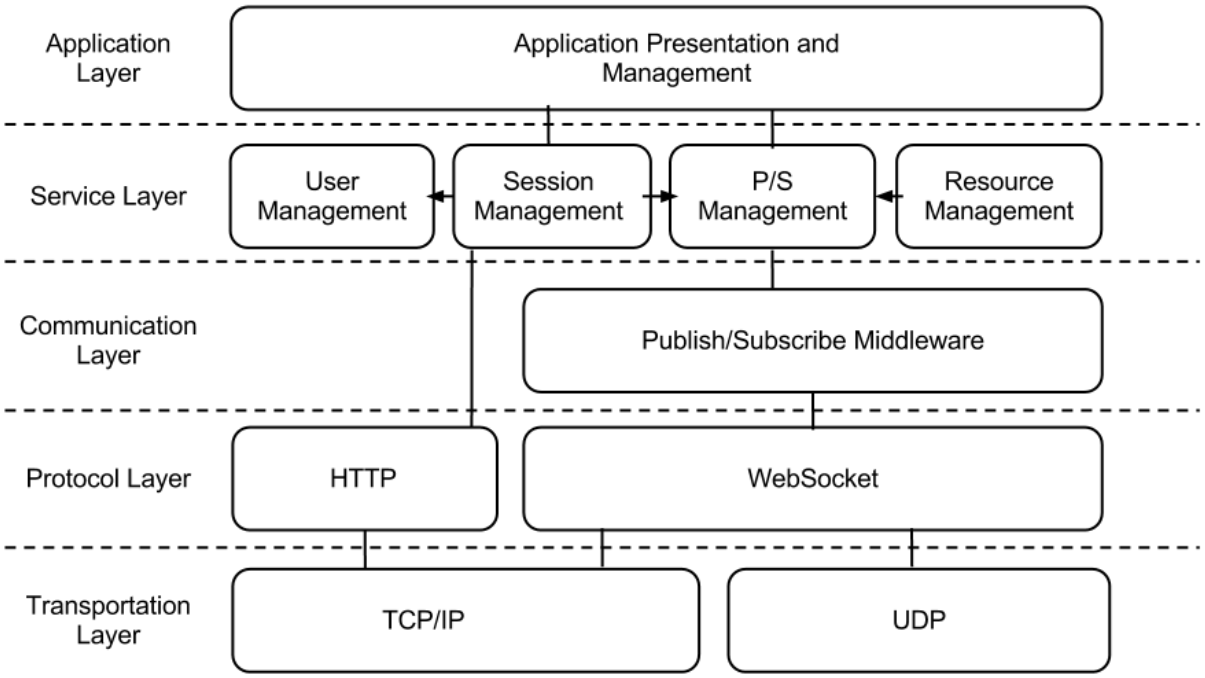


Figure 3.10: NSAF Architecture in Layers View

The Communication Layer contains the Publisher/Subscriber messaging middleware. It enables the interaction between the message publishers and subscribers. A Worker Agent in this middleware pushes messages in each channel to the corresponding subscribers.

The Service Layer contains the services that are needed by NSAF. This layer includes resource, session, user profile and preference management, storing and managing the device-dependent content. The P/S Management component is a mediator between the P/S middleware and Application Layer. Its duties include: managing channels in the P/S message middleware, finding a corresponding message channel for a specific message from upper Application Layer (or Resource Management) and pushing it into that channel, receiving messages from specific channels and forwarding it to Application Layer or Resource Management. This P/S Management component is also distributed. The Resource Management component implements the resource logic and stores the resource. Its duties include interpreting a message from the P/S Management component and doing corresponding updates in the resource. The Session Management component provides its network interface in a RESTful WS way. And it fetches the user preference from the User Management component, as well as creates channels or queries channels information by communicating with P/S Management components. Once it gets the channel information and the user preferences, the Session Management component combines them together, chops off unnecessary resource information according to the user preferences and generates platform specific return

messages to the clients. The return message is in various formats (HTML, XML and JSON) according to the client side application. The User Management components stores and manages user profiles and preferences.

The Application Layer presents the visualization to the end user which can be considered as the View component in the MVC model. It also catches user actions and connects relative APIs from Service Layer.

The Figure 3.10 also shows the interaction between these components in two scenarios: session initialization scenario (user first connects to the WS component to start a session channel), as well as publish (a publisher releases content to a session channel) and subscribe (a subscriber subscribes a session channel) scenario:

Session Initialization Scenario If a device needs to create a new session via RESTful WS API, the Session Management component creates a new session for that user. By interacting with the Publish/Subscribe (P/S) Component, a new session channel is created. If the user is already using a session channel and this device is a new device from the same user that is trying to connect to this session to share the same resources, the Session Management component tries to communicate with the P/S Component to get the existing channel information. The Session Management component also finds the user profile and preferences from the User Management component. A resource information message (with channel information) is then generated and returned back to the client side application. According to the client-side application, the resource information can be in different types. If the client side application is implemented in Java, the returned message is in either JSON or XML format. If it is a browser-based application, the returned message is an HTML file. Since this part primarily uses RESTful WS to manage sessions, the Session Management component directly connects to the HTTP protocol in the Protocol Layer as shown in Figure 3.10.

Publish/Subscribe Scenario Once the channel information is returned to the user, that user is also registered as a subscriber to that channel on the server side. By leveraging WebSocket protocol in the Protocol Layer, a bidirectional connection is established between the channel and the end user. When a message needs to publish (usually from the Resource Management component) to a specific group of clients, the P/S Management Component first pushes it to the corresponding channel, and then a Worker Agent in the P/S middleware pushes the message in that channel to each client.

3.4.2 Distributed MVC View of NSAF Architecture

Connector between Components

The NSAF utilizes the design model described in the distributed MVC section of the literature review chapter. The model, view and controller components are spread on different machines. As mentioned in

the literature review chapter, for distributed system, the connector plays an important role. The classical MVC utilizes method call as the connector handling communication between MVC components. This type of connector is not suitable for a distributed architecture, since it is not directly available for interacting with the distributed components [32] and may be overloaded by a large number of calls over a short time period. Consequently, in order to provide a stable and efficient communication between components for our NSAF system, a new type of connector must be designed. Some of those requirements have been already discussed in previous chapter, which include: loosely coupling among distributed components, transport of information, description of dependency and format of information. As shown in Figure 3.11, the connector between distributed components over different physical devices in NSAF system is described as a four-layer connector stack: TCP/IP protocol (or UDP protocol), WebSocket protocol, Publish/Subscribe middleware, and message format protocol. The TCP/IP protocol (or UDP protocol) provides the socket connection between different devices. The WebSocket protocol allows bidirectional communication between subscriber and channels in the P/S middleware. The Publish/Subscribe middleware delivers different messages to corresponding channels and coordinates different components in the Service Layers and the message format protocol sets up a message format agreement for the whole system.

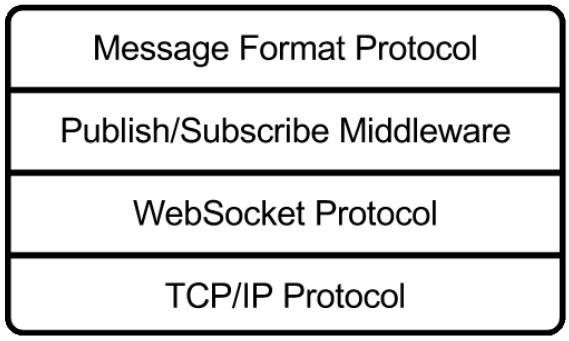


Figure 3.11: Connector Stack

Distributed MVC Components

From the MVC perspective to view NSAF architecture, unlike the traditional MVC that all components reside on a single machine, in NSAF, the model is represented as the Resource Management Component in the Service Layer on a server and the view is the visualization component in the Application Layer on different client side applications. The controller is not just a single component on one machine, but a set of components with different functionalities distributed on different physical machines. Its function is not only monitoring user activity from outside, but also acting as a bridge between Model and View and sending/receiving messages from inside.

Figure 3.12 describes the MVC architecture of NSAF. As shown, the three kinds of components (Model, View and Controller) can be applied in different devices over the network.

The working progress in NSAF in this MVC model can be described as below: first, the controller detects one external events (for example, a mouse click on a button); then the controller translates this action event into internal messages; after the translation, the controller publishes this message to models and views of the specific subscriber. If the subscriber is remote, then the message is pushed to it via WebSocket and P/S Middleware; next, the model (Resource Management) accepts the message, processes corresponding action according to the message and collects the result. Once getting the result, the model sends to the controller; the controller then publishes the result to the view of specific subscriber. In the end, the subscriber receives the message and updates its presentation.

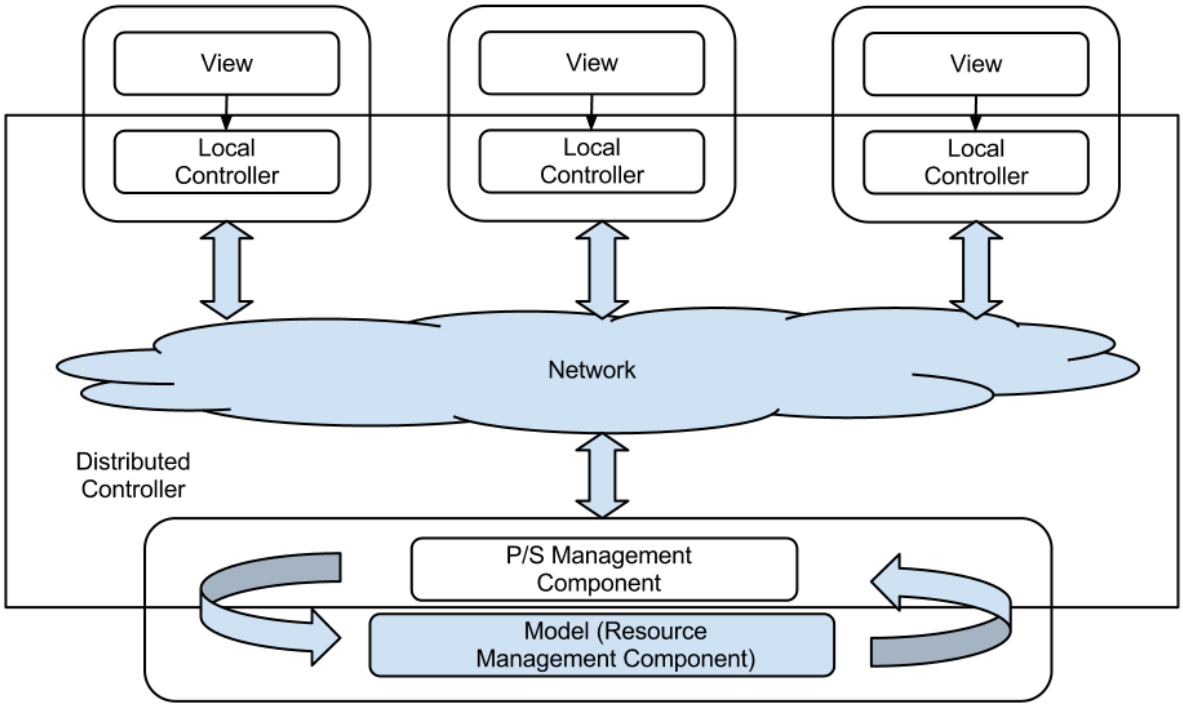


Figure 3.12: The Distributed MVC Design of NSAF Architecture

3.5 RESTful WS Resource Representation Design

As mentioned in Section 4.4, the NSAF uses a Session Management component to manage sessions, as well as a User Management component to manage the user profile and preference (which resource should be displayed on which type of devices). Devices from the same user are considered to use the same session in order to make sure it does not mess up with devices from other users. Those two components are managed by Web Services. Since this connection is always initialized from the user side device and communications are always based on user side request, instead of merging it with the pushing based message management system, the server holds a RESTful Web services for user and session management, as well as initializes and passes the

resource representation information to the client. The resources are represented by URLs over network. The server URL contains different levels.

3.5.1 Resource Representation via URL

Domain URL (DOURL)

The server's domain looks like following:

`http://www.example.com/`

Session URL (SURL)

In NSAF, every device logged on as the same user will be considered as viewing the same session. According to the design principle of RESTful Web service, the session is represented as a resource on the network. An example of Session URL is:

`http://www.example.com/sessions/`

Session Resource URL (SRURL)

When a device tries to initialize a new session, it sends an HTTP POST request to the SURL. Then a new session id will be created which will be shared among devices viewing the same resource content. The session ID is a unique ID number. And the resource URL is called Session Resource URL (SRURL). For example:

`http://www.example.com/session/0230562/`

Device Resource URL (DRURL)

The first device viewing this session will automatically receive a device id. The resource is also represented by a URL named Device Resource URL (DRURL). It looks like:

`http://www.example.com/session/0230562/12345/`

3.5.2 Manipulation of Resources through the Standard HTTP Methods

Session URL (SURL)

SURL will only receive one HTTP method, POST.

- **POST:** An HTTP POST sent to SURL creates a new SRURL with a unique ID.

Session Resource URL (SRURL)

SRURL will receive three HTTP methods, GET, POST and DELETE.

- **GET:** The server returns a list of active devices viewing this session.
- **POST:** If the authentication is successful, the server creates a new DRURL for the new device and returns the DRURL as the response. If the authentication fails, the server returns an error message.
- **DELETE:** The server deletes the SRURL if there is only one active device viewing this session. If more than one device is viewing the current session, the server returns an error message.

Device Resource URL (DRURL)

DRURL will accept three HTTP methods, GET, DELETE, and UPDATE.

- **GET:** The server returns the display content as a XML tree.
- **DELETE:** The server deletes this DRURL and mark the device as non-active in the database.
- **UPDATE:** The server updates the display content.

When a new device is trying to connect to an existing session, an HTTP POST request will be sent from the client-side application to the session URL(e.g., http://www.example.com/session/0230562/) with the authentication information, which will be checked by the server. If it matches with the current session, a new DRURL will be created and sent back to the client-side application.

When a device wants to change the display content, it will send an HTTP PUT request to the DRURL with corresponding change information in the PUT request entity.

When a device wants to leave the session, it will send an HTTP DELETE request to the DRURL. Then the server will delete the resource for the corresponding device.

3.6 Summary

In summary, the NSAF provides the following features to **solve** the challenges listed in the Problem Definition chapter:

- **For Challenge 1. Distributed MVC Architecture**
  - **Messaging System:** Messaging is a suitable and reliable mechanism for a distributed system to substitute method-based communications between components in classic MVC architecture. One component publishes a message to a shared message channel, others can read it from the channel at a later time. It provides loose coupling among components in the distributed system.
- **For Challenge 2. State Consistency Among Many Devices:**

- **Server Push:** When the server updates the server side resources, the update message is pushed to the corresponding resource subscribers without any client side query. While receiving the update message, the subscriber updates its corresponding representation automatically without any user actions (such as a click on the refresh button).

- **For Challenge 3. Bandwidth:**
  - **Publish/Subscribe(P/S) Pattern:** The P/S pattern significantly reduces wasted network bandwidth consumption, while comparing with two traditional data dissemination technologies, either broadcasting to every alive device on the network and letting the receiver decide whether to drop or process the message, or the client side application periodically queries the state and initialize the update connection.
  - **Result Optimization:** Result optimization reduces the size of the service results. Unnecessary data from the original resource state are trimmed, thus reduces the bandwidth used to interact with WS.

- **For Challenge 4. Web Standard Compatibility:**
  - **Browser-based Application:** By using WS, not only the native application can easily create, query, and delete resources on the network, but also do those Web browser-based applications. In NSAF, besides supporting the native application built for smartphones, tablets and personal computers, we also build web-based applications running in browsers enabling the NSAF to support as many devices as possible with minimum effort.

Chapter 4

Experiments

This chapter evaluates the performance of the NSAF with various loads. These experiments aim to judge whether the solution of NSAF reaches its goal, which is to provide seamless user experience among multiple devices mentioned in Chapter 2, and examine the performance and portability of the system by analyzing and evaluating the result.

4.1 Experiment Goal

In this section, I introduce three major tests according to the problems in the chapter 2. The portability test judges whether the NSAF framework is able to work with multiple devices running on different platforms. The data dissemination test tries to demonstrate that, compared with traditional approaches, the data dissemination technologies used in NSAF can provide faster real-time information update and reduce network bandwidth consumption, which are two key issues described in the problem definition chapter. The Table 4.1 presents a series of experiments evaluating the design of NSAF regarding the research challenges in chapter 2.

Research Challenges	Experiment
Resource Information Representation	Service and Portability Test
Distributed MVC Architectur	Service and Portability Test
State Consistency Among Many Devices	Bandwidth consuming experiment for Push vs. Pull
Bandwidth	Bandwidth consuming experiment for P/S and Result Optimization
Portability	Service and Portability Test

Table 4.1: The relation between experiments and research challenges

4.2 Experiment Setup

As mentioned in the design chapter, the NSAF system is deployed on many devices with different operating systems. Following systems are used during the whole experiment process:

- **Android Platform** In order to test the portability of the device on Android system, a few devices with different Android platform versions and different resolutions are going to be tested. They include:

  - **LG P925G Optimus 3D Smartphone** The Android platform running on this phone is 2.2.2 and the kernel version is 2.6.35.7, which is a comparatively old but popular version in smartphones market. The specification is shown as Table 4.2:

Hardware	Specification
System	Android 2.2.2 (Froyo)
Display	Size 480 x 800 pixels, 4.3 inches
Processor	Dual-core 1GHz ARM Cortex-A9 processor
Memory	512 MB RAM

Table 4.2: Hardware Specification of LG P925G Optimus 3D Smartphone

  - **Google Nexus 7 Tablet** This is a a 7-inch 1200x800 HD tablet designed and developed by Google in conjunction with Asus. The operating system running on Nexus 7 is the newest Android system 4.1.2 named Jelly Bean. The specification is in Table 4.3

Hardware	Specification
System	Android 4.1.2 (Jelly Bean)
Display	Size 1200 x 800 pixels, 7 inches
Processor	1.3 GHz quad-core Cortex-A9 processor
Memory	1 GB RAM

Table 4.3: Hardware Specification of Google Nexus 7 Tablet

  - **Acer Iconia Tab A500** This product is a tablet developed by Acer. The operating system running on this phone is 4.0.3 and the kernel version is 2.6.39.4+. Compared with the Android smartphone used above, this tablet has a larger screen resolution and newer version Android platform. The detail hardware information is shown in Table 4.4:

- **BlackBerry OS 7** Besides the Android based mobile devices, the system is also going to support BlackBerry devices with platform OS 7, which is the newest BlackBerry smartphone platform. The smartphone for experiment is BlackBerry Bold 9900 with specification showed in Table 4.5:

- **Windows** The performance of the application running on the Windows platform will be examined on a ThinkCentre desktop machine. The machine's hardware specification is listed in table 4.6.

Hardware	Specification
System	Android 4.0.3 (Icecream Sandwich)
Display	Size 1280 x 800 pixels, 10.24 inches
Processor	Dual-core 1GHz ARM Nvidia Tegra 250 processor
Memory	1.00 GB RAM

Table 4.4: Hardware Specification of Acer Iconia Tab A500

Hardware	Specification
System	BlackBerry OS 7.1
Display	Size 640 x 480 pixels, 2.8 inches
Processor	Qualcomm Snapdragon MSM8655 processor 1.2 GHz
Memory	768 MB RAM

Table 4.5: Hardware Specification of BlackBerry Bold 9900

- **Chrome Browser** As mentioned in the design chapter, in order to provide maximum portability support, NSAF also supports browser-based client. Since it supports the newest WebSocket specification RFC 6455 which is used in NSAF, Google Chrome Browser will be used in the experiment. There are two types of Google Chrome browsers: The desktop version which is Google Chrome Version 19.0 and mobile version Chrome Beta 0.18. Both of them will be set as the running environment in the experiment.

The resource server in NSAF will be held on a Macbook Pro with following specification in Table 4.7:

4.3 Service and Portability Test

4.3.1 Description

This experiment is carried out by running the client application on the devices listed above. The evaluation looks into whether the NSAF running on those devices gets the expected result. If the result on one client side device is expected, it means that NSAF system supports the corresponding device and the service is running correctly; While if not, the NSAF system does not support the device or the service is running incorrectly. According to the design chapter, the NSAF system with full functionality should be able to run on devices running with Android platform later than 2.1, BlackBerry OS 7, Windows and Mac OS machine which support Java Development Kit 6, as well as computer-like devices running a browser that supports WebSocket specification RFC 6455.

Hardware	Specification
System	64-bit Windows 7 Enterprise
Display	Size 1920 x 1200 pixels, 24 inches
Processor	Intel(R) Core(TM)2 Quad CPU Q9450 2.67 GHz
Memory	4.00 GB RAM

Table 4.6: Hardware Specification of ThinkCentre Desktop

Hardware	Specification
System	OS X Lion
Processor	Intel Core i7 Quad-core 2.4GHz
Memory	8.00 GB RAM
Hard Drive	750GB Serial ATA Drive 5400 rpm

Table 4.7: Hardware Specification of Macbook Pro Laptop

Since the major component that will affect the portability is the state synchronization connector (the push technology may be restricted to WebSocket specification RFC 6455), the test is divided into a client pull part and a server push part.

4.3.2 Result

The result of the Service and Portability Experiment is shown in the following form (Table 4.8).

Environment	NSAF with Client Pull	NSAF with Server Push
Android 2.2	Yes	Yes
Android 4.0	Yes	Yes
BlackBerry OS 7	Yes	No
Windows 7 Desktop	Yes	Yes
Chrome Browser Desktop Version	Yes	Yes
Chrome Browser Mobile Version	Yes	Yes

Table 4.8: Portability Test

As shown in Table 4.8, the NSAF is fully supported on most devices except BlackBerry OS 7, because the BlackBerry OS 7 only supports the Client Pull scenario. This is because the BlackBerry OS 7 only uses

Java Software Development Kit (SDK) 1.4 syntax, which was released in 2002, with limited standard Java Application Programming Interface (API) support. This increases the effort of porting the Java code running on the other types of machines (e.g. Android, Windows 7, Linux, Mac OS). After judging the implementation effort needed for supporting Server Push scenario on BlackBerry OS 7, I decide to only support Client Pull mechanism on it.

4.4 Data Dissemination (Push versus Pull) Test

NSAF uses WebSocket push technology as the message deliver mechanism transferring the real-time state of resource from the server to client side. As mentioned in the literature review chapter, the WebSocket push technology is considered as having faster data dissemination rate and lower bandwidth consumption compared with traditional pulling technology since the WebSocket eliminate the client pulling step in the pulling technology. The purpose of this experiment aims to demonstrate these two advantages of the WebSocket server push technology. It contains two parts: time consuming experiment and bandwidth consuming experiment. Both experiments will compare the push scenario with the traditional pulling scenario. To illustrate pulling, NSAF will use client models which request real-time data updating from the resource server periodically using a traditional pulling model.

4.4.1 Time consuming experiment for Push vs. Pull

Description

Since it is difficult to synchronize the machine time between two different devices in millisecond, instead of recording the One-Way Delay (OWD) time, which is the time that a message spends in travelling from device A to B, the experiment records the round trip time (RTT), which is the time interval between the sending of a message and the receipt of its acknowledgement [23], as in the commonly used PING utility [1]. In this experiment, it focuses on round trip time for a single update message travelling from its initial sender to the other client side subscribers. The path of this message includes sending from its generator to the central server, as well as propagating from server to corresponding subscribers. Since there is more than one travel station in this path, in order to record the precise RTT of a single message, every receiver (e.g. the central server, the client-side device) sends an acknowledge packet back to the last sender (e.g. the message generator, the central server). And the total RTT of the message is calculated as:

$$\text{the total RTT} = \text{the summation of RTTs between each nodes} \tag{4.1}$$

As discussed in the literature review chapter, there is a major difference between the push and pull technologies. For the push scenario, the server directly sends the message to the client when it receives an update message from the message generator, as shown in Figure 4.1. For the pull scenario, when the update message arrives at the server, instead of directly sending it to the client, the server first waits for the client

request and then sends the update, as shown in Figure 4.2. In this time consuming experiment, because the update message is not directly sent to the client pull scenario and the server also needs time to processing the update data, the time for an update message travelling from its initializer to its end consumer also includes the delay time on the server, which is calculated as:

$$\text{the total time} = \text{the summation of RTTs between each nodes} + \text{the delay on the server} \tag{4.2}$$

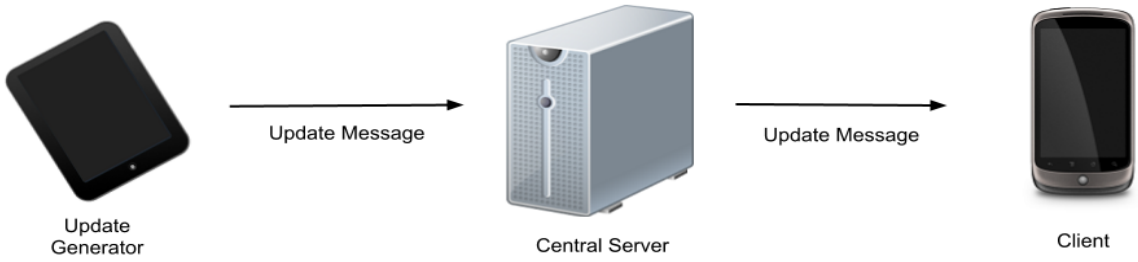


Figure 4.1: Server Push Scenario

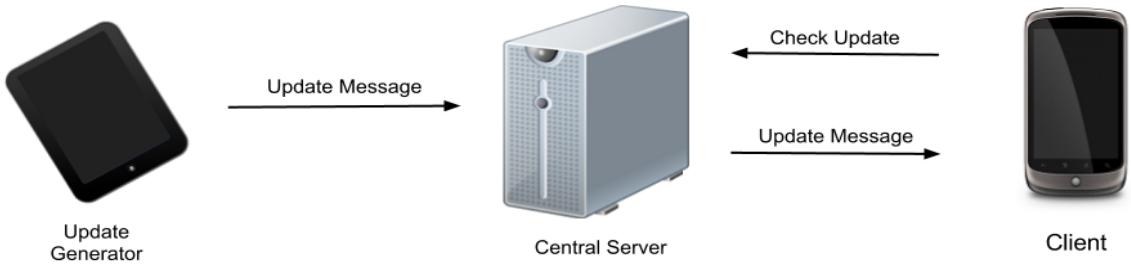


Figure 4.2: Client Poll Scenario

Result

The experiment result of each individual sample (50 samples) is shown in Figure 4.3. In this experiment, the time interval between each client pulling request is 2 seconds and the size of each message is close to 5 KB. As discussed in the literature review chapter, the expected result should demonstrate that the implementation of push technology improves the speed of data dissemination by reduction in extra latency. From Figure 4.3 we can see that the time consuming for a message travelling from its initializer to the end consumer in the server push scenario is much less than in the client pull scenario in most cases. The primary reason is because the update requests in the client pull scenario only happen every 2 seconds. If the update request is sent right after the update, both scenarios consume similar time. If the update request is sent just before the update, then the time consuming result of the client pull is almost 2 seconds more than the server push result since the next update request happen 2 seconds later.

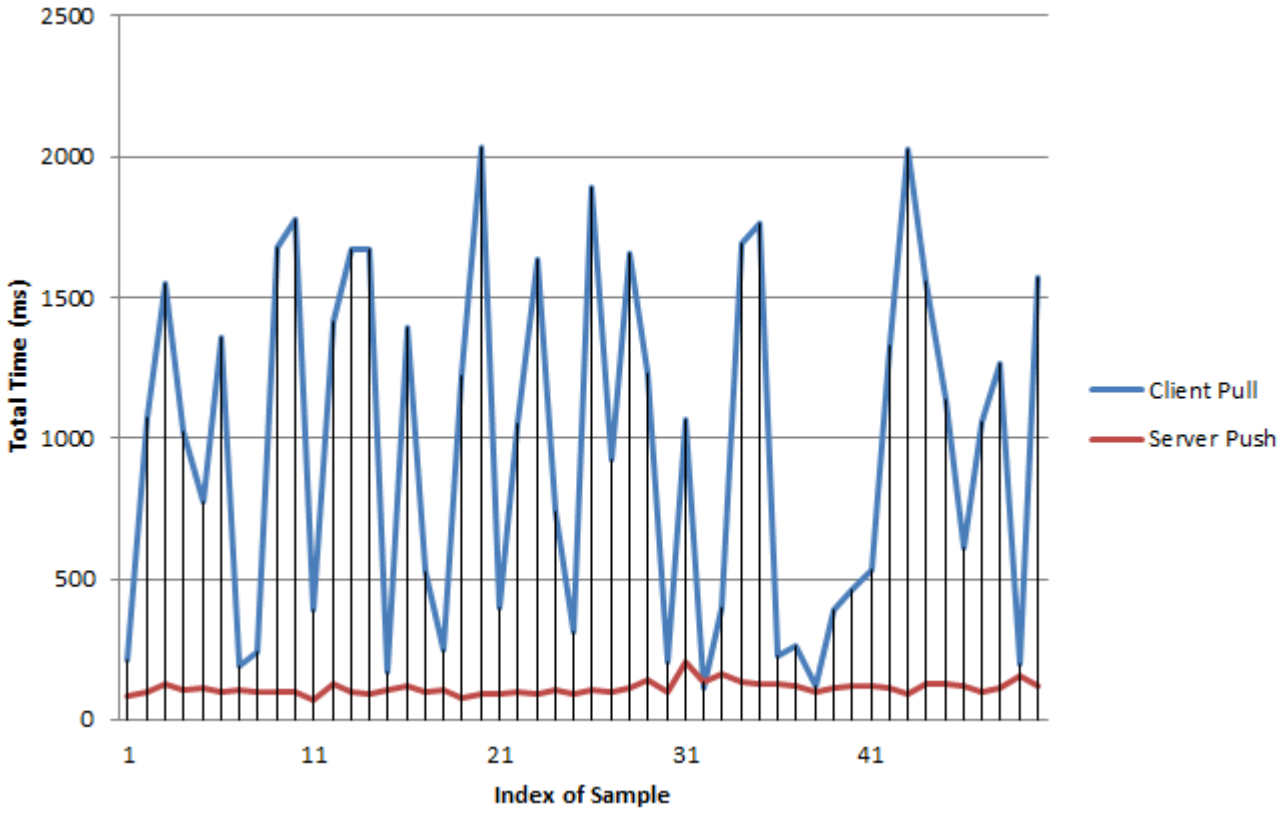


Figure 4.3: Time consuming experiment for first 50 samples: Client Pull vs. Server Push. The time interval between each client pulling request is 2 seconds and the size of each message is around 5 KB. The blue data series shows the total time for a message travelling from its initializer to the end consumer in the client pull scenario, and the red data series is for the server push scenario.

The experiment also evaluates the average result and standard deviation over the 50 collected samples to show that the push technology enhances the communication speed over the case using traditional client side pulling. The formula of calculating the samples average is:

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} X_i \tag{4.3}$$

The formula of calculating the standard deviation is:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i - \overline{x})^2} \tag{4.4}$$

This result is shown in the Table 4.9. From the table we could easily find that the average time used in the push scenario is 8 times less than the average time in the pull scenario, which demonstrates, comparing with the pull scenario, that the push scenario is a faster choice. The standard deviation shows how much variation exists from the average consuming time in each scenario. In this case, the push scenario collects a lower standard deviation. This indicates that data samples collected tend to be closer to the mean in push

scenario, whereas the data samples collected in pull scenario are spread out over a larger range of values. This is primarily caused by the update time interval between each client pulling for the pull scenario.

Data Dissemination Scenario	Property	Result
Server Push	Average message travel time	111.12 ms
	Maximum message travel time	202.91 ms
	Minimum message travel time	70.81 ms
	Standard deviation	22.59 ms
Client Pull	Average message travel time	968.99 ms
	Maximum message travel time	1987.43 ms
	Minimum message travel time	68.85 ms
	Standard deviation	613.42 ms

Table 4.9: Time consuming experiment: The message travel time starts from the update message sending from the client to the update message received by its corresponding subscribers

4.4.2 Bandwidth consuming experiment for Push vs. Pull

Description

As mentioned in the literature review chapter, the push scenario not only speeds up the transmission rate, but also significantly reduces the bandwidth consumption. This experiment tries to demonstrate a dramatic reduction in unnecessary network traffic in the NSAF compared with traditional polling model connectors.

The bandwidth consuming experiment measures the total data transferred for two end message consumers with different data dissemination mechanisms during the experiment, including both the update and download. The end message consumers are set as following:

- The Google Nexus 7 tablet with the client pull scenario which sends an update request to the server every 2 seconds.

- The Acer Iconia Android 4.0 Tablet with the WebSocket server push scenario.

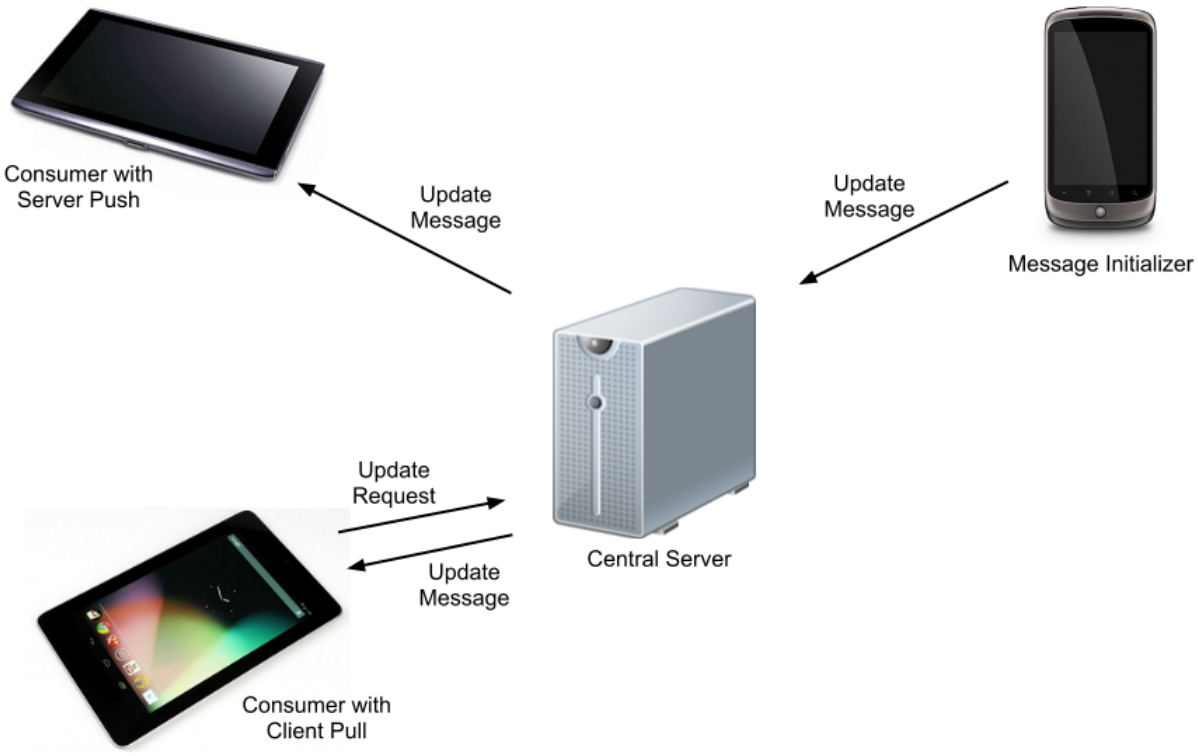


Figure 4.4: Bandwidth Consuming Experiment

As shown in Figure 4.4, all of the consumers subscribe the same channel which means the update messages they receive all have the same contents. The only difference is the message overhead (WebSocket header or HTTP GET header) and the data dissemination mechanisms (push or pull). The updater device sends update messages to the server randomly based on the user action. Since all of the consumers subscribe the same channel, those update messages will be finally received by the end consumers and both the download and upload rates on each devices will be evaluated.

Result

The experiment result is shown in Figure 4.5. It records the consumed bandwidth for every 2 seconds. In the pushing experiment, the client only uploads around 235 Bytes at the first 2 seconds and then 0 Bytes in the rest of the experiment, because it only needs to send the handshake message to server to initialize the WebSocket session at the beginning and then the server automatically pushes every update message to the client once there is a state change at the server side. In the pulling experiment, the client has a constant upload rate because it sends an update request message to the server every 2 seconds and the request messages are always the same. For the download rate, although both clients always receive the same message content, the download rate in the push scenario is less than in the pull scenario in every 2 seconds. The reason caused this phenomenon is because the WebSocket has a smaller overhead, which is only around 18 bytes, than a normal HTTP header (more than 100 bytes).

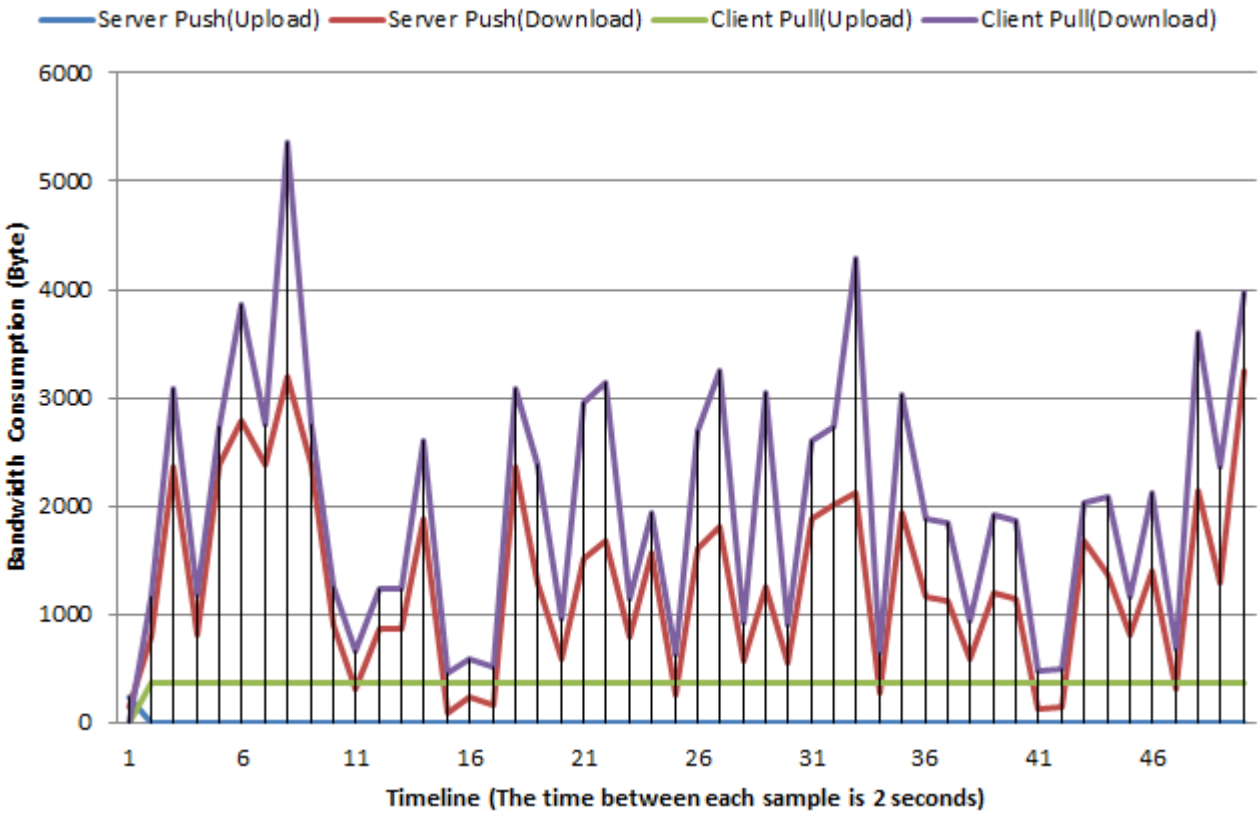


Figure 4.5: Bandwidth consuming experiment: Client Pull vs. Server Push. The experiment records the bandwidth consumption (both download and upload) at the client side for every 2 seconds.

Data Dissemination Scenario	Property	Result
Server Push	Total download bandwidth consumption	64.40 KB
	Total upload bandwidth consumption	0.24 KB
	Average download bandwidth consumption	643.98 Bytes/Second
	Average upload bandwidth consumption	2.35 Bytes/Second
Client Pull	Total download bandwidth consumption	99.28 KB
	Total upload bandwidth consumption	18.38 KB
	Average download bandwidth consumption	992.75 Bytes/Second
	Average upload bandwidth consumption	183.75 Bytes/Second

Table 4.10: Bandwidth consuming experiment: Comparison of unnecessary network overhead between the push scenario and traditional polling traffic.

The evaluation is in Table 4.10. The server push scenario has a superior upload advantage comparing with client pull. The client pull has a constant upload rate at 183.75 Bytes/Second, since it sends an update request message to the server every 2 seconds and 50 times in total during the experiment. Noted that this

bandwidth consumption can be reduced by increasing the pulling interval. However, as shown in the last experiment, it causes less frequent pulling which increases the update elapse time. The average download bandwidth consumption is 643.98 Bytes/Second in server push scenario, which is also less than the 992.75 Bytes/Second in the client pull.

4.5 Bandwidth consuming experiment for P/S and Result Optimization

4.5.1 Description

As mentioned in Chapter 2, besides server push, two additional approaches are used to reduce bandwidth consuming. Comparing with traditional broadcasting data dissemination technologies, the P/S pattern significantly reduces waste in network bandwidth, while the broadcasting scenario sends update to every alive device on the network and lets the receiver decide whether to drop or process the message. The result optimization approach reduces the size of service results by trimming unnecessary data from the original resource state.

This experiment tries to demonstrate the bandwidth consuming benefit provided by the P/S pattern and result optimization approaches. The experiment result is from a performance monitor function running at the server side to monitor the outgoing rate from the server. The transfer mechanism is server push which ignores the incoming bandwidth consumption to the server since it does not require clients to send an update request to the server. There are five client devices connected to the server. All of them are in the same session with a unique user but each of them subscribes a different portion of the resource.

4.5.2 Result

Scenario	Property	Result
P/S with result optimization	Average bandwidth consumption	470.84 Bytes/Second
P/S without result optimization	Average bandwidth consumption	1939.86 Bytes/Second
Broadcast with result optimization	Average bandwidth consumption	2354.2 Bytes/Second
Broadcast without result optimization	Average bandwidth consumption	6270.1 Bytes/Second

Table 4.11: Bandwidth consuming experiment: Comparison of unnecessary network overhead between the P/S approach with result optimization and broadcasting without result optimization.

In this experiment, I use the first 50 data samples of the upload bandwidth consumption at the server side in the first 100 seconds. The result is shown in Figure 4.6 and the average bandwidth consumption table is shown in Table 4.11. In the result diagram, the broadcast with no optimization function scenario,

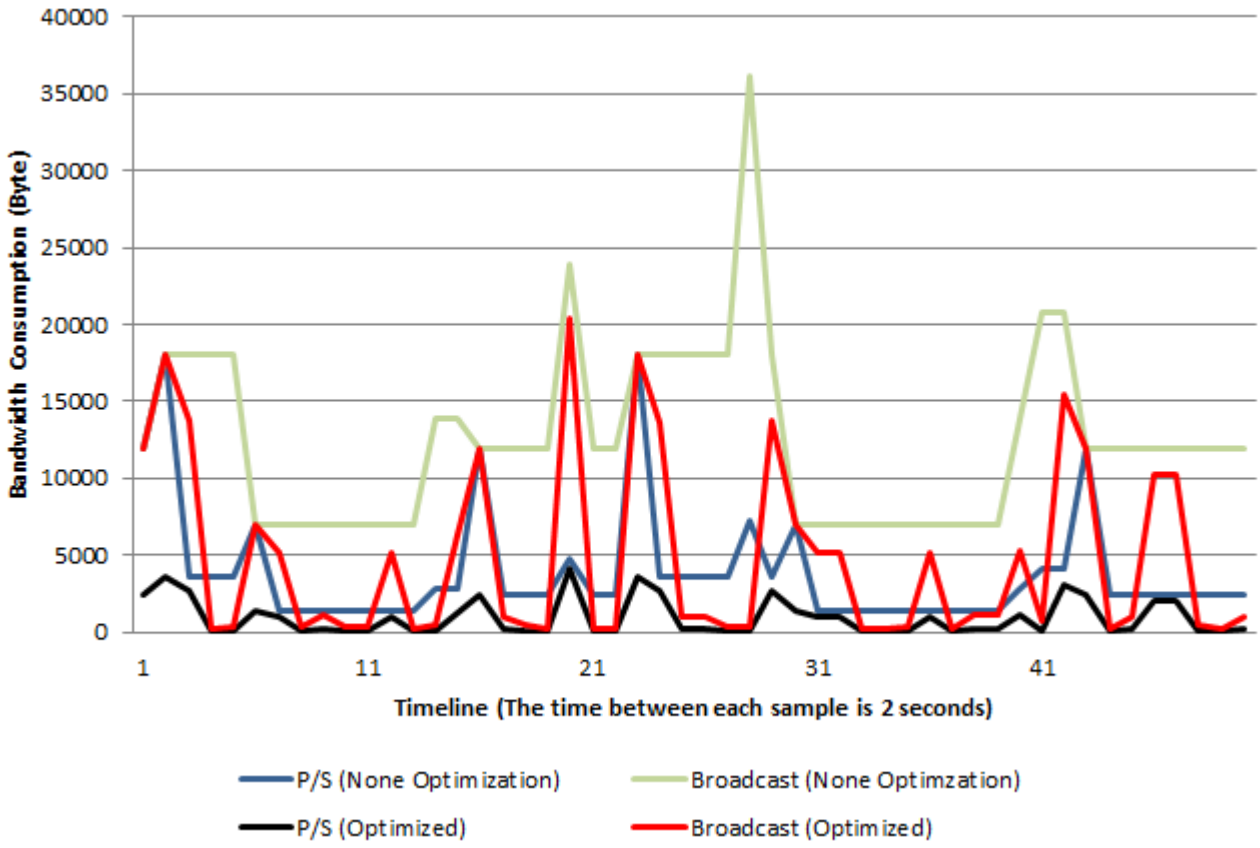


Figure 4.6: Bandwidth consuming experiment: P/S and Result Optimization. This figure shows the first 50 data samples of the upload bandwidth consumption at the server side in the first 100 seconds. The message contents during the four scenarios are the same.

which is the easiest scenario to be implemented, has the most bandwidth consumption. The experiment shows in one sample, while the outgoing message is two, this scenario uploads 36.1 KB to the client side devices. However, at the same time, the P/S without optimization function uses 72.2 KB, the broadcast with optimization function uses 0.365 KB, and the P/S with optimization scenario only uses 0.073 KB. The P/S with result optimization function, which requires the most implementation effort, has the best bandwidth consumption performance. The average bandwidth consumption of P/S with result optimization scenario is only 470.84 Bytes/Second, while the broadcast without result optimization scenario has 6270.1 Bytes/Second. The average bandwidth consumption results of the P/S without result optimization scenario and the broadcast with result optimization scenario are similar to each other. The P/S without result optimization has 1939.86 Bytes/Second consumption and the broadcast with result optimization has 2354.2 Bytes/Second. Even though from the average number, the second scenario (2354.2 Bytes/Second) has more bandwidth consumption than the first one (1939.86 Bytes/Second) in average. However, Figure 4.6 shows the first scenario has better performance in some samples. Experiment shows that when the size of update portion is smaller than the resource size divided by the number of connected client devices (e.g. lots of client devices are connected at the same time), the broadcast with result optimization scenario has a better

bandwidth consumption performance than the P/S without result optimization scenario.

4.6 Summary

Experiments in this chapter evaluated the NSAF framework from three aspects: the portability feature over multiple devices with different platforms, the time and bandwidth consumption performance in two data dissemination scenario: the Client Pull & the Server Push scenario, and two different functions built in NSAF to provide effective use of bandwidth: Publisher/Subscriber pattern and result optimization function.

Portability is an important factor for distributed applications such as NSAF to judge the possibility of working on many devices with different operating systems. In the portability experiment, the NSAF framework works as what it is designed on most devices except BlackBerry OS 7, since the BlackBerry OS 7 uses an old version of Java (Java SDK 1.4) and lacks many APIs that are used in the NSAF. The experiment result does not reject the possibility of implement NSAF in BlackBerry, however, it requires more effort to implement it than on the other devices. According to the timeline of this thesis and the future of BlackBerry (in the next generation of BlackBerry, it switches from Java to C++), I decide not to implement NSAF on BlackBerry.

Performance is another crucial factor for distributed systems. The performance of a network application usually focuses on two parts: the message travelling latency performance and the bandwidth consumption performance. In order to improve both the latency and bandwidth performance, NSAF uses the WebSocket push scenario to allow the server directly sending the update message to the client side; while traditional mechanism uses client sending an update request to the server and then the server response the request according to the update. The experiment result demonstrates that the WebSocket push scenario provides a dramatic improvement on both the latency and the bandwidth performance by simulating a full-duplex connection. Also, there is a notable phenomenon that the client pulling scenario consumes more battery power during the experiment since it initializes more TCP connections from the client side during the pulling step, and how to reduce battery consumption is a vital factor for designing applications running on mobile devices such as smartphones and tablets.

Besides the WebSocket server push scenario, two other approaches are developed to reduce the amount of unnecessary network overhead. As the resources are configured in the XML format, the result optimization function trims off the XML nodes for unnecessary information in the update message sending to the client to save the size of the message. The P/S pattern ensures that only the message subscriber receives the message. This also saves bandwidth comparing with traditional broadcast approach which every client in the system receives the message and they decide whether to process the message or drop it. The experiment shows significant reduction in bandwidth consumption while using these two technologies. In average, utilizing both of them can save more than ten times bandwidth consumption while comparing the scenario that uses neither the result optimization nor the P/S pattern. Experiment also shows the performance of just using

one of them are very close. None of the two approaches can be considered as being better than the other. Their result are relied on the number of connected devices and the number of XML nodes in the resource configuration file.

In conclusion, these experiments prove the advantages of the NSAF framework in portability that allows deployment on multiple types of devices from different manufacturers, scalability which is a significant property for distributed applications, and performance improvement while comparing with traditional data dissemination scenarios. The NSAF framework achieves its planned goals.

Chapter 5

Summary, Contribution & Future Works

This chapter sums up this research and discusses its general research contribution. In the second section, the possible expansion and future works are also discussed.

5.1 Summary & Contribution

The primary contribution of this work is to demonstrate a novel distributed MVC architecture pattern implemented in a message-based manner. As shown in Figure 3.11, NSAF utilizes message-based connector to exchange information between distributed components. The message-based connector is divided by four layers (Message Format Protocol, Publish/Subscribe Middleware, WebSocket, and TCP/IP) and each layer provides a unique functionality. This connector tightly combines the local controllers on the client side and the Publish/Subscriber Management Component on the server side to form a new distributed controller. All devices could understand messages from the others and those messages are only delivered to their corresponding consumers to save the bandwidth consumption.

In order to provide real-time message exchanging mechanism inside the distributed controller, several state synchronization scenarios are also reviewed and examined. By comparing it with existing data dissemination means, the NSAF selects a newer bi-directional, full-duplex web communication technology named WebSocket, which was introduced in HTML5. This approach expands NSAF's portability to browser-based environment as long as the browser supports WebSocket implementation. It also aims to significantly reduce the message travel latency as well as the unnecessary bandwidth consumption. What is more, experiment results show this novel bi-directional, full-duplex push technology also saves battery life of mobile devices, which is considered as an important factor for mobile applications, by reducing the unnecessary TCP/IP requests initialized from the client side.

Since the session is always initialized by the client side, instead of merging it with the pushing based Publish/Subscriber Message Management Component, NSAF provides RESTful Web Services to manage user profile, session's information and initialize resource representation information. The use of RESTful Web Services support and manage interoperable machine-to-machine interaction over the network. As leveraging the generality of standard Web interfaces, user information and resource representation are managed as resources over URIs and manipulated by standard HTTP verbs. It also gives a scalability bonus since the

server does not need to memorize the user state. While returning the resource information, RESTful Web Service allows the server to generate resource representation based on different devices, different requests and different user preferences. Unnecessary information will be trimmed to reduce the bandwidth consumption.

The experiments evaluate the portability, message travelling latency improvement and bandwidth optimization. The results of those experiments prove the advantages of the NSAF framework both in portability that allows deployment on multiple devices from different manufacturers and performance improvement (both in latency and bandwidth consumption) while comparing with traditional data dissemination scenarios.

To sum up, NSAF provides a systematically study and research in software architecture style design, data dissemination technology, information representation, and Web Services. By implementing NSAF, we have demonstrated a new way of building distributed application using classic architecture design pattern around explicit messaging. Meanwhile, we also realize that this research opens a door of a few major topics that need further digging which might lead to more important research contributions.

5.2 Future Works

There are few other desirable properties that could be added to NSAF in the future to improve its performance and user experience, as well as demonstrate the research value. In this chapter, we will give an outline for those possible improvements and research ideas.

- **WebSocket Version Support** In the proposed NSAF framework, it only supports the newest WebSocket specification RFC 6455 which largely restricts its compatibility among browsers. According to literature review chapter, as shown in Table 2.1, many browsers (such as Safari and Opera) only supports the WebSocket implementation using an older version of the protocol. And tests have shown that NSAF cannot initialize the WebSocket connection in those browsers. In order to support NSAF on those browsers, it is worthy to implement older WebSocket implementations.

- **Peer-to-peer support** Current NSAF uses a distributed connnector for communication among devices, but a centralized server-based service model in which the resource is kept on a centralized server and each update message needs to be sent to the server first and then pushed by the server to its corresponding subscriber. This increases the vulnerability of the entire system, since all clients highly rely on the existence of server, if the server fails, the whole system will be down. It also increases the travelling latency of the update message because all messages need to be sent to the central server first. Adding a peer-to-peer feature will eliminate those two drawbacks. Each device connected in the system can act as a client or server for the other computers in the network. The message could be directly sent from the update initializer to its end consumer to save the message travelling time. Also, if one node in the network fails, it will not affect the others.

- **Predictability** In distributed systems, a system is considered to be predictable if it faithfully executes

actions requested by the user on one device and provides a reasonable degree of explanation for any other events occurring in the other devices[30]. For example, as user browsing over the movie list and reading a movie's story line on the tablet, the computer connecting the TV which plays the movie should be able to display a trial of that movie and start to download the movie file which "explains" the fact that the user was interested in that movie and might watch it in next few minutes. Based on this prediction, the client side could start to download the movie content and cache it in the local storage. This approach could improve both the user experience as it can properly explain the user needs, as well as the system performance by predict user action and reduce the latency.

- **Consistency** In NSAF, to keep the ideal update consistency requires that all updates appear as atomic actions. In the current proposed NSAF, this provided by push technology and message cache on the server side. However, it is worthy to have a look at local cached model and data replication consistency to provide an improvement in both the performance and reliability.

- **Architecture Pattern** NSAF demonstrates one possibility of combining distributed system with classic architecture design pattern and using modern pushing technology as the component connector. There are still other design ideas around from other researchers. Some of them are discussed in the literature review chapter. Restricted by the research program length, those ideas are not implemented and compared with current approach of NSAF. In the future, it is a good research idea to re-design the NSAF architecture by utilizing other distributed architecture patterns with other pushing technologies. By comparing valuable properties of all those approaches, we could optimize the NSAF framework to provide the best user experience.

In conclusion, this research provides a fertile ground for many other research issues that could rise from current work. The work on NSAF could still be expanded.

References

[1] The story of the ping program. http://www.webcitation.org/5saCKBpgH, 2010.

[2] Best MVC Practices. http://www.yiiframework.com/doc/guide/1.1/en/basics.best-practices, 2012.

[3] Blackberry 7 documents: Web sockets api. http://docs.blackberry.com/en/developers/deliverables/29271/Web_Sockets_support_1582781_11.jsp, 2012.

[4] When can i use ...: Compatibility tables of websocket. http://caniuse.com/#feat=websockets, 2012.

[5] S. Acharya, M. Franklin, and S. Zdonik. Dissemination-based data delivery using broadcast disks. *Personal Communications, IEEE*, 2(6):50 –60, dec 1995.

[6] P. Alto. Google's android becomes the world's leading smartphone platform. http://www.canalys.com/static/press_release/011/r2011013.pdf, 2011.

[7] Engin Bozdag, Ali Mesbah, and Arie van Deursen. A comparison of push and pull techniques for ajax. In *Proceedings of the 2007 9th IEEE International Workshop on Web Site Evolution*, pages 15–22, Washington, DC, USA, 2007. IEEE Computer Society.

[8] Gaëlle Calvary, Joëlle Coutaz, and Laurence Nigay. From single-user architectural design to pac*: a generic software architecture model for cscw. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '97, pages 242–249, New York, NY, USA, 1997. ACM.

[9] I. Fette and A. Melnikov. The websocket protocol, rfc 6455. http://datatracker.ietf.org/doc/rfc6455/?include_text=1, 2011.

[10] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[11] M. Fowler. Presentation Model. http://martinfowler.com/eaaDev/PresentationModel.html, 2004.

[12] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[13] Michael Franklin and Stan Zdonik. Data in your face: push technology in perspective. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 516–519, New York, NY, USA, 1998. ACM.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, January 1995.

[15] D. Garlan and M. Shaw. An introduction to software architecture. ambriola & tortola(eds.). In *Advances in Software Engineering & Knowledge Engineering, vol. II*, pages 1–39. Word Scientific Pub Co., Singapore, 1993.

[16] J. Garrett. Ajax: A new approach to web applications. http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications, 2005.

[17] Gartner. Gartner says apple ios to dominate the media tablet market through 2015, owning more than half of it of the next three years. http://www.gartner.com/it/page.jsp?id=1626414, 2011.

[18] J. Gossman. Introduction to Model/View/ViewModel pattern for building WPF apps. http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx, 2005.

[19] I. Hichson. The websocket api. http://dev.w3.org/html5/websockets/, 2012.

[20] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[21] A. Hornsby. Xmpp message-based mvc architecture for event-driven real-time interactive applications. In *Consumer Electronics (ICCE), 2011 IEEE International Conference on*, pages 617 –618, jan. 2011.

[22] jQuery. jquery. http://www.jquery.com, 2012.

[23] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. *SIGCOMM Comput. Commun. Rev.*, 17(5):2–7, August 1987.

[24] H. Mcheick and Yan Qi. Dependency of components in mvc distributed architecture. In *Electrical and Computer Engineering (CCECE), 2011 24th Canadian Conference on*, pages 000691 –000694, may 2011.

[25] MSDN. Model-View-Presenter Pattern, MSDN. http://msdn.microsoft.com/en-us/library/ff647543.

[26] MSDN. Model-View-Controller. http://msdn.microsoft.com/en-us/library/ff649643.aspx, 2012.

[27] G. Muhl, A. Ulbrich, and K. Herrman. Disseminating information to mobile clients using publish-subscribe. *Internet Computing, IEEE*, 8(3):46 – 53, may-jun 2004.

[28] G. Mulligan and D. Gracanin. A comparison of soap and rest implementations of a service based interaction independence middleware framework. In *Winter Simulation Conference (WSC), Proceedings of the 2009*, pages 1423 –1432, dec. 2009.

[29] Netscape. An exploration of dynamic documents. http://www.citycat.ru/doc/HTML/Netscape/pushpull.html, 1992.

[30] W. Greg Phillips. Architectures for synchronous groupware. Technical report, 1999.

[31] M. Potel. MVP: Model-View-Presenter the taligent programming model for c++ and java. *Taligent Inc*, 1996.

[32] Xiaohong Qiu. *Message-based mvc architecture for distributed and desktop applications*. PhD thesis, Syracuse, NY, USA, 2005. AAI3177017.

[33] Xiaohong Qiu, Shrideep Pallickara, and Ahmet Uyar. Making svg a web service in a message-based mvc architecture. In *in Proceedings of SVG Open Conference, September 2004*, 2004.

[34] T. Reenskaug. MVC XEROX PARC 1978-79. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html, 1979.

[35] A. Rubin. Google+ post Andy Rubin. https://plus.google.com/u/0/112599748506977857728/posts/Btey7rJBaLF, 2012.

[36] J. Smith. WPF Apps With The ModelView-ViewModel Design Pattern. http://www.techguyonline.com/wp-content/uploads/2010/08/wpf_mvvm.pdf, 2009.

[37] W3C. Web services architecture. http://www.w3.org/TR/ws-arch, 2004.

[38] W3C. Soap version 1.2. http://www.w3.org/TR/soap12-part1/, 2007.

[39] Michael zur Muehlen, Jeffrey V. Nickerson, and Keith D. Swenson. Developing web services choreography standardsthe case of rest vs. soap. *Decision Support Systems*, 40(1):9 – 29, 2005. Web services and process management.

Appendix A

Implementation

A.1 Client Side Implementation

This section describes the implementation of client side. The data dissemination implementation on the client side is not covered in this section. It will be covered in the Section 5.3 Data Dissemination Implementation.

A.1.1 Mobile Client Implementation

The mobile client side includes implementation on the Android phones and tablets with SDK later than 2.3, as well as BlackBerry OS 7.

Android is a mobile operating system powered by a modified version of Linux kernel. It was first developed by a small company named Android Inc. and then by Google that bought Android Inc. in July 2005. Over 300 million Android devices were in use by February 2012[35]. It allows programmers to develop the code with the Java language and Google-developed Java libraries that are very easy for new developers to get start with it. However, it is only Java like development style.

BlackBerry is a series of smartphones developed by the Canadian company Research In Motion (RIM). Its major selling feature is instant, secure, mobile access to email. Development on BlackBerry devices are primarily based on the Java 2 Platform, Micro Edition (J2ME) and some BlackBerry Java APIs.

A.1.2 Desktop Client Implementation

The desktop client side implementation is primarily based on Java. There are two reasons for using Java. First of all, since the Android and BlackBerry developments are also strongly related to Java programming, some libraries can be shared between desktop client side application and mobile client side development, as long as those libraries are implemented by standard Java API. Secondly, Java inserts the Java Virtual Machine between the application and the computer environment allowing a single Java program to work on different platforms wherever Java is supported.

A.1.3 Browser Based Client Implementation

Besides mobile and desktop client implementation, the browser based client implementation can be deployed on different devices as long as the platform supports technologies used in the client application.

In NSAF, the browser based client application uses HTML5, CSS3 and JavaScript that are the next generation of the web technology. Those technologies refine the possibilities in web design and development. In HTML5, it improves the language with the support of the latest multimedia, WebSocket API, Geolocation API, Storage API and many other features. In CSS3, the "module"feature helps developers to handle CSS design in a flexible and convenient way.

Use of jQuery

The function in the browser based client application is implemented by Javascript and its library jQuery. jQuery[22] is a fast, concise and lightweight JavaScript Library which can run nearly all devices that support JavaScript. It provides cross-browser compatibility and simplifies HTML document traversing and event handling by providing the concept of selector. The rich animation library and Ajax interactions enable rapid web development for developers.

A.2 RESTful WS Session Management Implementation

The session management component is built in a RESTful WS way. As discussed in the literature chapter, RESTful WS design principles identify the key architectural principles of the reason why World Wide Web (WWW) is so prevalent and scalable. In NSAF, we will apply the concepts of RESTful WS by defining a set of distributed RESTful interfaces to manage session information and user preferences.

JAX-RS: Java API for RESTful WS is a framework that focuses on applying Java annotation concept, which was first introduced in Java SE 5 and later became a official part of Java EE 6, to plain Java objects. These annotations are used to bind specific URI patterns and HTTP operations to methods in the Java classes. The parameter injection annotations allow the program to easily pull the information from HTTP request. What is more, the message body readers and writers can even decouple data format marshalling and unmarshalling from the plain Java data objects.

Following shows some basic annotations in JAX-RS that can map the Java data object to a web resource:

- **@Path:** specifies the relative path for the resource in the URI
- **@GET, @PUT, @POST, @DELETE and @HEAD:** specify the HTTP method on the resource
- **@Produces:** specifies the response MIME media type
- **@Consumes:** specifies the accepted request MIME media type

There are several JAX-RS implementations available on line, such as Apache CXF, Jersey, RESTEasy, Restlet and Apache Wink. NSAF uses Jersey since it is the reference implementation from Oracle that provides stable update and support.

A.3 Data Dissemination Implementation

The data dissemination component contains two parts: a Publish/Subscribe system to ensure only subscribed information is going to be delivered to corresponding client device and a WebSocket transportation component to provide information and state synchronization.

A.3.1 Implementation of Publish/Subscribe System

When the RESTful WS creates a new session, a new channel is also created in the Publish/Subscribe middleware. It is the Publish/Subscribe channel for that session.

As mentioned above, the Publish/Subscribe channel work scheme is as following: It has one input channel, which is a queue as following:

```
private Vector<byte[]> queue;
```

This input channel splits into multiple output channels (multiple queues), one for each subscribers. This subscribe relationship is kept by a data structure, such as array or vectors. Once a message is pushed into the channel, the Publish/Subscribe channel delivers one copy of the message to each one of the output channels. Since each channel only has one subscriber, the message can only consumed once in that channel by the pop method of queue. Therefore, each subscriber reads the message once and then that message is deleted from the queue.

The queue has a worker thread to check whether the queue is empty or not. If it is not empty, the worker pops out the message and delivers it to corresponding transportation component, e.g. WebSocket component. The WebSocket component wraps the message into a WebSocket message according to its protocol. It then pushes the message to the corresponding clients. The worker thread is implemented as following:

```
public void run() {
    isRunning = true;

    while(isRunning) {
```

```
            if(queue.isEmpty()) {
                try {
                    Thread.sleep(SLEEP_INTERVAL);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            else {
                synchronized(queue) {
                    try  {
                        out.write(queue.remove(0));
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
```

A.3.2 Implementation of WebSocket

The WebSocket contains two parts: the client side implementation which initializes the WebSocket connection and a server side implementation which accept the handshake request from the client and exchanging message with WebSocket client.

Implementation of WebSocket Server

In NSAF implementation, the server monitors the port 5150 by creating a new ServerSocket object. Once the ServerSocket object is successfully created then it starts an accept method. This method waits until a client starts up and requests a connection on that port. Once the connection is successfully established. The accept method returns a new Socket object and that Socket is passed to a ClientThread class. The ClientThread contains two threads: one for reading bytes from the socket, the other one for sending bytes to the socket.

Before entering the data transfer phase, the client and server must properly establish the connection in an one step handshake process. As mentioned in literature chapter, an example of handshake message (Client: Google Chrome 17, Server: NSAF) is shown as following:

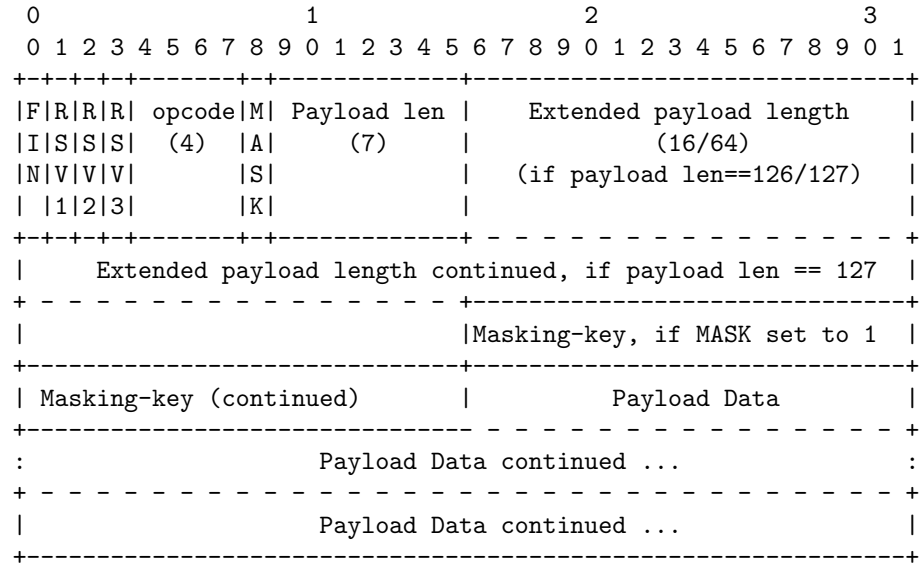
```
GET / HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: localhost:5150
Origin: http://localhost
Sec-WebSocket-Key: qfpSNmDK3K8BVtF1hhjmhA==
Sec-WebSocket-Version: 13

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
ec-WebSocket-Accept: G6RjYkDEC916Mt9iuSrdlaQgq/s=
```

The key factor to generate a successful handshake is the "Sec-WebSocket-Key" and "Sec-WebSocket-Accept" field. The value in the "Sec-WebSocket-Accept" is generated based on "Sec-WebSocket-Key". The server fetches the value in "Sec-WebSocket-Key" and concatenates it with the Global Unique Identifier (GUID) "258EAFA5-E914-47DA-95CA-C5AB0DC85B11". Then the server would take the SHA-1 hash on this value and encoded it with base64-encode. The result is the value in "Sec-WebSocket-Accept" field. In the implementation, this part is done by Java and utilizes the Apache Commons Codec API. Apache Commons

Codec software provides implementation of common encoders and decoders such as Base64, Hex, Phonetic and URLs. The server handshake information is generated as follows:

```
public static final String GUID = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
public static String sendServerHandshake(String request) {
    StringBuffer response = new StringBuffer();
    response.append("HTTP/1.1 101 Switching Protocols\r\n");
    response.append("Upgrade: websocket\r\n");
    response.append("Connection: Upgrade\r\n");
    response.append("Sec-WebSocket-Accept: " +
        WebSocketUtil.getHashKey(WebSocketUtil.getSecSocketKey(request)) + "\r\n\r\n");
    return response.toString();
}
public static String getHashKey(String key) {
    return Base64.encodeBase64String(DigestUtils.sha(key + GUID));
}
public static String getSecSocketKey(String request) {
    String fieldName = "Sec-WebSocket-Key: ";
    int index1, index2;
    index1 = request.indexOf("Sec-WebSocket-Key: ");
    if(index1 != -1) {
        index2 = request.indexOf("\r\n", index1);
        if(index2 > index1) {
            return request.substring(index1 + fieldName.length(), index2);
        }
    }
    return null;
}
```

After the connection is established, the WebSocket messages can be sent bi-directionally between the client and the server. As a result, the message can be sent full-duplex, in either direction at the same time. In Websocket, data is transmitted using a sequence of frames. A client must mask all frames when it sends messages to the server. The server must close the connection if it receives an unmasked frame from the client. A server must not mask any frames that it sends to client. A client must close a connection if it receives any masked frame from the server[9]. The frame structure is shown as below:



The FIN bit indicates whether it is a final fragment in a message. RSV1, RSV2, and RSV3 are reserved bits and must be 0. Opcode is 4 bits and it defines the interpretation of the payload data. MASK bit indicates whether the payload data is masked. If the value is 1, a masking key is presented in masking-key. As mentioned, all messages sent from client side must be set to 1 and all messages from server side must be 0. The "payload len" indicates the length of the payload data. If it is in the range from 0 to 125, the length of the payload is exactly the value in "Payload len". If the value is 126, the following 2 bytes will be translated to be a 16bit unsigned integer to be the length of payload. If the value is 127, the following 8 bytes interpreted as a 64-big unsigned integer with most significant bit to be 0. If the MASK bit is set, there will be a 4 bytes masking key. This masking key is used to do the XOR operation with the data in "Payload Data". The data should be encoded as UTF-8 format. In the implementation, the code to decode the payload data is shown as following:

```
StringBuffer payload = new StringBuffer();
while(index < lengthOfFrame) {
    if(isMasked) {
        payload.append(Charsetfunctions.stringUtf8(
            new byte[]{(byte)(frame[index] ^ b[(index - indexOfPayload) % 4 + 2])}));
    }
    else {
        payload.append(Charsetfunctions.stringUtf8(new byte[]{frame[index]}));
    }
    index ++;
}
```

To generate the frame of return message from the server is simpler since the data sends from the server must never be masked. The following example shows how to use Java generate a text message which is less than 126 bytes in server frame:

```
byte[] data = new byte[2 + text.length()];
data[0] = (byte)0x81;
data[1] = (byte)(0x7f & text.length());
byte[] tmp = Charsetfunctions.utf8Bytes(text);
for(int i = 0; i < tmp.length; i ++) {
    data[2 + i] = tmp[i];
}
```

Implementation of WebSocket Client

For Java based client, such as applications working on Android, BlackBerry and desktop computers, the implementation of WebSocket protocol is similar to the server side implementation, since they both interpret messages within a same frame; therefore, we will primarily focus on using JavaScript to implement WebSocket client and use it in browser-based applications.

WebSocket is a standard communication feature in the HTML5 specification[19]. It has well defined APIs for standard HTML5 JavaScript and can be easily used. For those web browsers which support WebSocket, it is the browser developers duty to implement the connecting half of the WebSocket Protocol.

During the loading of the page, the browser based NSAF client checks if the browser supports the WebSocket API and this is done by jQuery:

```
$(function() {
    if(window.WebSocket) {  // If WebSocket is supported
    ...
    }
    else {  // If not, pop a warning message
    ...
    }
}
```

To create a connection to the server, we just need to create a new WebSocket instance with a URL representing the server. "ws://" is used to indicate it is a WebSocket connection.

```
url = "ws://192.168.100.100:5150/";
socket = new WebSocket(url);
```

Implementation of WebSocket Client

Once connected, the client side application can simply wait for events without polling the server anymore. A set of well defined callback functions are used to listen for events. A WebSocket object can dispatch four events: open, close, error and message.

```
socket.onopen = function() {
    // Open connection call back
}
socket.onmessage = function(event) {
    // Message received call back
}
socket.onerror = function(event) {
    // Error call back
}
socket.onclose = function(event) {
    // Close call back
}
```

When the connection is open, the send function can be used to send messages, as shown in NSAF implementation:

```
socket.send($("#inputBox").val());
```

This section shows how NSAF implement the communication in WebSocket and this provides a simple, yet powerful way for creating real-time applications. I also provide a guide to implement this protocol on the server side and show how easy to use it on a browser based client side application.

A.4 Summary

This chapter describes specific technologies used in the implementation of NSAF. Section 5.1 introduces how the client side implementation can provide portability to NSAF that allows it running on many kinds of devices. The client side application can be classified into three categories: mobile client application, PC client application and browser-based application. The mobile client implementation primarily focuses on two types of devices: Android and BlackBerry 7 phones. The PC client application is implemented by Java since the JVM is able to run on different PCs. The browser based application utilizes the newest Web technologies such as HTML5, CSS3 and jQuery to provide fast and user-friendly interfaces to the end user. Section 5.2 describes how to use JAX-RS to implement a RESTful WS Session Management component. A set of WS APIs will be defined by using Oracle's Jersey library to allow remote connection and manipulations on session information and user preferences. Section 5.3 is divided into two parts. The first part describes the implementation on Publish/Subscribe middleware system that delivers information to its corresponding subscribers. The second part gives an example of WebSocket push implementation to show how to achieve the goal of state synchronization.