Widget Communication Architecture with Emphasis on Mobile Devices

Yesheng Xu
Abstract

**Widget Communication Architecture with Emphasis on Mobile Devices**

Yesheng Xu

Small applications called widgets have been widely accepted on desktop and are on their way to be successful on mobile devices too. Widgets are usually managed by the widget runtime engines. This thesis considers that parts of the widget logic can be migrated to run in the network (for example, due to limited mobile device resources or other design reasons). In this case the mobile part of widget logic has to communicate with its network counterpart. It is also preferable to allow the mobile widgets to communicate with each other. Hence widget communication becomes an important aspect of the widget engine architecture. This thesis investigates the possibility of applying RESTful paradigm to the widget communication and proposes a corresponding architecture along with the prototype implementation.

Key Words: Widget, Mobile 2.0, Widget Communication, REST Framework, Mobile Web Server, Mobile Web Services.
# Table of Contents

1 Introduction ............................................................................................................... 3
  1.1 Background ........................................................................................................ 3
    1.1.1 Widget ........................................................................................................ 3
    1.1.2 Mobile Widget ............................................................................................ 3
    1.1.3 Mobile Web Services .................................................................................. 4
    1.1.4 Client-Server Dual Role Mode ................................................................. 4
    1.1.5 Mobile 2.0 ................................................................................................... 5
  1.2 Thesis Motivation ............................................................................................... 5
  1.3 Thesis overview .................................................................................................. 5

2 Related work ............................................................................................................... 7
  2.1 Mobile Widget Engines ...................................................................................... 7
  2.2 Mobile Web Services .......................................................................................... 7

3 Mobile Platform limitations ........................................................................................ 9
  3.1 Mobile Devices Performance .............................................................................. 9
  3.2 Mobile Platform programming ........................................................................... 9
  3.3 Mobile Addressing .............................................................................................. 9
  3.4 Machine communication on mobile .................................................................. 10
  3.5 Push mechanism ................................................................................................ 11

4 Widget Communication Architecture ....................................................................... 13
  4.1 General Ideas of Widget Communication ......................................................... 13
    4.1.1 Widget Runtime Network ......................................................................... 13
    4.1.2 Widget Communication Modes ................................................................ 13
    4.1.3 External and Loopback Communication .................................................. 14
    4.1.4 Cross Platform Communication ................................................................ 15
  4.2 Technologies for Widget Communication ........................................................ 15
    4.2.1 Technologies Overview ............................................................................ 15
    4.2.2 RESTful Approach .................................................................................... 15
    4.2.3 Restlet, Jersey and JSR-311 ...................................................................... 16
  4.3 Widget Communication Framework ..................................................................... 17
    4.3.1 Mobile Platform Three Layer Design ....................................................... 17
    4.3.2 Server Three Layer Design ....................................................................... 19
  4.4 User Group ........................................................................................................ 20

5 Connectivity Layer Architecture ............................................................................... 21
  5.1 Architecture Overview ...................................................................................... 21
    5.1.1 Server Connectivity Layer ........................................................................ 21
    5.1.2 Mobile Connectivity Layer ....................................................................... 22
  5.2 Server Servlet Adapter ...................................................................................... 23
    5.2.1 Working Mechanism ................................................................................. 23
    5.2.2 Request Response Transformation ........................................................... 23
  5.3 Tunnel Gateway ................................................................................................ 24
    5.3.1 Tunnel Gateway Architecture ..................................................................... 24
    5.3.2 Addressing Widgets .................................................................................. 25
    5.3.3 Comet Push and Connection Keep-alive .................................................. 26
    5.3.4 Message Flow ........................................................................................... 27
  5.4 Mobile HTTP Server ........................................................................................... 29
1 Introduction
This chapter introduces the background of mobile widget communication. After describing these notions, a short definition of thesis purpose will be given, followed by a thesis overview. The thesis overview will give a short preview of the following chapters which chase after the same purpose throughout the thesis.

1.1 Background

1.1.1 Widget
Widgets are lightweight applications that are designed for single specific tasks. Lightweight characteristic of widgets indicates that widget applications are both small in byte size and easy to implement. Designing for only specific tasks makes widgets loosely coupled applications which are highly reusable.

Widgets are deployed and run by widget engines. Widget engines are available on three kinds of environment, desktop, web and mobile. Apple Dashboard and Microsoft Gadgets are well known desktop widget engines.

Figure 1.1.1.a Widgets on Mac OS dashboard
Figure 1.1.1.b Widgets on Windows Vista Sidebar

Widgets are popular among developers because of ease of development. They are small in size and most of them can be developed with several lines of XML or scripting language. Users prefer widgets for the reason that they consume less system resources and they are highly usable.

According to the World Wide Web Consortium (W3C) [1], widgets are web application based on web technologies such as HTML, CSS, JavaScript and XML. However, from the thesis point of view, a widget is not necessary web based. Although some widgets might be able to access internet contents (including web services and other widgets), this is not a must for all widgets. Specifically, the process of accessing the web by widget in this thesis is called widget communication.

1.1.2 Mobile Widget
Desktop widgets and web widgets have been highly successful along with the notion of Web 2.0 for years. Mobile widget is comparative new, but with larger potential. The point is that mobile devices are restricted by the limitations of their bandwidth, screen size, processor, memory, etc., which means large-scale applications are not suitable for running on these devices. Widgets, on the other hand, known by their light weight and task specified are ideal form of applications for mobile devices.

1.1.3 Mobile Web Services

The success of web services has dramatically changed the overall structure of the internet applications. The good thing about the web services is that they introduce a well structured way of inter-machine communication. Today’s web applications, or should say components, focus only on specified task and get the rest from the web services.

Although web services have been widely deployed on the internet, the supporting rate of mobile device accessing those services is rather low. This is mainly due to the fact that the traditional approaches of web service identified by well known technologies such as XML (eXtensible Markup Language) and SOAP (Simple Objective Access Protocol) are considered to be too heavy to run on mobile device. This might no longer be the case in the near future for that the hardware performances of mobile devices increase continuously every year. Mobile devices will eventually be able to run large-scale applications. At present, however, the traditional technologies for web services can still be substituted by some new lightweight technologies to reduce the requirements on mobile device performance.

A comparative lightweight web service approach called REST (REpresentational State Transfer) has currently been popular. The idea of REST is that it makes full use of existing HTTP stack rather than adding an additional layer of messaging format. This reduces the size of message and the complexity of using web services. Moreover, REST is not bound to any specified data format. Instead of using XML, REST applications can choose more lightweight data format e.g. JSON (JavaScript Object Notation).

This thesis is even more ambitious than just describing an architecture for accessing web services on mobile. The more desirable feature of mobile widget communication is that mobile devices themselves can act as servers providing services.

1.1.4 Client-Server Dual Role Mode

Some network architectures enable nodes in the network act as both client and server. A peer in P2P (peer-to-peer) architecture is one of those kinds. We can call that kind of mode: client-server dual role mode. The concept behind that is that
every node in the network can use services as well as provide services. Thus the network can generate services and increase the value of the network. We can imagine that widget communication architecture using that mode will dramatically enlarge its potential.

The portability of mobile devices gives them some natural advantage over the desktop PCs. Mobile devices can be equipped with camera, GPS or any kind of sensors. This equipment can provide some valuable information that can be used as served resources. By applying web service interface to widget engine, mobile widgets can be peer nodes on mobile network, thus add value to the whole network. For example, a GPS cell phone with a temperature sensor can detect local temperature and send it to others. The same cell phone with a carbon dioxide sensor can detect the local air pollution level. Suppose we have many those kinds of phones in the city. All these services from the phone, if collected together, can provide a complete graph of the city’s temperature or pollution status.

1.1.5 Mobile 2.0
The notion of Mobile 2.0 is derived from that of Web 2.0. Nellymoser [2] described mobile 2.0 as: “Mobile 2.0 Services integrate the social web, or what some call Web 2.0, with the core foundations of mobility.” With no surprise, most of the implementations of Web 2.0 cannot apply directly into mobile device due to the difference in platforms. But the ideas and the rich accumulation of the internet content more or less can be reused on mobile devices. The challenge of Mobile 2.0 is to find out how Web 2.0 features can be applied to mobile devices in an optimized way.

1.2 Thesis Motivation
The purpose of the thesis is to investigate a Mobile 2.0 compatible architecture for the mobile widget communication. Building on the top of mobile widget engine, mobile widgets will be able to:

1) Communicate with each other;
2) Access web services on internet;
3) Access other internet contents.

Communication follows client-server dual role mode. This will lead to a serious of concrete problems, which will be put into a deep discussion in later chapters.

1.3 Thesis overview
Some related work on mobile widget engines and mobile web services will be introduced in chapter 2. In chapter 3, possible limitations and obstacles of the architecture will be investigated. Chapter 4 gives an overview of the widget communication architecture design, followed by the detail design description in
Chapter 5-7. Chapter 8 draws the conclusion and opens the discussion of the architecture flaws and states what can be improved in the future work.
2 Related work

This chapter introduces related work in two scopes: mobile widget engine and mobile web service. The thesis work is closely related to these two scopes.

2.1 Mobile Widget Engines

Several companies have developed widget engines in the last couple of years. Most of them have almost the same approach following the widgets 1.0 standard Defined by W3C [1]. The W3C definition suggests that widgets are developed using standard web technology such as HTML, CSS, JavaScript and XML. Therefore widget engines act more or less like web browsers. Apart from following W3C standard, most widget engines also choose MIDP 2.0 as platforms. The main consideration of this is that MIDP is popular, easy for implementation and device independent.

Widsets [3] is one of the dominant mobile widget runtime engine based on MIDP 2.0 launched by Nokia. Like other widget engines, Widsets widgets employ XML and CSS like syntax, a scripting language called Helium and also generic knowledge about RSS or Atom technology for syndication widgets. Now, Widsets can run on over 300 phone models with around 4000 widgets available.

Another successful widget engine is Yahoo! Go engine [4]. Yahoo! Widgets integrate user interfaces with backend services taking advantage of accessing of Yahoo! Web services. Under the user experience, Yahoo! uses declarative XML based language called Blueprint for UI design rather than scripting language. According to Yahoo!, Blueprint makes widgets possible to be run on hundreds of devices.

iPhone is yet another well know widget engine implementation. Although the engine is tightly embedded in the hardware system, iPhone applications perform like widgets. iPhone widgets are normally considered as well designed and have many supporters worldwide.


2.2 Mobile Web Services

Some efforts have been made to explore the way of accessing web services from mobile devices. Oscar Mauricio and his colleges’ work [6] “Architectures for Web Services Access from Mobile Devices” introduce architecture for mobile devices web service accessing. A lot of works done on mobile AJAX also introduce web based architecture for mobile web services accessing.
Johan Wikman and Ferenc Dosa published the article [7] “Providing HTTP Access to Web Servers Running on Mobile Phones” from Nokia research discussed the possibilities of mobile web services.

Nokia has released their mobile web server [8] as experimental products currently on Nokia S60 series phones. The mobile web server is also known as Raccoon, consists of a port of the Apache web server and connectivity solution that provides a mobile phone with a global URI and with HTTP access to it.
3 Mobile Platform limitations

Before looking into architecture design, there are several constraints to be considered. This chapter will explore these considerations and analyze the potential limitations which may influence design strategies.

3.1 Mobile Devices Performance

Mobile devices are normally considered to have worse performance than PCs. The major concern is in the aspects of low processor frequency, low memory capacity, low bandwidth, and small screen size. Low processor and memory limit mobile devices’ calculation ability means that mobile devices are not suitable to run large-scale applications which require a lot of computation. Low bandwidth makes mobile devices slow to access internet which causes delay and reduces usability. Small screen size makes screen display difficult to read and increases difficulties of developing UI. All of these make it difficult to program. Special steps must be taken to optimize performance of mobile applications.

3.2 Mobile Platform programming

Due to the limited platform performance, programming for mobile phones must be optimized. Java language Mobile Edition has less classes and interfaces than standard Java edition. Code size, use of memory and number of classes are limited on mobile devices. Creation of threads should also be carefully planned. Programming strategies and design patterns will consequentially be different from that of desktop platform.

3.3 Mobile Addressing

Mobile devices run in the mobile operator networks are allocated only private addresses. The entire address space of an operator network is hidden behind a single public IP address. Any device from the operator network is thus not addressable in the external network. When a mobile device want to access some resources on internet, a mechanism called NAT (Network Address Translation) must be used to translate a private address into a public one. The mechanism is implemented in a routing device that uses a stateful translation table to map the private address to the public one. However, NAT only allows traffic that originates from internal network. It does not work for traffic originates from external network. The same situation happens when an operator network is behind a firewall. The firewall will block any attempted traffic from external network to internal.
In widget communication architecture, there are mobile widget engines running on mobile devices that are behind firewalls. There are also server widget engines running on the internet that can be considered as external network. Widgets on any engine should be able to talk with any other freely. Thus a mechanism should be provided to walk around NAT and firewall, known as NATs traversal.

There are many ways to perform NATs traversal. One of them is to design a bridging system that establishes a bridge between mobile device and the external network. As long as the connection is kept alive, traffic can go through the connection both ways.

3.4 Machine communication on mobile

How to enable web service efficiently on mobile is the main concern of the thesis. It leads to several questions:

1) What web technology shall be used in mobile machine communication?
2) To what extent shall the chosen web technology be applied?
3) What optimizations shall be carried on?

For performance, the web technology chosen should be lightweight and resource saving. Traditional web technology like, for example, SOAP uses XML message format which costs a lot of system resources to parse and bandwidth to transfer over the network, and is less suitable for this purpose. In the precondition that a web technology is chosen, it is not necessary all the features will be reapplied. Certain cutting and optimization can be carried on according to the requirements.
3.5 **Push mechanism**

Communication strategy is another issue in the widget communication. The classical request-response strategy works well in most of the situations. But it is not enough in some cases that requires low latency, real time data accessing such as real time exchange rates, real time stock price, status notification, etc. Push mechanism is the strategy that meets the requirements.

![Figure 3.2.2.a Polling](image)

The polling strategy is shown in Figure 3.2.2.a. The client establishes a connection and sends a request. The server immediately sends back a response through the same connection. The connection is closed after the procedure. The polling strategy is not real time because the client does not know when the data update.

Push is the mechanism that can get real time data. There are roughly two approach of push:

1) **Streaming**
2) **Long polling**

![Figure 3.2.2.b Streaming](image)
As shown in Figure 3.2.2.b, streaming is the mechanism in which the web server does not terminate the response to the client after the data has been served. Thus during push, the connection between the server and the client is always open.

Figure 3.2.2.c shows the long polling mechanism. When a subscribe request has been sent from the client, instead of sending an empty response, the server holds the request and waits for the information to be available. Once the information becomes available (or after a timeout), a response is sent to the client.

Either approach of push requires keeping the connection for a long time. This leads to a series of problems on mobile devices.

By holding a TCP connection, an extra keep-alive message must be sent periodically to ensure the connection is still alive. This kind of message sending turns on the radio. Thus increases battery drain. In order to minimize this side effect, there should be an analysis of how the value of timeout will affect battery drain, and find out the optimal timeout value.

If the TCP connection is lost, a mechanism should be provided to reestablish the connection, and make sure the connection is always available.
4 Widget Communication Architecture

This chapter gives an overview of the widget communication architecture. The chapter first introduces some original ideas of the design. Then comes a description of the technologies used in the architecture. At last, the chapter ends with a three-layer architecture design, and explanation what are the layers and what is the relationship between them.

4.1 General Ideas of Widget Communication

4.1.1 Widget Runtime Network

Widget runtime network is the network that widgets communicate on. The network covers both Internet and operator network.

4.1.2 Widget Communication Modes

Every widget is able to act as a peer node in the widget runtime network. Communication between two nodes is a client-server structure. A widget can have a client, a server or both. A client can access any server in the widget runtime network.

![Figure 4.1.2.a Typical Widget Communication](image)

Figure 4.1.2.a shows a typical client-server connection between two widgets. Note that the operation between client and server can be in any form: request-response, subscribe-push, etc.

Communication between widgets can be organized in a chain. As shown in Figure 4.1.2.b, Widget A indirectly communicates with widget C through widget B.

![Figure 4.1.2.b Widgets Communication in Chain](image)

Several clients can communicate with the same server at the same time, and vice versa.
4.1.3 External and Loopback Communication

A client may connect to a server that belongs to the same widget. This kind of communication inside a widget is called loopback communication.

Widget engine should provide a mechanism for handling this kind of situation. Local requests are sent over the loopback, while all the others to the external
network. This mechanism will prevent loopback communication from accessing external network in order to save resources.

4.1.4 Cross Platform Communication

![Diagram of Widget Communication across Platform](image)

In the so-called mobile widget system, widget engine should be available on both mobile platform and server platform. The interface provided by widget engine to widgets should be common so that widgets can be independent from platforms, that is:

1) Widget code is cross platform,
2) Widgets communication is transparent

4.2 Technologies for Widget Communication

4.2.1 Technologies Overview

Widget communication is an inter-machine communication. Communication protocol should be defined and followed by widgets so that widgets can understand each others’ messages. It is always possible to customize such a protocol. However, there are existing web technologies provide good communication interfaces we can use directly. SOAP and REST are two well-known web technologies can be used in widget communication.

4.2.2 RESTful Approach

REST (REpresentational State Transfer) is a style of software architecture first introduced by Roy Fielding in his dissertation [9]. It strictly refers to a set of
network architecture principles which describe how resources are defined and addressed. But the term is often used in looser sense to describe web service technique building on HTTP protocol by REST style as an analogy to other web services technologies such as SOAP. In this case, it is more appropriate to call it as RESTful web service.

Resource is the most important concept in REST. RESTful web service is a resource oriented web service. Architecture of the resource oriented web service is constructed by features of URIs, addressability, statelessness, representations, link and connectedness and uniform interface according to Leonard Richardson and Sam Ruby’s description in their book [10] “RESTful Web Service”.

Comparing to the another popular web service technology SOAP which sets up an additional layer of XML SOAP message for web service communication, RESTful web service takes advantage of existing HTTP features. REST interface is based on HTTP methods, POST, GET, PUT and DELETE. All the operations on resources are performed with these four methods. There are no additional layers over HTTP in REST. So REST is normally considered to be easier to implement and more lightweight. It’s gaining popularity when developing web services.

4.2.3 Restlet, Jersey and JSR-311

There are two well known implementations of the REST framework, Restlet [11] and Jersey [12].

Restlet is a lightweight framework for RESTful web services. It is Java based and composed of elements like connector, router, component, representation, etc. Most of the elements come from the REST dissertation. Restlet extends these notions and successfully puts them into use.

Jersey is another REST framework that implements JSR-311 specification. JSR-311 is Java Specification Request for RESTful Web Services Java. It defines a Java API that is POJO-based, HTTP-centric, format independent, container independent, and includable in Java EE.

Unlike Jersey, Restlet only partly supports JSR-311. Figure 4.2.3 is a picture from Jerome Louvel’s article “Restlet API and JSR-311 API” [13] showing how Restlet API is related to JSR 311.
JSR 311 is good for standardize REST Java API. However, according to the standard, Jersey requires annotations to define RESTful Web Services. That are not supported by MIDP 2.0. Thus it is not possible to port Jersey implementation to Java ME directly.

4.3 Widget Communication Framework

To clarify the architecture, it is divided into three layers.

1) Connectivity layer
2) Resource layer
3) Widget layer

4.3.1 Mobile Platform Three Layer Design

Mobile platform three layer design is shown in Figure 4.3.1.
The connectivity layer of mobile widget communication architecture is called HTTP tunnel gateway. This tunnel acts as a bridge of operation network internal devices and external network for bypassing NATs and firewalls. The two major challenge of Mobile HTTP Tunnel are:

1) There is no existing HTTP container available on mobile.
2) Mobile devices have an addressing problem. That is external client cannot access mobile HTTP server due to NAT structure and firewall.

In the mobile widget communication architecture, the solutions to these two problems are:

1) A light-weight mobile HTTP server
2) A tunnel based networking architecture that keeps connections from mobile devices to gateway server so that external client can access mobile devices though those connections.

The resource layer of the architecture is called Mobile REST Server. It is independently designed, but takes reference to Restlet components. For the reason that Java Annotation is not available on MIDP, Jersey framework which largely depends on annotation has not been introduced too much into the implementation.

Mobile widget layer includes mobile widget container and widgets. Widget container is the logic part of widget engine. It manages life cycles of all widgets and also in charge of widgets UI display and user interaction. Widget container
together with the HTTP and the resource layer implementations constructs the whole architecture of widget engine. Widgets are platform-independent applications building on top of widget engine. In order to be platform independent, there is a layer of widget container common interface that abstract a common interface between server engine and mobile engine.

4.3.2 **Server Three Layer Design**

![Figure 4.3.2 Server Platform Three Layers Design](image)

Figure 4.3.2 shows the three layers architecture implementation on the server.

Unlike mobile connectivity layer, the two problems exist on mobile HTTP tunnel does not exist on server for that:

1) There are already mature HTTP Servlet Containers existing on server.
2) Widget engines on server are on “external” network that do not have NAT firewalls problem.

The resource layer on server almost has the same structure as that on mobile.

The functionality of server widget container and mobile widget container are also almost the same. The common interface building on these two containers opens the same APIs to widgets makes widget development uniformed. Almost all the problems due to platform difference are handled by engines.
### 4.4 User Group

Before going to architecture detail, it is important to make clear what the user groups are. There are three groups of people, engine developers, widget developers and widget users. Engine developers are those who design and implement widget engine and provide framework and APIs for developing widgets. Widget developers use the frameworks and APIs to develop widgets and publish them so that they are accessible to widget users. Widget users choose widgets from the list. These widgets will be integrated with widget engine and run on users’ devices as an application.

The connectivity layer and the resource layer in widget communication architecture are implemented by the engine developers. They provide a RESTful service interface to the widget layer. The widget layer is mainly implemented by widget developers. All the three layers implementations integrate together with non-communication implementations are compiled to executable widget engine applications.
5 Connectivity Layer Architecture

The connectivity layer is the lowest layer of the widget communication architecture. The main functionalities of the connectivity layer are:

1) Listening to the incoming HTTP package
2) Creating HTTP request and response according to the incoming package
3) Creating request thread, give the thread handle to resource layer
4) Getting handle back from resource layer, sending network package according to HTTP response

This chapter first gives an overview about general structures of connectivity layer on mobile and server separately. Then it will start detail discussions into some key modules of the structures.

5.1 Architecture Overview

5.1.1 Server Connectivity Layer

The widget engine is designed to run on both mobile device and server. On server, there are much more possibilities for scalable HTTP servers than on mobile devices because of fast processors, large memories, etc. HTTP server running on server is quite a mature technology. There are many solutions available. Apache Tomcat Servlet container is the one used in the server connectivity layer architecture. Any Servlet container can be a substitution for this purpose.

As is shown in Figure 5.1.1, Apache Tomcat container is deployed on the bottom layer of the architecture. The container listens to incoming package and handles them through the implementation of Servlet.
The HTTP Servlet adapter is built on top of Tomcat container. The adapter is actually a Servlet that grabs the handle of the request, transfers it to the formats that the upper layer can receive and forwards them to REST engine.

Response is handled the other way around. The REST engine generates response. HTTP Servlet adapter gets the response and transfers it to the format Tomcat can handle, and then forwards the handle to the Tomcat container. After that, the Tomcat container sends the response.

The architecture is compact. Implementation of this architecture is mainly the Servlet adapter.

5.1.2 Mobile Connectivity Layer
Mobile connectivity layer has the same functionality but different structure as server connectivity layer. On server connectivity layer, Apache Tomcat is the one which does most of the work. HTTP Servlet adapter is just a bridge that connects Tomcat and upper layer REST engine. But on mobile, no existing HTTP server is available. Thus a design of an HTTP Server on mobile is essential.

As is shown on Figure 5.1.2, mobile connectivity layer contains two major modules: Tunnel Gateway and HTTP Server.

As has been discussed in section 3.1.3, there exist problems that NATs and firewalls block any outside request from accessing mobile devices. If no request can access mobile devices from external network, HTTP server on mobile will be meaningless.

Tunnel Gateway is a gateway deployed in external network. The gateway bridges mobile devices inside NATs and external networks in order to walk around NATs and firewalls. Tunnel Gateway is a mechanism provided for low level request accessing that ensures that HTTP request and response can go in and out of mobile devices.
On top of the tunnel gateway is a mobile HTTP server. The HTTP server is a lightweight server due to performance limitation of the mobile device. The lightweight HTTP server should be able to handle essential functionalities of the connectivity layer as well as the problem of loopback and external communication.

5.2 Server Servlet Adapter

Server Servlet adapter bridges the gap between Apache Tomcat Servlet container and the REST engine.

5.2.1 Working Mechanism

Server Servlet adapter is working as a Servlet on Tomcat container. It extends the class HttpServlet. Several methods from HttpServlet are implemented in order to:

1) Initialize REST engine
2) Transform HTTP request/response to REST request/response
3) Give/get the handle to/from the REST engine

The REST engine initialization is implemented in the method of init. This method is called once by the Servlet container when the Servlet is placed into the service.

Four methods of request handling: doGet, doPost, doPut and doDelete are implemented. There are the four basic methods supported by resource layer. The implementations of these four methods are giving a handle to a method called handle.

The handle method transforms between HTTP request/response and REST request/response. Then give handle further to the REST engine.

5.2.2 Request Response Transformation

There are two directions of transformation, HttpServletRequest, HttpServletResponse to RestRequest, RestResponse and the other way around.

In REST data design, RestRequest and RestResponse are both extended from RestMessage. RestMessage has the common data fields of attributes and entity.

Attributes are all the key-value pair information in RestMessage. They can be parameters, attributes and HTTP headers in HttpServletRequest.

The entity is the payload of the message. Entity in RestMessage is in type of RestRepresentation. Representation is an important idea in REST, the definition of RestRepresentation will be discussed in later chapter.
The transformations of attributes are simply reading key-value pair information from one message format and putting it to another. The data structure of different message format implementations can be different. But the keys and the values can always be extracted as strings and added to another format.

The transformations of entities are different from that of attributes. The entities types of HTTP Servlet message is arrays of bytes, while the entities of REST message is `RestRepresentation` which is a java data structure. Transformation between those two requires a parsing and generating mechanism. This mechanism parses array of bytes to `RestRepresentation` data structure and generates array of bytes from `RestRepresentation` data structure.

There are more data fields to be transferred other than attributes and entity. The URI and the method in the REST request and status in REST response should be transferred accordingly.

5.3 Tunnel Gateway

In order to bypass NATs and firewalls, tunnel gateway is deployed on external network. Whenever mobile devices start up widget engines, register requests are sent automatically to tunnel gateway. Tunnel gateway will keep the connections and reuse them for requests targeting mobile devices. These connections are called tunnels.

Tunnels should be kept alive all the time as long as widget engine on mobile devices are alive. Therefore, there are also questions of how to keep the connection alive.

5.3.1 Tunnel Gateway Architecture

Figure 5.3.1 shows the network diagram of the tunnel gateway. Firewall blocks any outside request from accessing devices inside NAT.
As can be seen on the graph, tunnel acts like a bridge bridging mobile devices and external network. The tunnel is kept alive, and stored in an address-tunnel map. Requests from external network first go to the tunnel gateway. The tunnel gateway looks up the map, finds the corresponding tunnel according to request address and then forwards the request to the target mobile device through the tunnel.

When responses from mobile devices are available, mobile devices send them to the gateway through the same tunnel. Tunnel gateway therefore sends the response to the requesting devices through the same connection they used for requesting.

There are several challenges in the architecture:

1) How to address mobile devices?
2) How to keep the tunnel connection alive?
3) How is the response routed to the mobile device?

5.3.2 Addressing Widgets
Addresses of mobile devices inside operator network are local. They are only used within operator network and are not accessible from outside. A universal unique address must be given to each mobile device to identify its location.

In the tunnel gateway architecture, the tunnel gateway is a delegator that delegates all devices inside operator network. Any request that goes into operator network must go through the gateway. Gateway keeps address information of the devices and is the only one in the external network that can access these devices.
Inside the gateway, a map is kept that maps mobile devices to the connections. A unique address can be allocated to every mobile device by the gateway. Any external request attempting to go to a certain mobile device goes to the tunnel gateway with the unique address. And the gateway finds the connection and guides the request to the correct device.

Therefore, a mechanism based on Uniform Resource Identifier (URI) is provided to address mobile devices. The mobile address is in the format of: “gateway address/mobile identifier”. For instance, if the gateway is assigned an address 192.36.157.104 and port 8080, and the mobile identifier of a certain device is “yesheng”, identifying that it is yesheng’s mobile. Then this mobile will have a universal unique address: “http://192.36.157.104:8080/yesheng”. The address of the gateway can be replaced by domain name.

The gateway administrator can configure gateway address in a configuration file to inform the gateway of its own address. What gateway does is it reads the URI segment after the gateway address which is the mobile identifier. Then the gateway can looks up the map with the identifier and finds the connection.

### 5.3.3 Comet Push and Connection Keep-alive

Comet is a technology used for web application that allows web server push data to clients. The technology is similar to Ajax, except that Ajax data is fetched by classical request-response pair mechanism, while Comet data is pushed to client.

Comet has two approaches of push: long polling and HTTP stream. The approach of long polling does not really keep connections alive, as has been discussed in section 3.2.2, while the approach of HTTP stream does. Thus it is a practical option for implementing tunnel gateway.

Grizzly Comet is a framework built in the Glassfish application server that enables scalable IO operations and provides push mechanism.

![Gateway Architecture](image)
Rather than implementing IO framework by ourselves, we use Grizzly in the architecture design. Figure 5.3.3 shows a clear architecture of tunnel gateway using Grizzly Comet. The connection map is maintained inside gateway application and the connections are kept alive all the time until mobile devices are deregistered from the gateway.

Above discussions depend on an assumption that every tunnel connection is kept alive throughout the life circle of the widget engine. How to keep the connections alive is what will be discussed in this section.

Firewalls in the way of a TCP connection will close the connection, if there is no traffic goes through the connection in a given time. In order to keep the connection alive, keep-alive messages should be generated periodically to inform both sides that the TCP connection is still alive.

There is a question of how regularly the keep-alive messages should be sent. If the messages are sent too often, it causes more traffic, turns the radio on frequently and drains the battery. If they are sent too rare, the period might be longer than the period of the firewall timeout, and the connection will be closed. A good implementation will find out a timeout value that is close to the optimized value.

A push mechanism should consider the timeout problem. In order to have push solution, we need to perform connection maintenance using keep-alive messages.

5.3.4 Message Flow

Message flow indicates how messages are sent between mobile devices, tunnel gateway and external devices.

The purpose of the tunnel gateway is to make HTTP server on mobile available to the external network. The prospective the message flow is as simple as what is shown on the following message chart.

![Message Flow Diagram](image)
As Figure 5.4.3.a shows, client sends a request to the gateway. The gateway looks up in its map, gets the connection to the mobile device and forwards request to the device. After mobile server handles the request, the response is sent back to external client through the gateway.

In reality, however, the message flow between the tunnel gateway and the mobile devices is not exactly as simple as it is shown on the above message chart.

Review the streaming push mechanism on Figure 3.2.2.b, request is sent only once from client to server for registration. The connection is kept alive and the data from the server is always pushed to the client through the response output stream.

We can imagine that in the tunnel gateway architecture, a mobile device is a push client and the tunnel gateway is a push server. The mobile device will send a request to the gateway for registration when the widget engine starts up. The tunnel gateway will keep the connection available. When a request is sent from the external client to the tunnel gateway, it is the object to be pushed. The tunnel gateway will look up the connection map and will get the right connection for pushing according to the destination address of the request. Then the request is pushed through the response output stream.

After the request has been handled, a response should be returned to the external client through the gateway. Since in the streaming push mechanism, request for registration is the only message sent from push client to push server. There is no way for sending response from push client to push server in this mechanism. For this reason, another connection is opened for sending the response. The response is put inside a new request and marked as response. Tunnel gateway parses the request, extracts the response and immediately returns a 200 OK response back to mobile to end the connection.

The last procedure of the request-response chain is to forward the response to the external client who initialized the request. In order to do this, the same connection which is used to send request should be used to send the response. Unfortunately, the old connection is not available in a new request session. An extra step should be taken to store the connection into another map in advance so that the connection can be fetched later. A unique request ID should be generated to map a connection. The unique request ID is sent along with the request to the mobile server and back along with response to the gateway. Gateway gets the response, parses it and gets a unique ID. Gateway can get connection according to the unique request ID and send response through the connection.
5.4 Mobile HTTP Server

Mobile HTTP server is a lightweight HTTP server running on the mobile phone. It is built on top of the tunnel gateway and provides connectivity layer interface to the REST framework implementation.

5.4.1 Loopback Connection

As has been described in section 4.1.3, when a widget client sends a request, destination of the request can be inside the same device. In this case, a loopback mechanism at connection level is provided to avoid sending the request to the network. The loopback connection is responsible for establishing a connection between client and the HTTP Server on the same device.

Like a socket, the loopback connection mechanism contains loopback server connection and conventional loopback connection. The interfaces of these two connections are inherited from \texttt{javax.microedition.io.ServerSocketConnection} and \texttt{javax.microedition.io.SocketConnection}. Loopback Server connection is opened in a thread waiting for clients to connect. Input stream and output stream of these two connection ends are connected together using piped stream.

5.4.2 Incoming Messages

Since mechanism of loopback connection is provided, there are two kinds of requests that might go into the HTTP server, loopback and external. The HTTP server should listen on both kinds of connections and handles incoming message from them.
In order to do this, tunnel proxy and loopback proxy are created for listening to both kinds of connections. These two proxies run in separate threads so that they do not miss any packages. Requests received by both proxies are put into a synchronized queue. The queue is only accessible by one proxy at a time and it is consumed by the HTTP server. For every request, the HTTP server creates a separate thread. The thread is alive until a response to the same request is generated and sent. Thread handling consumes system resources, but it is necessary for the server to be able to handle embedded requests.

![Figure 5.4.2 Mechanism of Message Listening](image)

5.4.3 Message Parsing
One major task of HTTP server is to parse HTTP request and generate HTTP responses. Message parser and message generator and their subclasses, request/response parser/generator is created for this purpose. These are interfaces open towards lower level connections that translate between array bytes and HTTP server data structure. Request parser in the proxies shown in above figure parses incoming bytes and puts them into the request queue. HTTP server gets these passed messages from the queue and reads data contents from the parser as needed.

5.4.4 Protocol Definition
Not all HTTP protocol features defined in RFC 2616 are necessary to be implemented in the light-weight mobile HTTP server. It depends on the need of upper level implementation. In the connectivity layer architecture, a subset of the protocol features is defined considering the need of the resource layer and mobile devices.
6 Resource Layer Architecture

The resource layer is in the middle of the widget communication architecture. It is the major functional layer of the widget communication. The main functionalities of the resource layer are:

1) Getting handle of request thread
2) Finding a resource according to the URI in the REST request
3) Giving handle to resource if it is found
4) Return response

In this chapter, resource layer architecture is discussed. Several new concepts in this architecture are defined and described in the first part. Then an architecture overview is given for a global picture of the resource layer architecture. Following the overview, more details like class hierarchy, framework interface and resource reaching algorithm are described.

6.1 Components

Before looking into the architecture, there are some concepts to introduce.

1) REST engine: REST engine is the engine that runs all REST functionalities. It is a resource layer implementation that enables REST in widget communication architecture. REST engine constructs and maintains REST widgets and provides mechanism for reaching resources.

2) REST Interface: REST Interface is the hierarchical structure of the widget REST resources. It defines the URI pattern of the widget network interface.

3) Component: Component is an abstract container unit that contains REST elements, server, router, etc. A component itself does nothing particular other than provide an abstract concept of a container. This abstract concept is useful to separate REST resource hierarchy into several levels. It allows to reuse components

4) Connector: A connector is a mechanism that connects components so that they can communicate with each other. A connector can be a server that receives communication request or a client that initialize a communication request.

5) Server: A server connector is a connector that acts like an interface of a component.

6) Client: A client connector is a connector that connects to a server connector.

7) Router: Router routes requests to the attached resources, servers or routers.
8) Resource: Resource is the most fundamental concept in REST. More specifically, in REST architecture, resources are terminal objects every request is going to be routed to.

9) Representation: Representation is current or intended state of a resource. In resource layer architecture, it has the form of byte array. Metadata are provided to describe representation types.

### 6.2 Resource Layer Architecture Overview

Figure 6.2.a shows how typical REST interface can be defined.

![Basic Structure of a typical REST interface](image)

Figure 6.2.a Basic Structure of a typical REST interface

In this graph, server is an interface of the component opening for component services accessing. The server attaches a router and the router attaches two resources. Both server and router implement an interface allowing them to look up elements attached to them.

![Nesting in the REST Interface Structure](image)

Figure 6.2.b Nesting in the REST Interface Structure
A more advanced structure is shown in Figure 6.2.b. In this structure, components are embedded inside other components. We can also find in this graph several possibilities of server and router attachments, a router can attach a server and a server can attach a resource.

There are several rules for constructing resource hierarchy in the architecture:

1) A component has at least one or more servers.
2) A server can attach one and only one REST object, the REST object can be a router, a server or a resource.
3) A router can attach one or more REST objects, these REST objects can be server and resource.
4) A resource is a leaf node that cannot attach to other REST objects.
5) A component can be embedded in another component.
6) The only way a component can communicate with other components is through a server.

### 6.3 Class Diagram

In the architecture design, all REST elements are extended from a parent class called `RestObject`. `RestObject` has an abstract method called `handle` that handles REST request and response pair. Every child class of `RestObject` must implement the method.

![Resource Layer Class Hierarchy](image)

As it is shown on Figure 6.3.a, `Connector`, `Router` and `Resource` are REST elements directly inherited from `RestObject`. `Server` and `Client` are two connectors inherited from connector classes. Both `Server` and `Router` implement interface `Lookupper` that enables them looking up their attachments for terminal resources.
Figure 6.3.b RestRpresentation

Figure 6.3.b shows class hierarchy of the REST representation. There are several representations defined in the architecture. REST representation is extendable and can be customized by the widget developers.

A metadata should be associated with a REST representation to identify what media type the representation actually represents. A map of metadata and representation classes should be provided in the system configuration file to inform the engine in advance of the mapping. This mapping is useful to identify the type of REST representation so that REST engine can generate a correct REST representation from an incoming REST message according to the media type field of the message.

6.4 RESTful Resource API

REST framework opens an API to widget layer for widget developers to define REST interface and implement resources. The goal of REST framework API design is to make the interface as simple as possible and to hide the internal implementation of REST as much as possible.

In the architecture design, two abstract classes, RestWidget and Resource, are provided as interface for this purpose.

RestWidget is an abstract class that contains a REST component. We can consider it as a component with extra functionalities. The abstract class has an abstract method called initRest. This method initializes REST interface of a widget the first time REST engine deploys the widget by creating REST elements and combining them together.

The following example shows a weather widget extending RestWidget and the REST interface of the weather resources.

```java
public class CityWeather extends RestWidget {
    public boolean enableRest() {
        return true;
    }
}
```
public void initRest() {
    Server server = new Server(getWidgetComponent());
    Router router = new Router(getWidgetComponent());

    router.attachResource("Stockholm", WeaShhm.class);
    router.attachResource("Uppsala", WeaUpps.class);
    router.attachResource("Shanghai", WeaShai.class);

    server.attachRouter(router);

    this.addServer(server);
}

The implemented method enableRest is a method to define if the REST interface is enabled in this widget or not. Implementation of this method returns true or false. In the implementation of initRest, a router is created to attach three weather resources. The router is attached to a server and the server is attached to the widget’s component.

Resources are main logic of handling requests. In resource layer architecture, four HTTP methods are supported, GET, POST, PUT and DELETE. Abstract methods are generally the two types, allowMethod and doMethod. AllowMethod defines if a method is allowed. Return value of this method is simply true or false. doMethod is the implementation that actually handles request and returns response.

public class WeaShhm extends Resource {

    public boolean allowDelete() {
        return false;
    }

    public boolean allowGet() {
        return true;
    }

    public boolean allowPost() {
        return false;
    }

    public boolean allowPut() {
        return false;
    }

    public void doDelete(RestRequest request, RestResponse response) {
    }

    public void doGet(RestRequest request, RestResponse response) {
        RestHtmlRepresentation rep = (RestHtmlRepresentation)
                                                             RestRepresentationFactory.getInstance().createRepresentation("text/html");
        rep.setHTML("Stockholm Weather","<p><h2>Good</h2></p>);
        response.setEntity(rep);
    }
}
Above weather resource enables the GET method. The resource implements `doGet` and returns a HTML representation telling the weather in Stockholm.

### 6.5 URIs as Resource Addresses

Once resource hierarchical structure is built up, resources are accessible to incoming requests. Each resource has an URI address that specifies the route to the resource. The resource URI address is mapped from corresponding resource accessing route in resource hierarchical structure. The graph below is the resource hierarchical structure of city weather widget:

![Resource Hierarchical Structure of City Weather Widget](image)

The corresponding URI of the three resources are `ROOTPATH/Stockholm`, `ROOTPATH/Uppsala` and `ROOTPATH/Shanghai`.

The segments of the URI path can be parameterized. For example, a router can attach to a weather resource service which is not set to a certain city.

```
routing.attach("{cities}", WeaCities.class);
```

The URI address of the resource is thus `ROOTPATH/{cities}`. The path parameter is wrapped by `{ }` pair. The URI path matches any URI segment value. The value of the parameter can be, for example, Stockholm, Uppsala, Shanghai or any other cities. All the requests that match the URI will be forwarded to the `WeaCities` resource.
6.6 Request Routing

In implementation of Jersey, a global map is kept mapping resources to their URI addresses. URI match algorithm can thus be performed on the map by performing only string operations.

Jersey has a notion of sub resource. Resources and sub resources are defined in a single file in Jersey. It is not difficult for Jersey engine to generate a global map from the file. But in widget communication resource layer architecture, resources and structure of resources are defined in different places. The notion of component separates the resource structure into several reusable parts. These components are loosely linked with each other using connectors. The structure becomes dynamic and it is not possible to generate a global map from that.

REST interface is a combination of REST elements that are kept in memory. The structure is a tree that has nodes and branches. The terminal nodes of the tree are resources. The resource reaching algorithm is performed on the tree structure. The algorithm is basically a searching algorithm. Every time the program reaches a node, it decides which way to go according to a URI segment. The searching will not end until terminal resource is found or until the whole tree is searched.

Backtracking is performed when the result of the sub tree does not match a corresponding URI segment. Then the program will go to the previous choice point and choose another route. If no resource is found when the iteration over the tree is done, a 404 response is returned.

Different strategies can be implemented for resource reaching algorithm according to different demands. Strategies can be first match, best match, etc. Different strategies may return different resources.
7 Widget Layer Architecture

The widget layer is the top layer in widget communication three-layer architecture. The connectivity layer and the resource layer are implemented in the scope of the widget communication framework. The implementation is invisible for the widget developers. While widget layer is composed by the terminal applications called widgets, widgets are developed by widget developers using the framework and API provided by lower layers.

This chapter first overviews the architecture, then goes to the design of the widget layer architecture including the user interface and the representations. In the end of this chapter, MVC pattern will be introduced and discussed how the pattern can be used in widget layer architecture.

7.1 Widget Layer Architecture Overview

![Widget Architecture](image)

The contents of widget layer are nothing more than widgets. A widget has three components, the logic, the GUI and the REST server.

The logic is the main function of a widget. Widgets are free to do any logic control in this part, for example, doing calculation, initializing variable, fetching REST resources, etc. Code of the logic part is executed during the initialization of a widget. Thus the logic control is only available in a short term. A long term logic control of a widget can be realized by creating a new long-lived thread.

The GUI displays the widget data and handles widget-user interaction. The widget GUI uses a HTML like technology called Marco HTML for both displaying and handling user actions. The mechanism for widget GUI is like a browser that parses and renders Marco HTML files. Detail discussion of this mechanism will be given in a later section of this chapter.

The REST server enables widgets to be light-weight servers. Widgets can provide information they collected as REST services through REST server. With this mechanism, widgets can exchange data and communicate with each other. The REST server is based on the REST framework provided by the resource layer and HTTP support provided by the connectivity layer.
All the three components of a widget, the logic, the GUI and the REST server, are optional. A widget works with any of the three components. The logic part is the main logic of a widget. However, it has no direct control over the GUI and the REST server. The GUI and the REST server are independent of logic and can run independently without implementation of the logic. The GUI is a compulsory part in most traditional definitions of widgets. But in our widget engine, the GUI is not a necessary part of a widget. A widget can be an application running at background as servers or any logic implementations without the user interface. Still, a widget is not compulsory to provide services. The REST server is also optional.

A widget is a combination of the three components. Which part of the three components will be implemented is up to the widget developers. A good design is to separate different functionalities into different widgets so that each widget can focus on only one specific task. But it is still possible to integrate UI, logic and server services into one widget. Sometimes it is good to do so because it saves system resources and is comparatively easy to implement.

The logic part implementation of a widget is in a callback method implemented in a widget class. Implementation of a REST server is implemented through REST framework by using REST framework interface. Definition of GUI is a declaration of URI address to a Marco HTML resource. Both REST server and GUI of a widget should be declared before a widget starts running. In our widget engine architecture, the declaration is done in a configuration file.

The configuration file is in format of JSON.

```json
{
    widget:[
        {
            name:"WeatherDisplay",
            class:"com.ericsson.research.marco.widget.example.WeatherDisplay",
            rest-class:"com.ericsson.research.marco.widget.example.RestWeatherDisplay"
        },
        {
            name:"Weather",
            class:"com.ericsson.research.marco.widget.example.Weather",
            rest-class:"com.ericsson.research.marco.widget.example.RestWeatherReference",
            icon-url:"http://localhost:8090/WeatherDisplay/icon",
            index-url:"http://localhost:8090/WeatherDisplay/index"
        }
    ]
}
```

In the configuration file, the widget developer declares two widgets. The first REST widget is a pure REST server widget that provides resource of widget user interface in the format of Marco HTML. The second widget is a weather widget that has both the GUI and the REST server components. Notice that the GUI of this widget is URI address to the resources defined by the first widget. The weather widget also provides a REST service to the reference of this widget.
7.2 Fetching UI Declaration through REST

The widget GUI is declared by Marco HTML. How widget engine renders Marco HTML, displays it on devices’ screen and how user actions are processed are out of the scope of the widget communication. However, the mechanism of defining and accessing Marco HTML resources is tightly bound to widget communication framework.

The architecture of the widget GUI is based on the REST framework. Like a conventional web server mechanism, there are a client and a server. The server is a REST server that enables Marco HTML resources through the REST interface. The client is like a browser that requires Marco HTML resources using the REST client and renders them on the screen.

The benefit of using client-server framework is that widget developers have a choice to define widget GUI either locally or externally. Because REST clients can either access local REST server resources or external ones. It is more reasonable in our case to put the declaration of GUI in memory of the same devices with rest of the widget definitions. External GUI resources have extra advantages in the content of our architecture. External GUI resources are possible to reuse and they can be recombined easily into new GUI resources, which opens new possibilities for the GUI design.

7.3 Marco HTML Representation

Marco HTML is a HTML-like language. HTML is a mature web technology widely used for designing web pages. HTML is chosen as the language for the widget GUI developing by most widget engines, because it is an easy language to use for defining widget user interfaces.

Marco HTML is not exactly a static markup language like HTML. Marco HTML language can dynamically display contents generated at runtime. This is achieved by a mechanism of the content replacement.

The Marco HTML file format is almost the same as HTML except that there is the tag for string variable. A string variable is defined to start with ‘[’ and end with ‘]’. The contents between ‘[’ and ‘]’ is considered to be string variable and is replaced by the string value at runtime.

For example, Marco HTML can dynamically display pictures by defining a string variable of picture address.

```html
<HTML>
<HEAD>
...
</HEAD>
<BODY>
...
<IMG arc = [PicAddr] />
```
PicAddr is replaced with a value at runtime so that the page can display different pictures according to the different runtime scenarios.

The mapping of values to variables is maintained inside resource implementation.

When creating a Marco HTML representation, the widget developers must both create a Marco HTML file and a variable map. Marco HTML representation receives two parameters, the address of the Marco HTML file and the object of the map. The representation will replace tagged the variables with values according to the map and then return static HTML file as a response.

Marco HTML fulfills basic demands for dynamic GUI displaying. More powerful GUI display mechanism can be designed by extending Marco HTML representation or HTML representation directly.

### 7.4 Widget MVC Pattern

In the widget engine framework, a widget can be a server that provides data, a logic handler that processes data or a UI that displays data.

The Widget MVC pattern separates widgets into three kinds, model widgets, view widgets and control widgets. Instead of performing several roles by one widget, widgets in the MVC design pattern perform only one role.

The following graph shows the overall principle of the widget MVC design pattern.

![Figure 7.5 Widget MVC Design Pattern](image)

MVC pattern separates widgets into three layers of widget implementations. Widgets in model layer are generally data objects that hold data. Widgets in
control layer contain business logic, data adaption or control logic. Widgets in the view layer have a single task of displaying widgets on screen. MVC decomposes big applications into small widgets. REST is used as communication media between the widgets and links them together.

The concept behind widget MVC is that widgets may depend on each other. The dependency between widgets can be “one to one” or “one to many”. For example, a control widget might get data from several model widgets and a view widget might be controlled by several control widgets. Through this way, a widget can get service from one or many widgets. Every widget is an “expert” in one task either in model, view or control. And one widget can be substituted for another.

Widget MVC pattern increases flexibility and reusability of widget code. MVC widgets are loosely coupled to each other through REST. So it’s easy for the widget developer to replace and recompose widgets. Furthermore since REST is a communication mechanism over internet, widgets under MVC can communicate with each other across platforms. That means for example, the model and the control widgets can be deployed on a server and view widgets on a mobile can communicate with these widgets as if they were all on the same device.
8 Conclusions and Discussion

In this chapter, the conclusion of the thesis is first drawn. Limitations of the widget communication architecture are then stated, followed by the future work on improving the architecture.

8.1 Conclusion

The thesis is aiming at building a widget engine so that the widgets on the engine are capable of:

1) Communicate with each other;
2) Access web services on the internet;
3) Access other internet contents.

After designing and prototyping the widget communication architecture, we can say the purpose is fulfilled. The widget engine prototypes with the design of widget communication architecture described in the thesis are deployed on both server and mobile devices. The performance of widget engine communication is satisfying despite the limitations of the platforms and the technologies used. So we can draw the conclusion that the architecture of the widget communication works both theoretically and practically.

8.2 Architecture Limitation

Connectivity layer in the architecture is based on the tunnel gateway that bridges mobile devices to the external network. Tunnel gateway works fine in the widget communication prototype. However, it is only a prototype design that puts little consideration over reliability, scalability, security and manageability. For a real gateway that bridges millions of connections, this design is apparently far from being prefect.

Another limitation of the current architecture is that there are no authorization and security mechanism in the design. Widget communications over internet are all transferred as raw data without encryption. This is not safe, as everyone on internet can grab the packages and read the content directly.

8.3 Future work

It is necessary to find solutions to both of the above limitations. One possible future work can be on investigating a reliable, scalable, secure and manageable architecture to increase or to replace the work of tunnel gateway to bypass the problem of NAT traversal. A possible solution can be IMS (IP Multimedia Subsystem) that delivers IP to mobile devices and enables addressability of mobile devices. The second possible future work is adding security mechanism to the widget communication framework so that widgets can communicate in a secure way.
Above future works are based on improving current widget communication architecture. It is also interesting to investigate potential usage within the current architecture. Widget communication architecture opens possibilities for every widget to actively provide useful information to other widgets. Widgets in the community network can thus increase value of their community by providing these services. This is rather a Mobile 2.0 feature. The future works can focus on how Mobile 2.0 ideas can be applied within the widget communication architecture.
9 Reference


10 Acknowledgements

It has been a pleasure to do the thesis work in Ericsson Research for six months. I would like to thank Leonid Mokrushin and Göran Eriksson for enrolling me to the work. Special thanks to my supervisor Leonid for giving me the fabulous thesis topic and providing me a lot of help.

I wish to express my sincere thanks to my thesis reviewer Justin Pearson from Uppsala University who reviews my thesis.

I would also like to give my wholehearted thanks to my thesis work partner Lars Vising. We work along very well for six months in the same office. Without Lars I would not be able to finish the thesis alone. Also thanks to my colleague Vladimir Katardjieva and Jose Lucea who also give me a lot of help on this thesis work.