Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Automated Proxy Injection for Redistributed Applications

Mihir Gore

Course of Study: INFOTECH

Examiner: Prof. Dr. Frank Leymann

Supervisor: M.Sc. Karoline Saatkamp

Commenced: March 1, 2017

Completed: August 31, 2017

CR-Classification: C.2.4, D.2.11, H.3.4

Abstract

Many businesses and enterprises use cloud platform to host their applications on the cloud. In the current era of cloud, there are various possibilities to host such applications on different platforms, for example, it can be hosted on-premise private or public cloud hosts such as Google App Engine, Amazon AWS, Windows Azure. In other words, it can be considered that they are distributed over several hosts either on local or on cloud hosts. It is highly possible that multiple applications communicate with each other to serve the user's need. Due to change in business requirements, performance or cost optimization purposes, these applications may be redistributed to different hosts and also while redistribution, the underlying communication protocol may be changed. In such cases, it is important to retain the same communication between these applications. Proxies are one of the design patterns used to route the communication between applications. This thesis proposes an approach to automatically generate and inject such proxies containing translators and adapters between such applications to handle redistribution of applications and communication protocol change. A goal is to develop a tool capable of generating and injecting these proxies.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

In recent years, cloud computing has become an important term in IT industry [Ley09]. Cloud computing benefits and it's several characteristics such as self-service provisioning, elasticity, and pay-per-use model attracts many people and developers [WBB+13] to build highly sophisticated and scalable applications. In today's world, it is apparent that most of the businesses have their own applications, be it an enterprise application or web application to achieve their objectives much faster and reach out to their customers. Such businesses and companies leverage the benefits of cloud computing paradigm.

These applications are distributed in nature and are hosted on different hosts including numerous on-premise, private, or public cloud platforms such as Google App Engine and Amazon AWS. Typically, these applications are made up of many interacting software components and can communicate with each other. Therefore, orchestration and wiring of such applications is a major issue [ZBL17]. Due to varying nature of business requirements, applications might have to be redistributed or migrated from one host cloud provider to another. After such redistribution, it is necessary that the overall functionality of applications remains the same and they deliver the same results possibly with improved performance. The Scope of this thesis revolves around handling this redistribution of applications without affecting the desired communication between these applications.

The phrase "enterprise application" is used to describe an application or collection of software components that particular organization uses to solve enterprise problems. Therefore it is a big business application and these applications are typically designed to integrate with other enterprise applications [Net17]. These applications may be deployed across a variety of networks such as the Internet, intranet, and corporate networks. Various design patterns described in [HW03] are widely adopted in enterprise integration. Out of these patterns, this thesis uses Proxy Pattern, Translator Pattern, and Adapter Pattern to handle the re-distribution of applications.

In general, software components or applications that are communicating with each other may be relocated or redistributed from one host to the other for example an application may be migrated from one cloud host to another cloud host. There is also a possibility that underlying protocol for communication is changed due to change of hosts. Reasons for such change being the change in business requirements, cost reduction, optimization in performance, new cloud platform offerings, hardware or operating system upgrades etc. In such scenarios, the application needs to be modified in order to support such changes and maintain the orchestration. Modifying the application is a manual process that requires rebuilding the application by making changes to its source code to facilitate communication based on new protocol and to update communication endpoints to retain same communication as before. For complex applications, the manual process may result in a cumbersome and time-consuming task. Hence there exist a need for automation process that handles this redistribution. This thesis attempts to provide an automation approach in such scenarios by injecting proxy components containing address routing component, translators, and adapters by requiring as minimum information as possible and thus solving the problem of maintaining the same communication between redistributed applications.

The rest of this thesis is structured as follows:

Chapter 2-Fundamentals: It includes the basic and important concepts that are necessary for understanding the terms related to the thesis.

Chapter 3-Related Work: It includes literature work related to automatic proxy generation, protocol translation mechanism, and adapters.

Chapter 4-Proxy Generation and Injection Approach: It describes the suggested approach of proxy generation and injection for application redistribution scenarios.

Chapter 5-Implementation: It states technical details regarding the prototype built in this thesis work. In general, it describes the setup of a working environment, built web application details and design of Proxy Generator (PG) tool.

Chapter 6-Validation: It states the validation results of implemented prototype in terms of response time measured as per scenarios described in Chapter 5.

Chapter 7-Conclusion and Future Work: It includes the summary of thesis work and discusses the possible extensions and improvements for this thesis.

2 Fundamentals

This chapter includes all the important concepts that are necessary to understand the content of this thesis. The concepts of application redistribution and design pattern are discussed in this chapter. This chapter also discusses application layer protocols HTTP and MQTT for supporting the prototypical implementation involved in this thesis.

2.1 Communication Paradigm

There are many definitions of distributed system. Tanenbaum [TS06], defines the distributed system as a collection of independent computers that appear to the users of the system as a single computer. These computers or nodes work independent of each other and are connected to each other by a network. An application running on such computers is treated as a distributed application. These applications can communicate with each other over a network by adhering to communication paradigms such as

- Remote Procedure Calls (RPC): RPC is a communication paradigm that is used by one program to invoke the procedure or service from another program located in another computer on a network.It follows client-server architecture pattern wherein client machine invokes the procedure which is located at server machine. Examples are CORBA, Java RMI, DCOM (Object based), DCE, Sun RPC (Data-based)
- Message Oriented Communication: In this paradigm, messages, which corresponds to events are used for communication between processes. Tanenbaum [TS06], classifies the message oriented communication as synchronous or asynchronous communication and transient or persistent communication. Examples are Socket Programming, Message-passing interface (MPI), Message-oriented middleware (MOM), Publish-subscribe communication.
- Stream Oriented Communication: RPC and message-oriented communication are based on the exchange of discrete messages but in stream oriented communication continuous stream of media is exchanged e.g. audio, video, or animated images.

- Multicast Communication: The basic idea of this communication paradigm is to support sending of data from one sender to multiple receivers. The example is Peer-to-Peer System.

The approach proposed in Chapter 4 can be superficially applied to RPC, Message Oriented Communication, and Multicast Communication paradigms. The approach needs some modification for Stream Oriented Communication due to its different characteristic.

2.2 Host

In terms of networking, a host is a computer node that is connected to other computer nodes in a network. Every host is identified by its unique hostname which is then mapped to an IP address followed by a port number. Hostname and port number forms the communication endpoint also called as Uniform Resource Locator (URL) for a particular host in a network. A host may be client or server or both. There is a fine distinction between host and server. Every server can be treated as host but every host may or may not be a server. If you run a web server on your local machine and access it via a browser of the same machine then you can treat that host as a localhost. There are various cloud hosts like Amazon EC2, Google App Engine, Windows Azure.

2.3 Application Redistribution

Cloud computing model attracts many enterprises and developers due to its myriads of advantages. There are many cloud providers and cloud service offerings available for hosting applications in the cloud. An application consists of many interacting components. It is completely possible to host some of the components on the cloud (off-premise) and remaining of the application remains on-premise [ABLS13]. Therefore, cloud applications are distributed in nature. Due to different types of cloud services and options available to the developers, choosing the appropriate cloud service and model becomes an important decision. Developers are required to think about various deployment options, adoption strategies, and selection of cloud offerings. Research such as [SALS14] and [GAWM14] has been carried out to guide the developer for taking appropriate decisions. The nature of enterprise cloud applications or any business is such that their requirements change over the time as per customer needs. Therefore, it becomes crucial to accommodate these requirements by making changes to applications and redeploying or redistributing them. This redistribution process may involve a change of application's host which may lead to host splitting as shown in figure 2.1. Applications

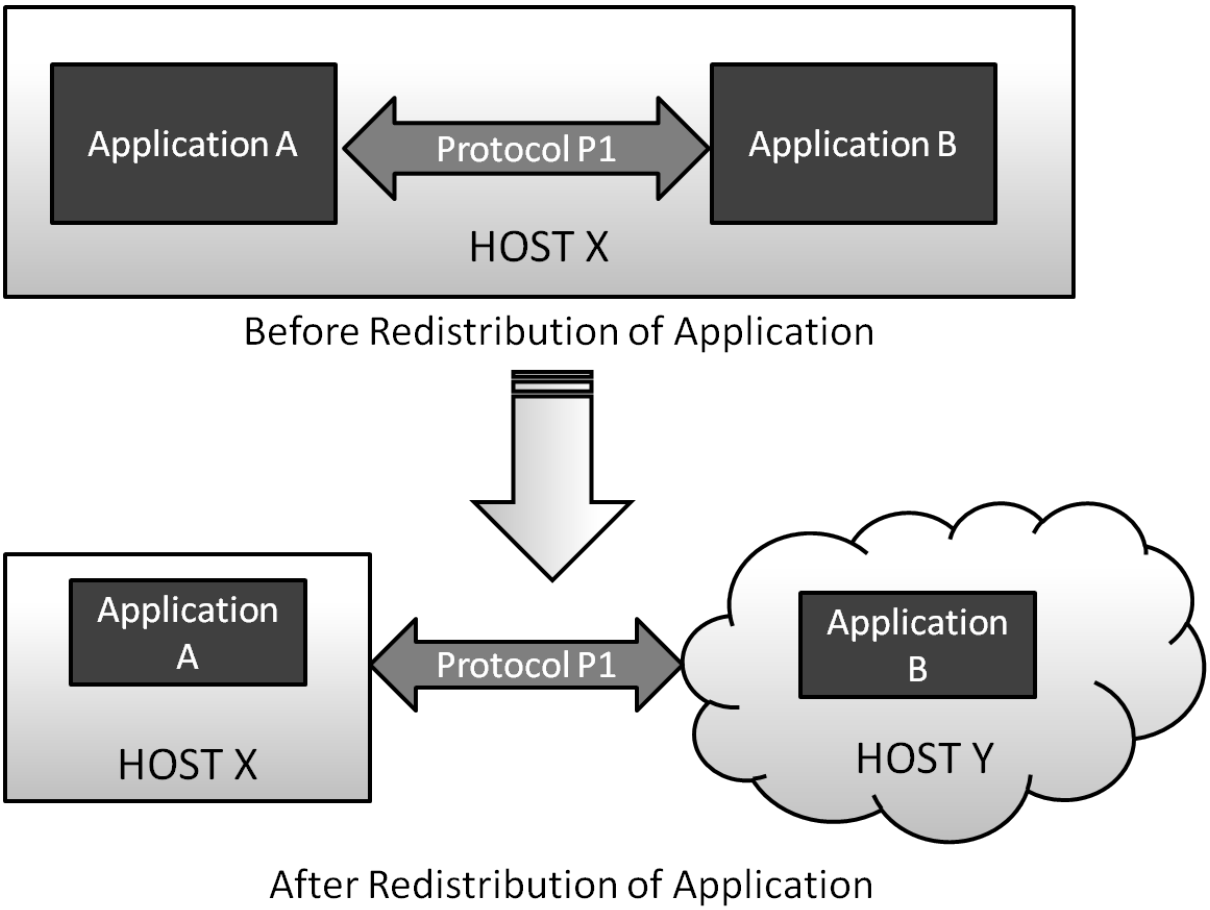

Figure 2.1: Application Redistribution

A and B were residing on same host namely HOST X, but due to change in requirements of application B, it is moved to new cloud host namely HOST Y. The figure depicts the scenario where the underlying communication protocol is unchanged but there may be a possibility where protocol may also be changed to some new protocol.

2.4 Communication Design Patterns

The concept of pattern takes a different meaning in different context [Cop14] such as in object oriented software development field, building architecture, education and cloud computing. In general, a pattern or more specifically a design pattern is a written document that describes a problem-solution pair to a design problem that occurs repeatedly. Every design pattern describes the problem formally and discusses its solution. With reference to various design patterns stated in [HW03][FLR+14], three

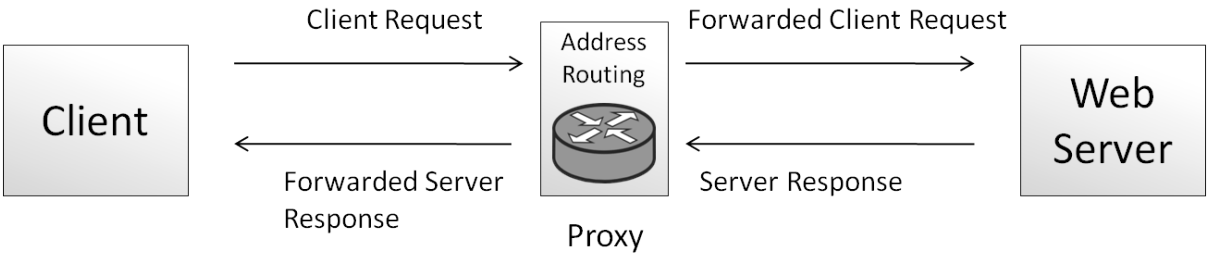

Figure 2.2: Proxy Concept

design patterns namely proxy, translator, and adapter are used in order to implement an approach for handling redistribution of applications. These patterns are discussed in following sections.

2.4.1 Proxy Pattern

In general, a Proxy is an entity that acts on behalf of some other entity. In computer networks, a Proxy server is an entity that acts on behalf of the actual server. Nowadays, use of proxy servers is common in the case of web applications with client server architecture consisting of the web client that makes a request and an actual application server that serves such request. A proxy server sits between such client and application server as shown in Figure 2.2 which is capable of serving the client's request. Such proxy server can actually forward the request to the actual application server as well and may not process the request at all [Bea]. Such proxies involve address routing mechanism to forward the request to the actual application server.

It is necessary for a proxy to know the communication endpoint of actual application server so that it can forward the client request. Here by communication endpoint, we mean the hostname and the port number of the actual application server. The advantages and disadvantages of using proxies are as listed below.

Advantages

1. **Improved performance:** Proxies have the ability to save the contents that are frequently retrieved by clients. These contents are called as a cache. Therefore, client's requests can be quickly served if the contents requested are found in the cache of the proxy. This reduces the waiting time for a client for its response.
2. **Anonymity:** Since proxy sends a request to the application server on client's behalf, client's identity such as geographical location and IP address will not be disclosed.

3. **Filtering:** It can filter the client's requests. The proxy server can be set up to block malicious and inappropriate web sites.

Disadvantages

1. **Security:** If the proxy is compromised then data theft can happen easily.
2. **Cache consistency:** Proxy cache needs to be consistent with the application server. If not then the data served to the clients may not be the latest which gives rise to data inconsistency issues.
3. **Maintenance:** Since proxy is an additional component, it has its own cost for installing and monitoring.

2.4.2 Translator Pattern

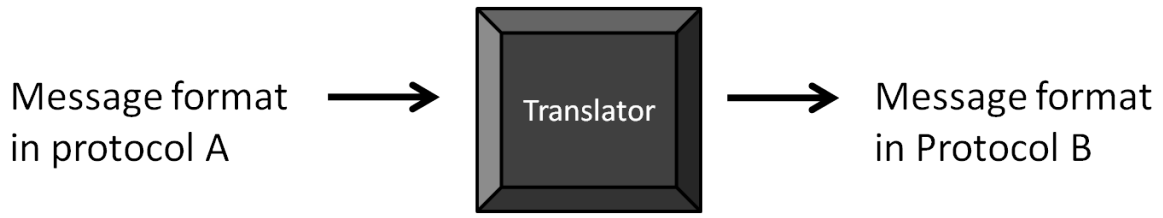

Figure 2.3: Translator Concept

In the case of messaging systems, translator is one of the enterprise integration patterns that transform the messages from one format to the other [Hoh16][HW03]. In this thesis, translator is a generic component used that should be able to perform protocol conversion. Messages from one protocol format are to be converted to another protocol format as shown in figure 2.3. There exists a number of application layer protocols over which applications can communicate with each other. It is possible that two applications have support for two different protocols and we want to establish communication between them. In such scenarios, there arises a need for translating the protocol from one format to other. For example, consider an application A is supporting HTTP and some other application B is working over MQTT protocol. In that case, if we want to transmit some data from A to B or from B to A then translator must be able to convert data formats to and from HTTP and MQTT. There can also be a situation where in two applications support HTTP protocol but the messages are exchanged over MQTT protocol. The prototype implemented for this thesis covers such scenario. Therefore, the code that maps HTTP requests/responses to MQTT messages and vices-a-versa is treated as a translator code. More details about how we have used translator design pattern can be read in Chapter 4 and Chapter 5.

2.4.3 Adapter Pattern

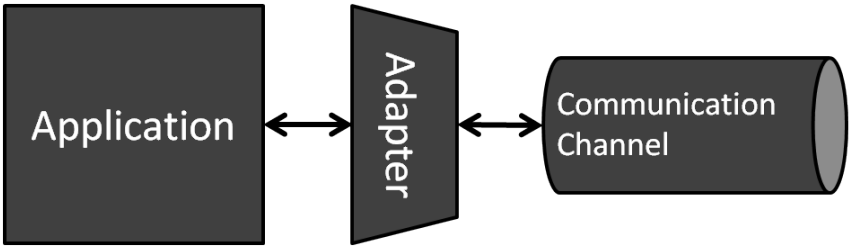

Figure 2.4: Adapter Concept

Normally an adapter has a different meaning in different contexts. In a case of hardware, an adapter is a physical device that allows one electronic interface to be adapted to the other electronic interface. In terms of software engineering, an adapter is a design pattern that is used when you want to convert one interface of a class to another interface so that incompatible classes can work together. In a case of messaging systems, adapter or to be more specific channel adapter is enterprise pattern that allows an application to connect to the messaging system as shown in the figure 2.4. So the adapter act as a client to messaging system [Hoh16] [HW03]. In this thesis, an adapter is a component that has a similar function to that of channel adapter which allows messages to be transferred on the channel so that they can reach appropriate destination. For example, in a case of MQTT protocol, the process of connecting publishers or subscribers to MQTT broker will be treated as an adapter code. Realization of such an adapter pattern is explained in Chapter 4 and Chapter 5.

2.5 Communication Protocols

Multiple applications can communicate with each other and exchange data over the Internet. This communication is governed by application layer protocols defined in TCP/IP protocol model [For02] and ISO OSI protocol model [Zim88]. There are various application layer protocols such as HTTP, FTP, MQTT, SMTP, SNMP etc. used for various types of communication. In this thesis, HTTP and MQTT protocols are discussed in details in order to understand the prototypical implementation of the concepts. These protocols are explained in detail in following sections. Section 2.5.3 provides comparison between these two protocols.

2.5.1 HTTP Protocol

HTTP stands for Hyper Text Transfer Protocol. It is most commonly used Internet protocol and serves as a foundation of data communication for the World Wide Web(WWW). In this thesis, HTTP is referring to as HTTP/1.1 as defined in RFC2616 specification [Fie99]. It is based on request-response mechanism between client and server. When you type any URL in your web browser e.g. *http://www.google.com*, you are making an HTTP request to Google web server. The web server then receives the request and sends HTTP response back to the client which is nothing but a Google web page. HTTP protocol works on port number 80 by default. Since HTTP is based on TCP, it is reliable and connection oriented protocol. But HTTP is stateless meaning no state is maintained between two request-response pairs. Following sections shortly describes HTTP Requests and HTTP Response structure.

2.5.1.1 HTTP Request

HTTP Request is HTTP message sent by the client to the server. Such HTTP message consists of:

- request line
- (General|Request|Entity) headers (zero or more)
- An empty line i.e. CRLF
- message body (optional)

Listing 2.1 HTTP GET request example

```
1 GET /WebApp1/index.html HTTP/1.1
2 HOST: www.testhost.com
```

An example of simple HTTP request is shown in Listing 2.1. The first line is called as request line and it contains HTTP method i.e. *GET*, a request Uniform Resource Identifier(URI) i.e. */WebApp1/index.html* and HTTP version i.e. *HTTP/1.1*. The next line contains request header called host i.e. *www.testhost.com* where the resource *index.html* can be found. We are not sending any data through message body as we just want to retrieve the web page *index.html* from the web server on *testhost.com*. There are various HTTP methods but most commonly used are GET, POST, PUT and DELETE as they represent read, create, update and delete operations on data shown in Table 2.1.

Method	Action
GET	Retrieves data from the server
POST	Sends the data to the server and create new entity
PUT	Sends the data to the server and update the existing entity
DELETE	Deletes an entity from server

Table 2.1: Most commonly used HTTP methods

Listing 2.2 HTTP POST request example

```
1 POST /WebApp1/welcome HTTP/1.1
2 HOST: www.testhost.com
3 Content-Type: text/plain
4
5 hello
```

Another example of simple HTTP POST request is shown in Listing 2.2. Listing 2.2 shows how to send the data to the server in a message body via HTTP POST method. Here, the plain text data *hello* is sent. The Listing 2.2 indicates that it is an HTTP POST request to HOST *testhost.com*. Content-Type is one of the Entity Header that indicates sent data is plain text form.

2.5.1.2 HTTP Response

Once the server receives the HTTP request as shown in the previous section, it processes the request solely based on request contents and generates the HTTP response. HTTP response is made up of:

- status line
- (General|Response|Entity) headers (zero or more)
- An empty line i.e. CRLF
- message body (optional)

Listing 2.3 HTTP response example for GET request

```
1  HTTP/1.1 200 OK
2  Date: Mon, 1 Jul 2017 12:00:00 ECT
3  Server: Apache/2.2.30 (UNIX)
4  Last-Modified: Wed, 21 Jul 2016 19:00:56 ECT
5  Content-Length: 70
6  Content-Type: text/html
7  Connection: Closed
8
9  <html>
10 <body>
11 <h1>Hello from index.html</h1>
12 </body>
13 </html>
```

An example of simple HTTP response with respect to HTTP GET request Listing 2.1 is shown in Listing 2.3. The first line contains HTTP version and status code of 200 indicating successful processing of the request. A status code is a three-digit integer where the first digit indicates a class of response. There are five categories of status codes as follows:

- 1xx: Informational- It implies that the request is received and the process is continuing.
- 2xx: Success- It implies that the request is received, accepted and understood.
- 3xx: Redirection- It implies that some action needs to be taken in order to complete the request.
- 4xx: Client Error- It implies that some syntax error in the request and hence cannot be processed.
- 5xx: Server Error- It implies that server has failed to process the request even though it was a valid request.

Subsequent lines show date, server name, and version, the date when the page was last modified. Content length indicates the length of returned body in bytes. Content type shows it is an HTML form and lastly the actual web page is returned in the body of the response. Some web applications adhering to the same HTTP request-response mechanism have been implemented in the prototype of this thesis.

2.5.1.3 REST

REST stands for Representational State Transfer. The term was first coined and defined by Roy Fielding in his Ph.D. [FT00]. REST is not a protocol or technology but it is an architectural style for building distributed applications. REST is not only used with HTTP but it is mostly implemented based on HTTP. REST is characterized by a set of constraints such as having a client-server architecture, layered system, statelessness, having a uniform interface and Cacheability. These constrained are briefly described as follows:

1. Uniform interface: This is the fundamental principle of REST style. While using HTTP as a transport protocol, all resources are identified by URIs (Uniform Resource Identifier). The resources (e.g database) and the representation (e.g. HTML/XML/JSON) holds different meaning. Manipulation of theses resources through these representations are possible by the clients. Every message communicated between client and server is self-descriptive meaning it contains enough information about how to process the message.
2. Client-Server: Client-Server constraint focuses on separation of concern. By separating the client (user interface) from the server (web server or application server), multiple clients can work with the same server and reduces the complexity of client.
3. Layered system: The client may be connected to the server directly or through some intermediate components. The position of the client should not affect the overall behavior of the system. Intermediate components may enforce some security rules.
4. Stateless: Similar to HTTP, no context information of the client is stored on server side. All the information is contained in request URL.
5. Cacheability: REST enforces the responses to clearly define themselves as cacheable or not so that clients will come to know whether the response can be stored or not. This helps to prevent clients from caching stale or inappropriate responses and improves performance and scalability.

Web application's functionality is exposed to the outer world by exposing an API (Application Programming Interface). The API that adheres to REST constraints stated above is called as RESTful API. HTTP based RESTful APIs uses same HTTP methods as described in Table 2.1.

2.5.2 MQTT Protocol

MQTT stands for Message Queue Telemetry Transport. It is an open, simple and lightweight protocol used in Machine to Machine communication and Internet of Things (IoT) industry. MQTT has now become an OASIS standard [OAS]. In contrast to HTTP's request-response pattern, MQTT follows Publish-Subscribe messaging pattern which is explained in following section.

2.5.2.1 Publish-Subscribe Messaging Pattern

Messaging systems based on Pub-Sub pattern consists of a number of senders called as publishers of the message and number of receivers called as subscribers of the message. Publishers publish a message to a particular topic or logical channels without knowledge of any subscribers. Subscribers, on the other hand, subscribe to one or more of these topics without knowing the existence of publishers. All subscribers will receive the messages published to the topics to which they have subscribed. In general, these publishers and subscribers are treated as clients. Figure 2.5 gives an idea of the publish-subscribe messaging system working over MQTT protocol.

2.5.2.2 Broker

The clients of the publish-subscribe messaging system require a mediator called as a broker. The broker is solely responsible for transmitting the messages between the clients. It decides whom to forward the published messages based on client's subscription. MQTT broker can be set up on localhost (private MQTT broker) or can be set up on the cloud (public MQTT broker). There are many MQTT brokers publicly available but in this thesis, we are using Mosquitto broker [com]. Mosquitto broker implements MQTT protocol version 3.1.1 [Lig17]. MQTT works on top of TCP/IP hence both the clients and the broker must have TCP/IP stack [Hiv17].

2.5.3 HTTP vs MQTT

HTTP is the widely used Internet protocol but for IoT, MQTT is more suitable because of its less complexity and overhead communication. A comparative study of HTTP and MQTT has been stated in [YS16], and it is concluded that MQTT performs better than HTTP. This section will differentiate between the two protocols explained in previous sections. The comparison is summarized in table 2.2

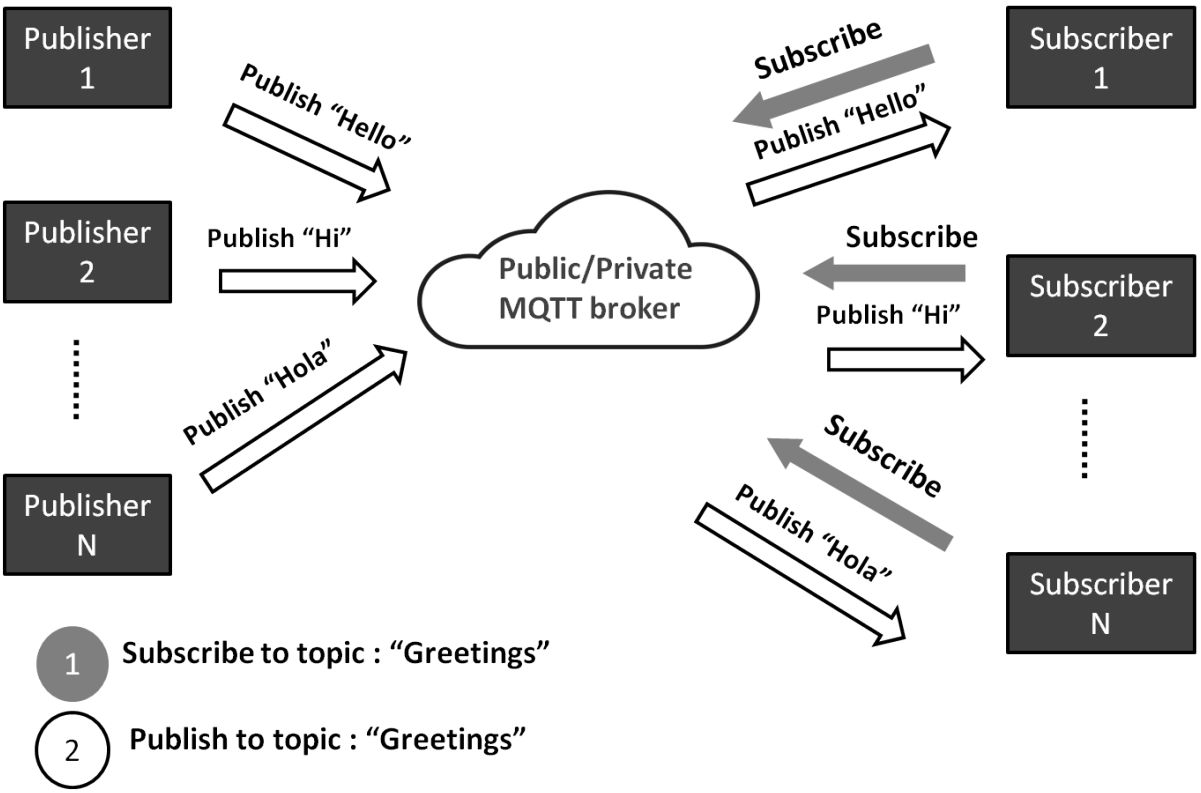

Figure 2.5: Publish-Subscribe system based on MQTT protocol

Parameter	**HTTP**	**MQTT**
Full-form	Hyper Text Transfer Protocol	Message Queue Telemetry Transport
Architecture	Request-Response Pattern	Publish -Subscribe Pattern
Use of broker	No	Yes
Design methodology	Document centric	Data centric
Security	HTTPS is used	Yes
Ports used	80 or 8080	1883
Complexity	Complex	Simple
Data distribution	1 to 1 only	1 to 0/1/N
Underlying protocol	UDP	TCP
QoS levels	None	3 QoS levels

Table 2.2: HTTP vs MQTT comparison

2.6 Summary

In order to maintain same communication even after redistribution of application and to support protocol change, this thesis has attempted to generate proxies consisting of address routing mechanism, translators, and adapters. In general, these patterns are stand alone patterns but in this thesis, adapter and translator pattern are included in the proxy pattern. This chapter has stated all the necessary fundamentals needed to support the suggested approach in order to achieve the goal of this thesis.

3 Related Work

This chapter discusses the existing literature related to the thesis work by considering the fundamentals explained in the previous chapter. In this chapter we will discuss approaches developed for generating the proxies in a case of distributed applications in section 3.1 and also some of the approaches for protocol translation are discussed in section 3.2. This section also includes some of the work that has been done in order to bridge the communication gap between application layer protocols like HTTP and MQTT. It guides in implementing the prototype for this thesis. Section 3.3 states the role of an adapter in application integration scenario.

3.1 Usage of Proxy in Distributed Systems

3.1.1 The Proxy Principle

The proxy pattern described in section 2.4.1 is commonly used in case of distributed application which follows client-server architectural style. Shapiro [Sha86] has proposed the proxy principle to structure and achieve the encapsulation in the distributed systems. In client-server architectural style, the client invokes some of the service objects presented by the server. The structure of these service objects may be complex but such complexity should not be visible to the client. This encapsulation can be provided by using proxy objects. The proxy principle states that in order to use some service, a client must acquire the proxy object first. This proxy is the only visible interface to the service. Objects represented by a proxy are treated as its principles. The proxy together with its principles forms a single distributed object, which is treated as a group shown in figure 3.1. Such system must implement following property:

- The proxy will always remain local to the client.
- All communication to server happens through the proxy.
- The proxy is transparent to the client.
- Internals of the proxy are not visible to the client. The proxy has full visibility of the group.

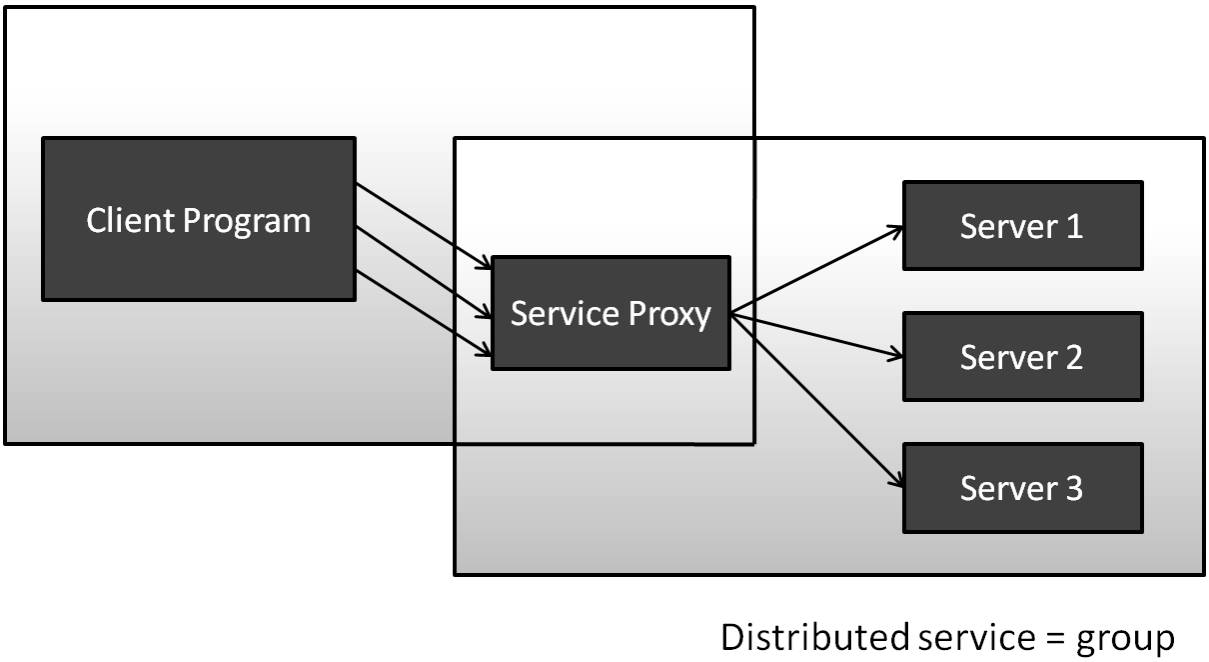

Figure 3.1: A proxy for a distributed service [Sha86]

It is claimed to be the powerful structure and provides better structuring and encapsulation. By using this proxy principle as the basis, an approach is proposed in Chapter 4 for application redistribution scenarios.

3.1.2 Distributed Application Proxy Generator

The invention [Jus07] relates to client server architecture pattern in distributed software applications. A distributed application is one which consists of two or more computer systems that are connected to each other over a network and one of them act as a server while others being clients. By using DAC (Distributed application contract) written in IDL (Interface Definition Language) and configuration file, client side proxies can be automatically generated between the client application and the server application. The relation between client, server, and proxy has already been illustrated in Figure 2.2 in the previous chapter. The DAC written in IDL describes the distributed application's functionality via its interfaces and method definitions while the configuration file consists of information regarding server's location in terms of network addresses such as URL of server implementation. The DAC and configuration file are the inputs to the proxy generator which consists of a lexer, parser, and code generator. Figure 3.2 gives an overall idea of how the proxy is being generated. Java compiler is receiving the outputs from IDL-to-Java Generator, server code generation function, client code generation

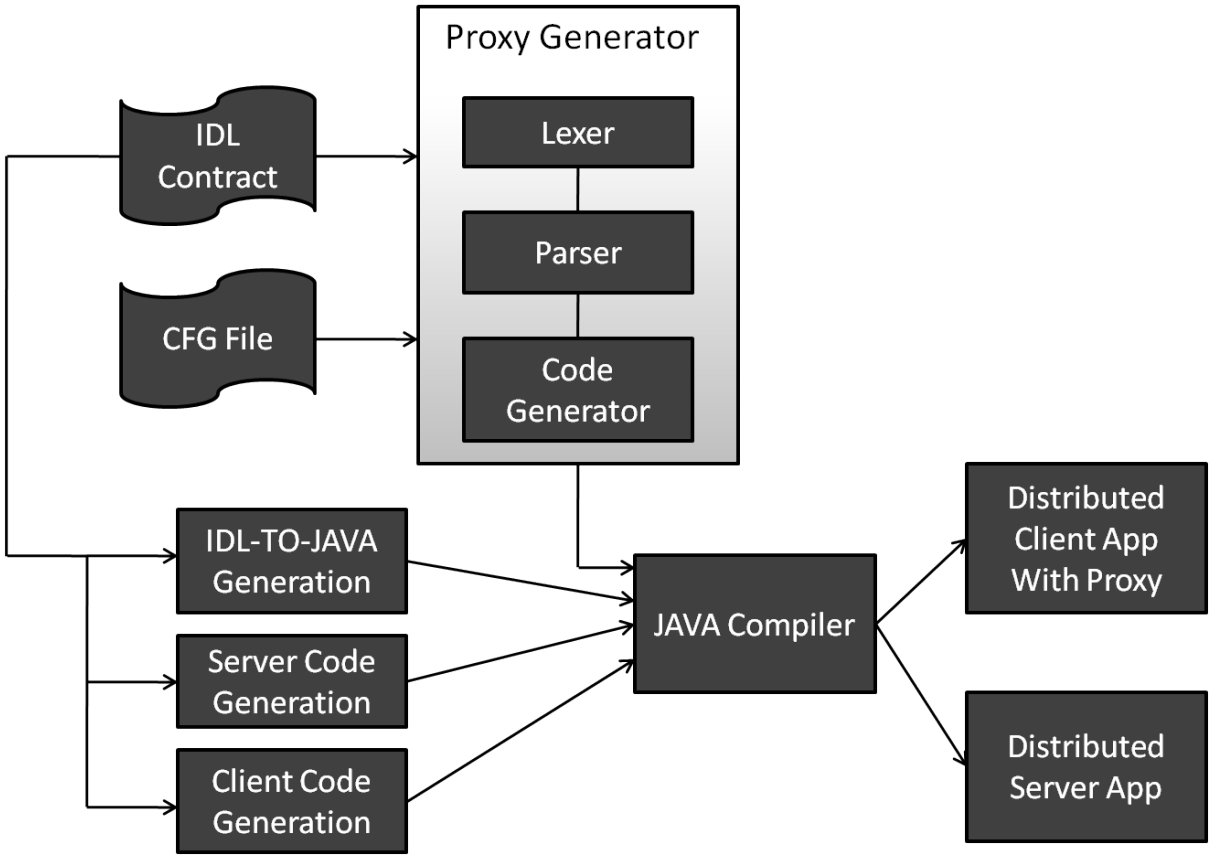

Figure 3.2: Distributed Application Proxy Generation as per invention [Jus07]

function together with the output from the proxy generator to generate client application with proxy and server application.

Such an approach offers many advantages for the rapid development of distributed application, error handling and lowering the maintenance burden but the invention under discussion focuses on Java programing language related to CORBA/IDL-based distributed application [VD98]. Even though it is claimed that it can be extended to Java/RMI(Remote Method Invocation)-based applications [Gro01] or MICROSOFT's DCOM(Distributed Component Object Model) [Cor96], protocol change is not supported. We are using the similar concept of building the proxy generator to inject the proxies between applications along with a translator to support protocol conversion and an adapter component. More details about the design of proxy generator are stated in Chapter 4.

3.1.3 Proxy Injection using Client Runtime Library

Another approach [AYD+03] of transparently injecting the proxies between client and server application is invented. Client runtime library is interposed between client and server application. This library consists of routines which are bound to program and are executed when the program is running. Such a library is responsible for injecting the proxies into the system. The generated proxy implements the server interface hence it is necessary to analyze the server's functionality via server's source code. This approach also requires modifying client application and replacing the call for a remote object by the call for the proxy. Such an approach provides an advantage in the sense that proxies can be injected into existing distributed application. Also, the proxies can cache the server's object at client side so to serve the request locally and reduce round trips between client and server application.

In this approach, it is required to have access to the source code of applications under consideration. The client application is manually configured so that a call for a remote object is replaced by the call to the proxy. Also, this method does not support protocol change mechanism, unlike the approach which is presented in the thesis.

3.2 Protocol Translation

As stated in section 2.5, there exist a number of application layer protocols for communication. The complexity of communication network can vary from the simplest form such as a single LAN (Local Area Network) to complex form involving multiple LANs that are connected via WAN (Wide Area Network) such as the Internet. Such networks involve various nodes such as PCs, workstations, PDAs, server computers, peripheral devices like printers, scanners, file systems, network components like bridges, routers, and gateways. These nodes and components might work on different protocols in different networks and there are increasing demands which require one network to connect to another network. In such cases, it is necessary to convert from one application layer protocol to another. Bridge networking component enables to connect two LANs that run the same protocol. While router interconnects two networks, it works till the network layer of the protocol model and forwards the message to correct destinations based on the destination address of the message. On the other hand, the gateway acts as a protocol converter and works till the application layer of protocol model but designing a gateway costs lot of time and money. Also, troubleshooting the gateway is a complex process. Adhering to these facts, there exists a need to design protocol conversion strategy.

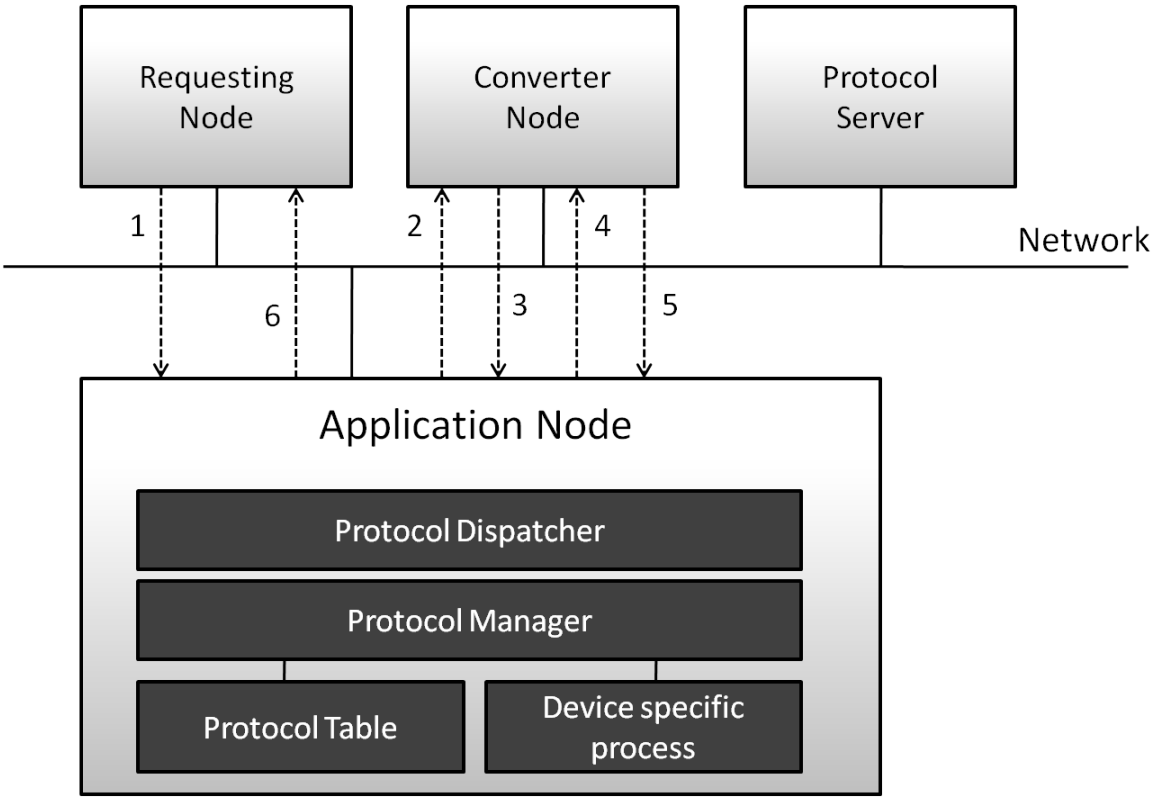

Figure 3.3: Protocol Translation Mechanism as per invention [Ara07]

3.2.1 Application Layer Protocol Conversion

A system [Ara07] for converting one application layer protocol to another is described. A system comprising of Requesting node which can be treated as client, Application node which can be treated as an application server and Converter node which can be treated as a translator is shown in figure 3.3. The requesting node and application node supports different protocols say P1 and P2 respectively and converter node is used to translate between these protocols. The operation of the system can be explained as follows:

1. Requesting node sends a request message to application node in protocol P1.
2. Application node receives the request message and forwards it to the converter node. Protocol dispatcher is responsible for forwarding the request by consulting protocol table. For every protocol, this protocol table contains an entry for an address of requesting node and the corresponding address of converter node. The converter node, after receiving the request message in the format of protocol P1, performs the translation of request message to protocol P2.

3. The converted message is sent back to the application node by converter node in protocol P2.
4. Application node, after processing the request, sends the response message to the converter node. Converter node translates the response message from protocol P2 back to protocol P1.
5. This converted response message is then sent to application node by the converter node.
6. Finally, the response message is forwarded to original requester node by the application node.

Another system [KOY98] for application layer protocol conversion is invented. It is stated that two nodes in two different networks using different protocols can communicate with each other using a protocol conversion system [KOY98].

3.2.2 HTTP and MQTT Bridging

In IoT world, a huge amount of tiny data is transferred between different types IoT devices per second. MQTT, which is best suitable for IoT communication has gained a lot of attention in recent years and is being the topic of worldwide interest. On the other hand, HTTP is the widely used Internet protocol and has been applied for data transfer. Therefore, a lot of different projects and efforts has been done so that HTTP and MQTT work together and leverage the advantages of both the protocols. This section discusses over such projects.

3.2.2.1 QEST Broker

In IoT world, most of the embedded devices have the ability to work with lightweight protocols and unable to cope with high-level protocols like HTTP. MQTT is one of the lightweight protocol as mentioned in section 2.5.2. In order to exploit the power of MQTT, libraries have been provided for major development platforms like Arduino, Android, and iOS. MQTT is primarily based on the publish-subscribe model of communication and uses a broker as an intermediary element. A new broker called as QEST Broker [CCV12] is implemented that bridges the gap between HTTP and MQTT protocol and aims at exposing the MQTT topics as a REST resource. The QEST Broker architecture is shown in Figure 3.4 and can be explained as follows:

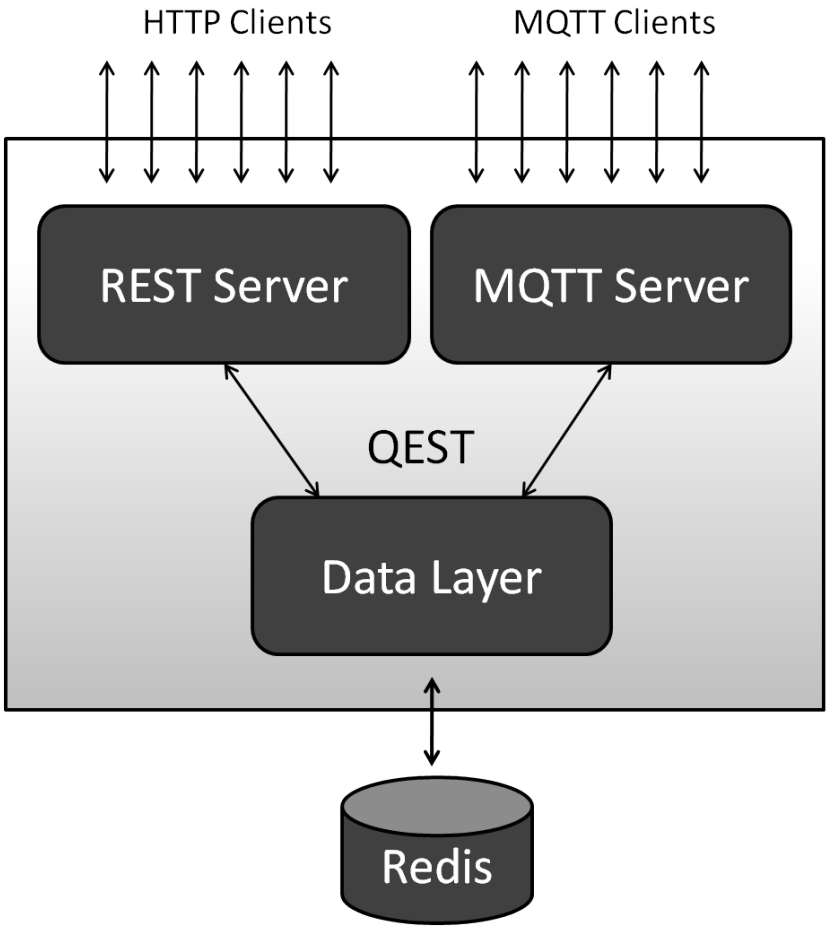

Figure 3.4: QEST Broker architecture [CCV12]

1. QEST has both MQTT Server and HTTP Server, therefore when client publishes the new message via REST front-end or MQTT front-end, writes the new value to the data layer
2. Redis [Car13] is the key-value pair database used in the implementation of QEST. The data layer itself stores the value and further writes the update to Redis key.
3. Clients can subscribe to REST or MQTT front-end resulting in changes to the data layer. Data layer, in turn, is subscribed to a particular Redis key.
4. Whenever a new message is updated on a value of Redis key, the message is forwarded to the data layer which further notifies the subscribed clients via front-ends.

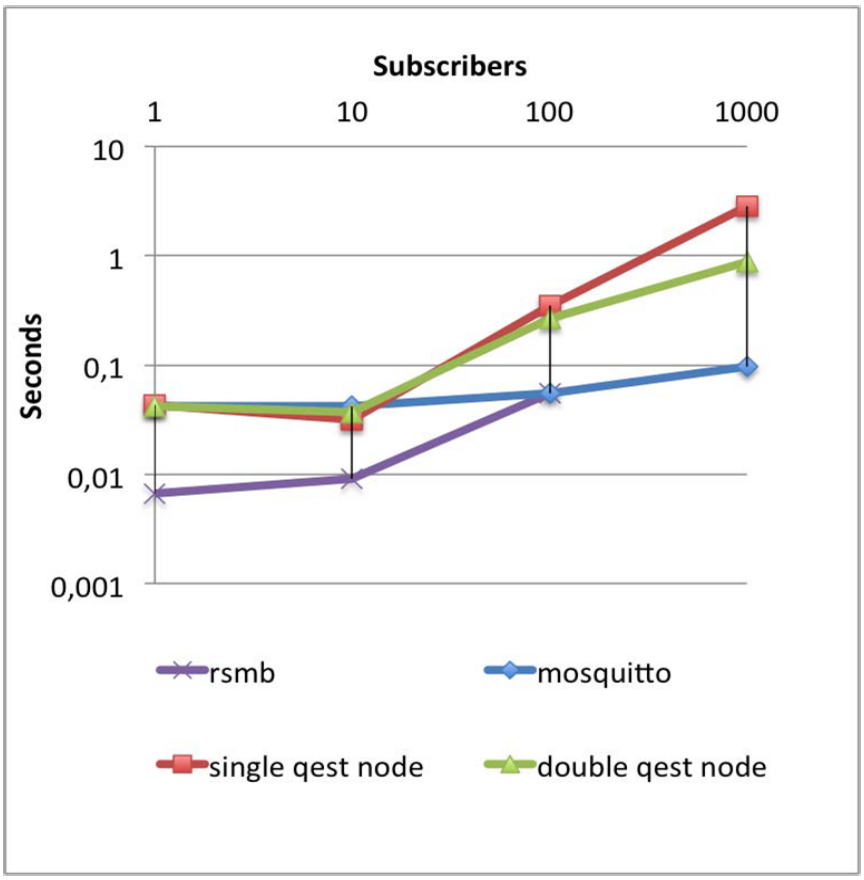

Figure 3.5: Performance comparison between Mosquitto, RSMB, single QEST node and double QEST node [CCV12]

QEST Broker can also be scaled horizontally meaning more than one QEST broker can be used that can act as a load balancer. The performance of QEST Broker is compared [CCV12] to other brokers like RSMB (Really Small Message Broker) [IBM] made by IBM and public open source broker called Mosquitto [com]. As shown in Figure 3.5, one can deduce that QEST performs is comparable to Mosquitto between 1 to 10 subscribers. Also, double QEST node performs faster than single QEST node as a number of subscribers increases.

3.2.2.2 Ponte Project

Ponte project [Fou13] bridges the gap between Internet of Things- Application Layer Protocols like MQTT, CoAP [SHB14] and HTTP. Ponte is a multi-transport IoT/Machine-to-Machine broker supporting MQTT, CoAP and HTTP [CBVC14]. It allows to have communication by publishing the messages from MQTT/CoAP enabled clients to HTTP

and vice versa. Figure 3.6 shows Ponte's architecture. Ponte uses some of the publish-subscribe brokers such as RabbitMQ, MongoDB, Redis, and Mosquitto. Once the data is published or submitted by one of the clients, it must be stored persistently until it is received. Therefore, Ponte supports MongoDB, LevelDB, and Redis as persistent storage engines. In general, Ponte can be deployed on top of various databases and brokers, and also can be extended to have support for more protocols. Currently, it supports communication between HTTP, MQTT, and CoAP. Mixing of the protocol can be done, meaning you can POST or PUT the data via HTTP and subscribe via MQTT. Essentially, Ponte is a node.js application and is still under development.

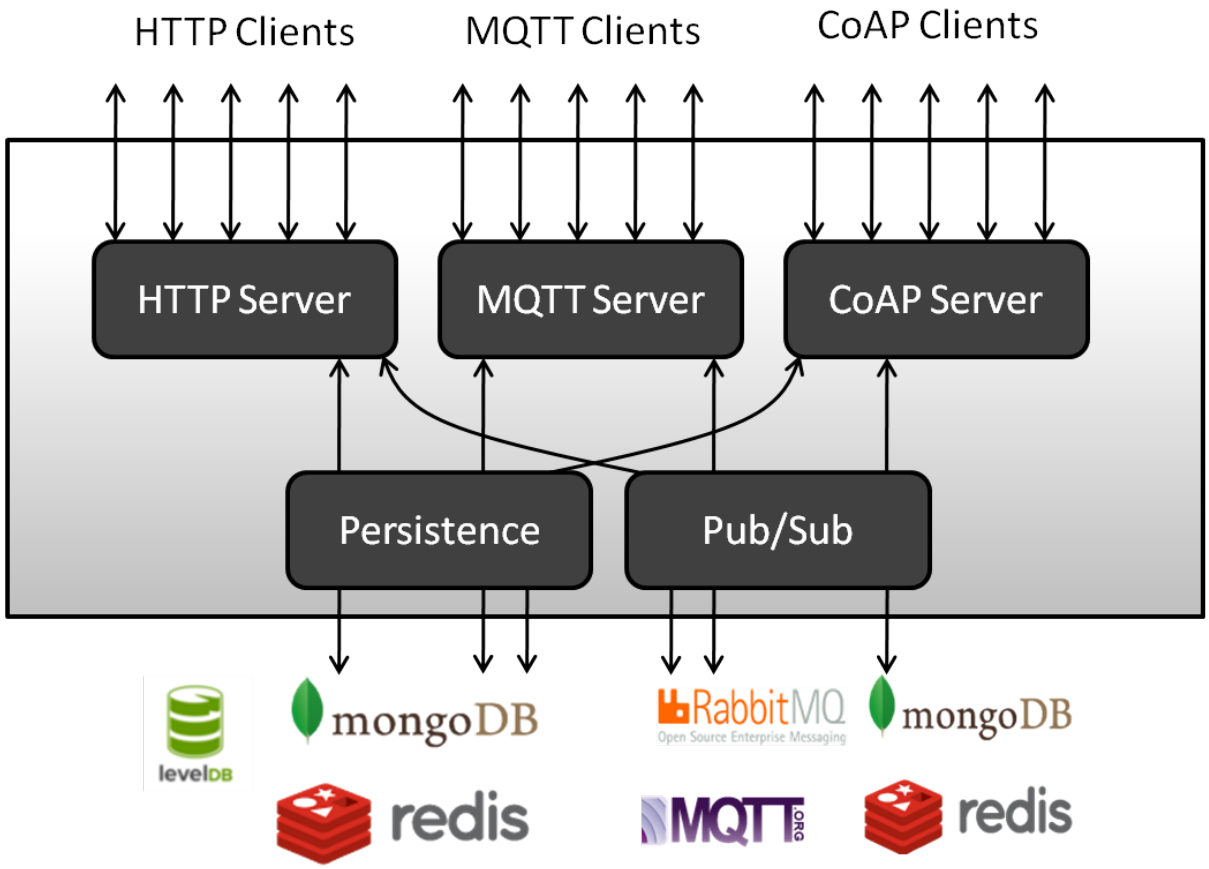

Figure 3.6: Ponte architecture [Fou13]

3.3 Use of Adapters in Application Integration

Enterprise Application Integration or EAI is a process of creating a business solution by combining applications [RBM01]. Resources such as data and functions are brought

from one application program to another. This is achieved by using some middleware technologies. These technologies are application independent and provide services that mediate between applications. An adapter component can be a part of such middleware technologies. In other words, application integration uses integration framework that is based on adapters which help to integrate your applications into your enterprise. Before the advent of middleware technology frameworks and various EAI tools, the task of EAI was followed by an ad-hoc means and included tedious programming with high scale of efforts. But application integration frameworks reduces this complexity by providing mechanisms by which applications can communicate with other applications and exchange of data can happen. Instead of manually wiring the enterprise applications together, integration frameworks build the adapter. Basically, an adapter is the software components that help to connect enterprise systems to an application server. Frameworks are either packaged by a vendor or developed on the custom basis. Most of these frameworks are based on and implement EAI patterns described in [HW03]. There are various open-source Java integration frameworks available for example Apache Camel [Foub], Mule ESB [Mul], and Spring Integration [Sof].

3.4 Summary

By studying the literature, concepts, and related work stated in this chapter, an attempt is made to propose the generic approach of generating the proxies and converting one application layer protocol to another. As stated in section 3.3, an adapter is a software component that is used in application integration process which helps to connect different applications together. This thesis uses an adapter concept for connecting proxies to communication channel. The work stated in section 3.2.2 is useful for implementing the prototype of this thesis. More detailed explanation about the approach followed can be found in Chapter 4 and Chapter 5.

4 Proxy Generation and Injection Approach

4.1 Motivation

An enterprise or web application consists of many interacting software components. These applications are distributed in nature. Due to varying nature of business requirements, cost optimization purposes, performance improvements, or new offerings in cloud platforms, it is a high possibility that applications are redistributed. Because of this, applications can be moved from one host to another and also the underlying communication protocol may be changed. This redistribution, as also shown in figure 2.1, requires applications to be configured correctly so that they continue to communicate as before. Configuring these applications in terms of handling the communication endpoints is not an easy task because these applications are quite big and complex in nature. Also, application modification requires a lot of time and manual efforts. Therefore it is necessary to address the problem of redistribution and design an automation approach.

As discussed in Chapter 2, proxies are the design patterns used to forward the requests or messages to correct destination, translators are the design patterns used for translating messages from one format to another, and adapters are the design patterns used for connecting the application to the communication channel. Using these design patterns, in this thesis, an attempt has been made to suggest an approach of proxy generation and injection.

This chapter describes an approach of automatically generating proxies consisting of address routing mechanism, translator and adapter functionality and how they can be injected between these applications in order to maintain the same communication even after applications have been redistributed to new hosts or underlying communication protocol has been changed. The generated proxies are responsible for routing the requests and messages to the new hostname and new IP address of the redistributed

application and capable of converting the protocol from one format to other. These proxies are generated by a tool called Proxy Generator (PG). The internal design and working of PG are explained in section 4.2.4 of this chapter.

4.2 Approaches

Before diving deep into details of approach, some assumptions are made that are clearly stated as follows:

1. The technology in which applications are developed is not taken into consideration. The concept can be superficially applied to solve the problem of redistribution.
2. We are considering client-server architecture in which client makes a request to the server and server returns a result object as a response to the client.
3. The technical details of the host on which applications are residing are not taken into consideration.

An approach of proxy generation has been proposed in two broad scenarios as follows:

- **Scenario 1:** Underlying communication protocol is same after redistribution of application.
- **Scenario 2:** Underlying communication protocol is changed after redistribution of application.

4.2.1 Design of Proxy

The proxy that is being injected into the system takes the design as shown in Figure 4.1. It consists of three components internally

1. Address routing component: It will reroute the message request to correct destination based on the endpoint given by the user of PG. For example, it will be the new hostname/IP and port number of the new host where the application is moved after redistribution.
2. Translator: Every communication protocol governs particular message format. Therefore, it is necessary to convert the messages to correct format as understood by the application. The translator will translate the message according to the format required by the communication protocol. The protocol has to be specified by the user of PG. Translator component is disabled if the underlying communication

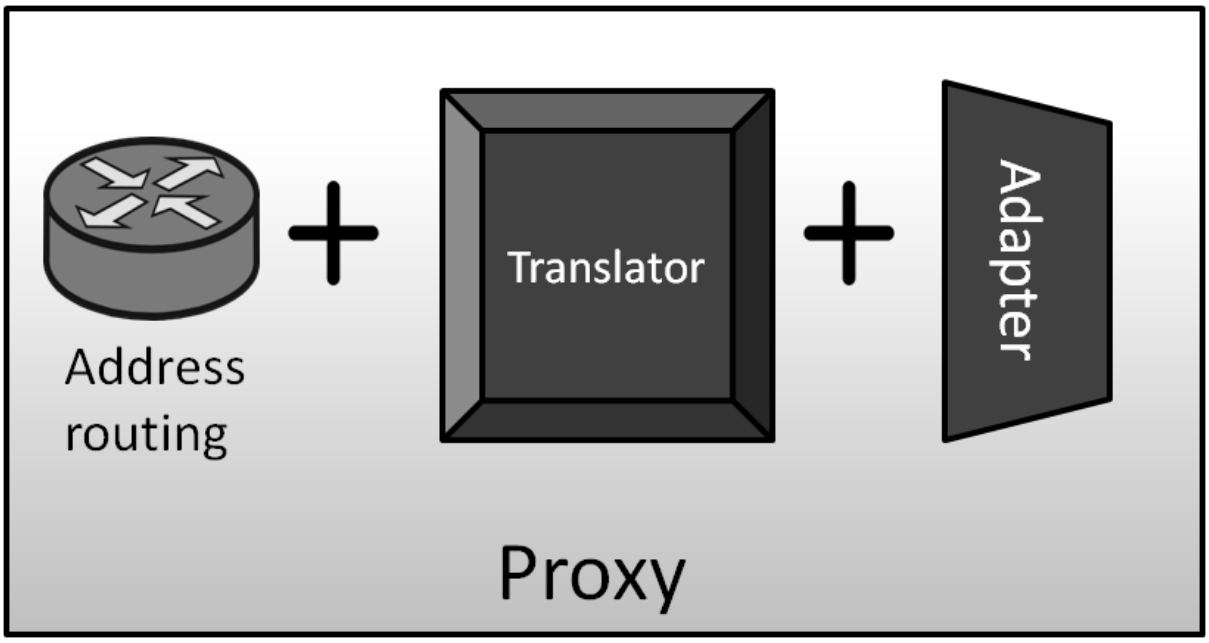

Figure 4.1: Proxy Design

protocol is same as there is no need to perform message translation. While translating the message, the information contained in the message must be retained. For example, in the case of HTTP protocol, a request message consists of header information which must not be lost after translating the message to target protocol format.

3. Adapter: Communication channel connects two applications which may be anonymous to each other. Basically, such a channel is some sort of logical address which is used by senders and receivers to place their data. An adapter serves the functionality of connecting the proxy component to the underlying channel of communication. Channel details such as queue name, message topic name, or intermediate broker ID may be needed, for example in the case of MQTT protocol.

4.2.2 Scenario 1: Same Communication Protocol After Redistribution of Application

As shown in Figure 4.2 , consider two applications, for example, A and B are located on same host namely HOST1 and communicating with each other over protocol P1. Here, the communication can be of type request and response. Application A sending a request to application B and then application B sending the response back to A. Due to change in requirements, application A is moved to new host namely HOST2 and

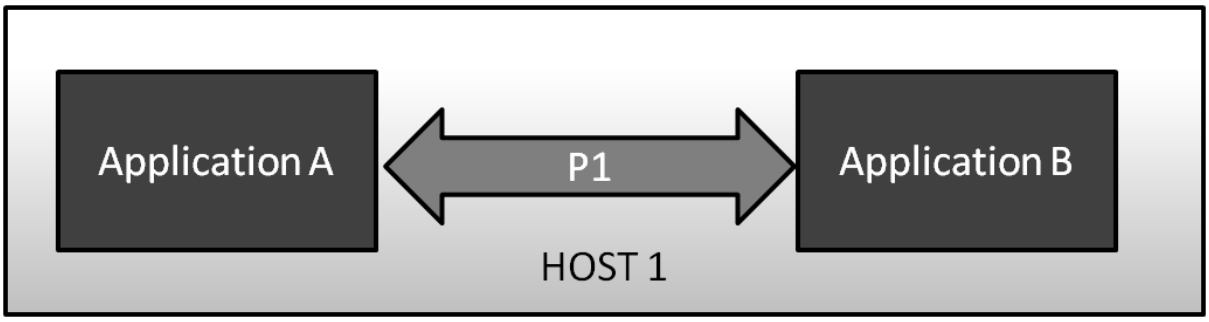

Figure 4.2: Application A and Application B on same Host

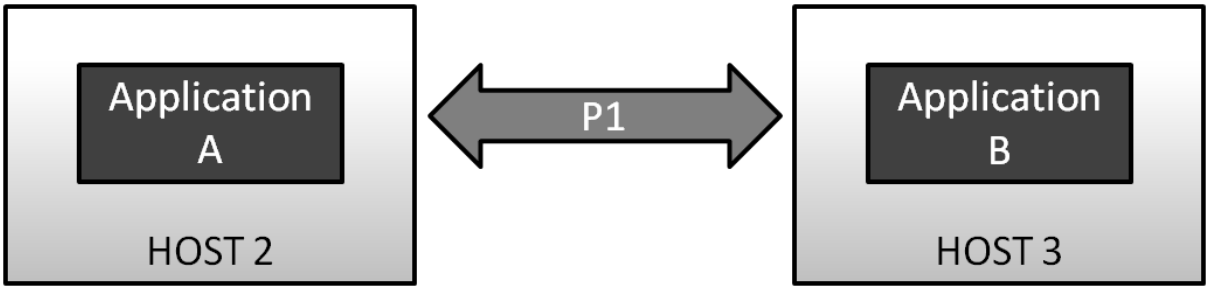

Figure 4.3: Applications A and B are redistributed to different Hosts

application B is now moved to some different host namely HOST3, then Figure 4.3 shows the desirable scenario after splitting of hosts. Note that the underlying communication protocol P1 is unchanged.

4.2.2.1 Solution

In order to maintain the same communication in such cases proxies are introduced as shown in Figure 4.4.

1. The "Proxy for B" will intercept all the requests coming from application A.
2. Since application B is moved to a different host, "Proxy for B" will internally reroute the request to application B.
3. After receiving the request, application B will respond to this request and hence it will be caught by "Proxy for B".
4. "Proxy for B" will forward the response back to application A.
5. Similarly, "Proxy for A" will intercept all the requests coming from application B.
6. "Proxy for A" will forward the requests to application A.

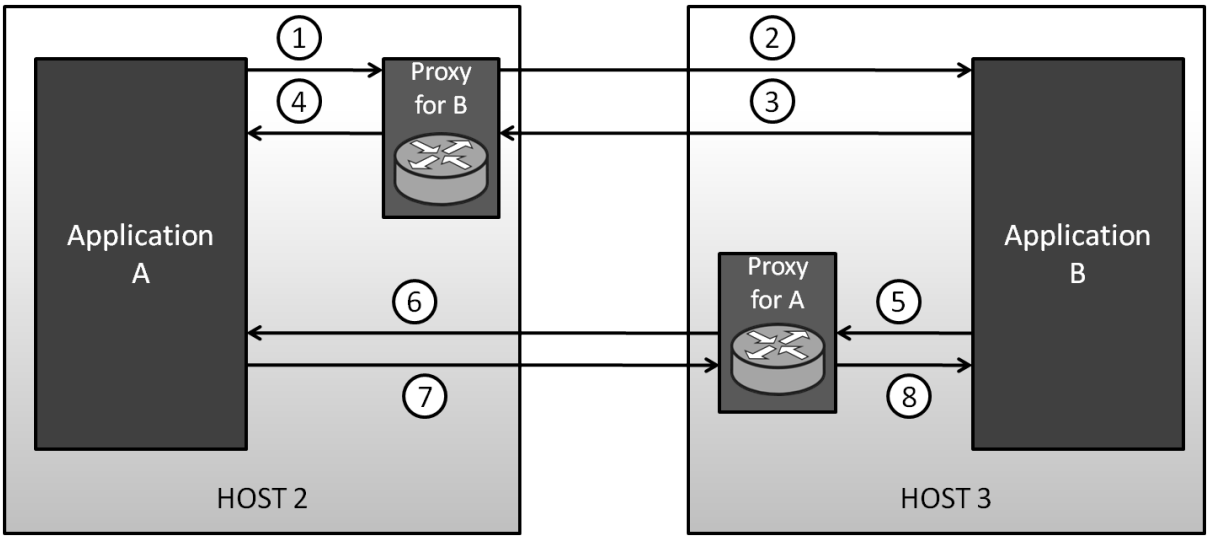

Figure 4.4: Introduction of proxies for same communication protocol

7. After processing, Web application A will send the response which will be captured by "Proxy for A".
8. Finally, "Proxy for A" will forward this response to application B.

4.2.2.2 Discussion

In Figure 4.4, the components "Proxy for A" and "Proxy for B" will have only the address routing mechanism that will forward the requests to the new location where the other application is listening. These proxies will only need to know the new hostname and the new port number of moved application. The new host name and new port number are the input parameters for PG tool. PG will generate desired proxies and inject them into the system. The applications A and B are unaware of the existence of proxy components and will continue to communicate as before. Proxies will not process or modify any original messages of communication.

4.2.3 Scenario 2: Changed Communication Protocol After Redistribution of Application

In this scenario, along with host splitting, the underlying protocol for communication between applications A and application B is also changed as shown in Figure 4.5. Note that the communication protocol is P2. Here, the communication is totally governed by rules as per protocol P2.

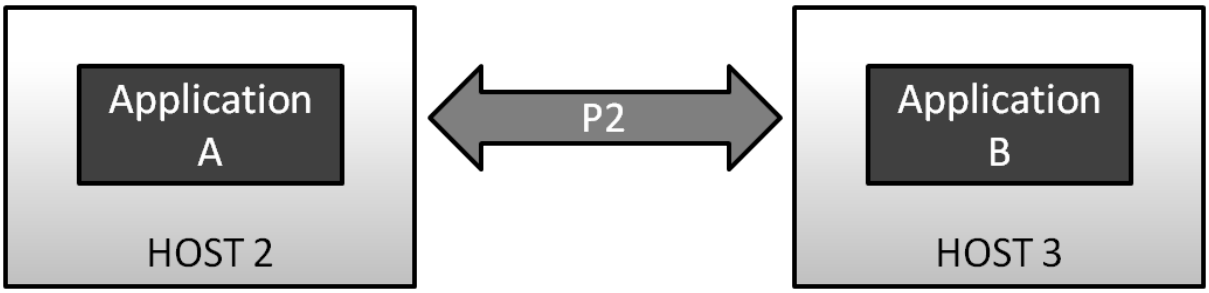

Figure 4.5: Redistribution of applications along with change of communication protocol

4.2.3.1 Solution

In order to maintain the same communication, in such cases, the proxy component having design as stated in Section 4.1 is introduced. Figure 4.6 illustrates an approach.

1. "Proxy for B" will intercept the requests coming from application A.
2. Address routing mechanism is needed in order to reroute the requests to correct destination. The translator will translate the request message to the desired format required by the communication protocol P2. The adapter will serve the functionality of connecting the proxy component to the communication channel. The request is then transferred to channel in the desired format by the proxy.
3. This request is then captured by "Proxy for A".
4. An adapter is needed which connects the "Proxy for A" to the communication channel. Translator re-translates the message to the original format as understood by application B. Finally, address routing mechanism serves the functionality of routing the request to application B. Application B receives the request from "Proxy for A".
5. Application B processes the request and sends the response which will be captured by "Proxy for A".
6. After receiving the response, "Proxy for A" needs to perform address routing, translation of the response message so that it can be sent over communication protocol P2, and connect to a proper communication channel via an adapter. Such a response is then sent over communication protocol P2.
7. "Proxy for B" receives the response and reverse process of translating the response to a format understood by application A is done. Address routing mechanism serves the functionality of routing the response to application A.
8. Finally, application A receives the response in the expected format.

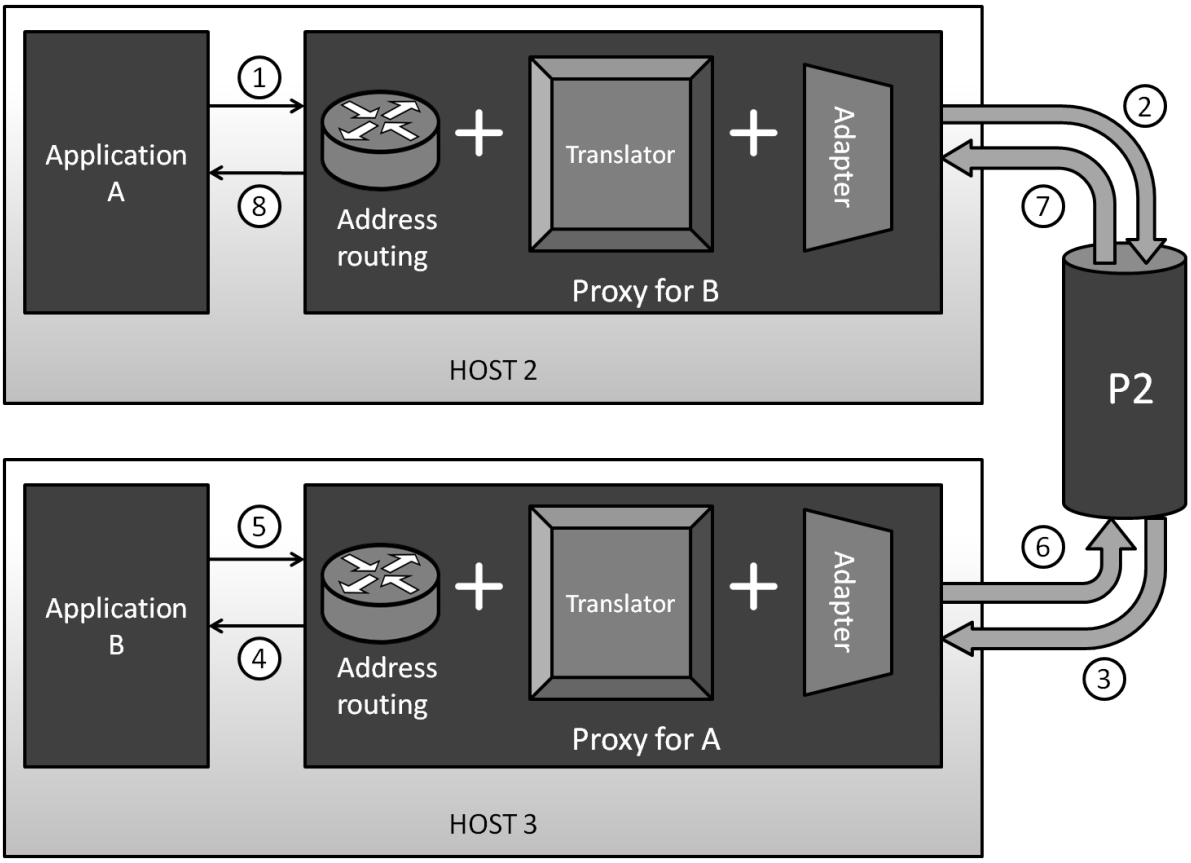

Figure 4.6: Introduction of proxies for changed communication protocol

If application B was requester and application A was responser then the sequence communication flow would be 5, 6, 7, 8, 1, 2, 3, 4 from Figure 4.6.

4.2.3.2 Discussion

In this scenario, all three proxy components are needed at both, requester as well as at responser side because translator and adapter mechanism is needed to support new communication protocol. Address routing mechanism at "Proxy for B" routes the requests to the correct communication channel over protocol P2. In this case, PG needs new hostname, the new port number of moved application and also the channel information for protocol P2. For example, in the case of MQTT protocol, message broker's IP address is needed. Then PG will generate the proxies and inject into the system.

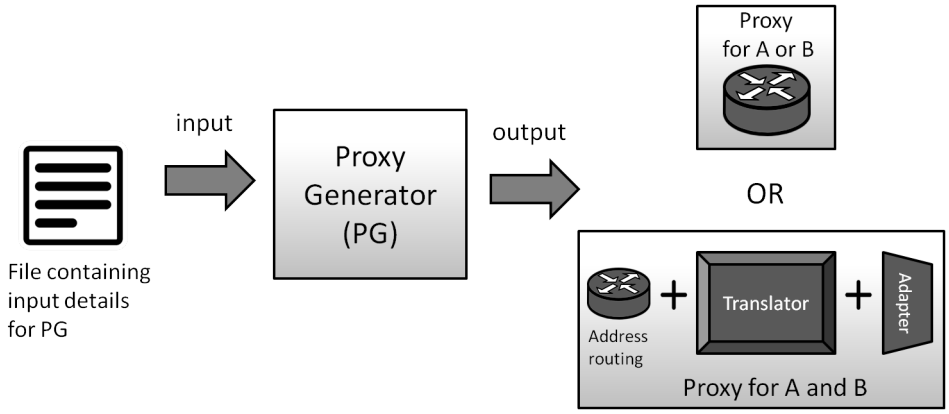

Figure 4.7: Proxy Generator

4.2.4 Proxy Generator (PG)

These proxies are generated by a special tool called Proxy Generator (PG). Figure 4.7 shows Proxy Generator taking some input information provided in a file and producing the necessary proxy component.

4.2.4.1 Input to PG

The input to PG must contain necessary details for producing the appropriate types of proxies. The input to the PG is to be given manually. It must have following information:

- New hostname and port number of application are required for routing the messages to the correct address. This information is required by address routing component of proxy.
- It must have source and target protocol name. This information is required for translator component for converting the protocol from the source format to target format.
- Channel specific details are required by adapter component so that proxy can connect to the communication channel. Basically, the channel is some sort of logical address where senders and receivers place their data. Channel specific details are required only if the source and target protocols are different.

4.2.4.2 Output of PG

As shown in figure 4.7, based on the input information appropriate type of proxy component is produced and injected into the system. The internal design of PG and detailed information regarding input and output of PG is discussed in more detail in section 5.4 of Chapter 5.

4.3 Summary

In this chapter, an approach of proxy generation for solving the problem of redistribution of application is proposed. Two broad scenarios in which the design of generated proxy differs are discussed. In the case of same protocol, generated proxy only consists of address routing component but in case if protocol changes then generated proxy involves translator and adapter in addition to the address routing component. Proxy Generator (PG) is a tool used for generating these proxies. This chapter has introduced the PG in brief. In next chapter, all implementation details of PG and considered scenarios are explained in detail.

5 Implementation

5.1 Test Environment

In the previous chapter, proxy generation approach was discussed in following two scenarios.

1. Same communication protocol after redistribution of application.
2. Changed communication protocol after redistribution of application.

A prototype for implementing these scenarios is built. In implemented prototype, applications are basically web applications communicating over protocols like HTTP or MQTT. In this chapter, implementation details of created prototype are discussed.

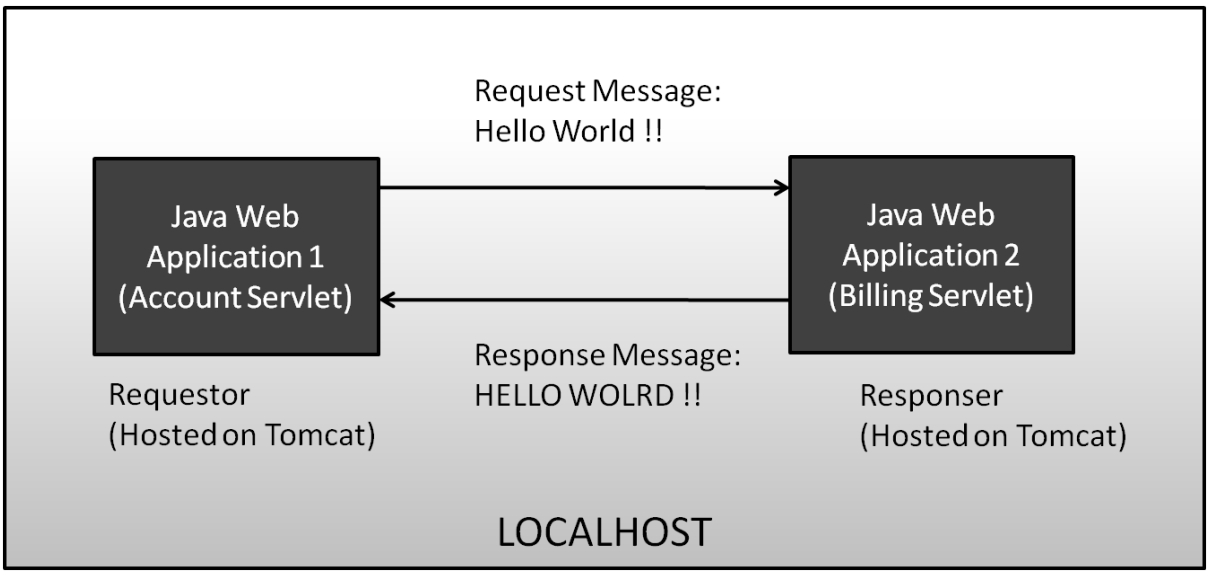

Figure 5.1: Overview of Test Setup

5.1.1 Setup

We have used Java servlet technology for creating test web applications. One web application is requester and another serves as a responser. Figure 5.1 gives an overview of this setup. We have used Eclipse IDE for developing and debugging purpose. These web applications can be hosted on localhost running on Windows or on Amazon's EC2 Linux instance. On both the hosts, we have set up Apache Tomcat Server 7.0.75 for running these web applications. We have used Google Chrome's Advanced REST Client for sending requests to web applications and displaying the response. For working with MQTT protocol, we have used Eclipse Java Paho Client which is an MQTT client library written in Java for developing applications that run on the JVM or other Java compatible platforms like Android [Foua]. For MQTT, we have used Mosquitto public broker but the implementation is independent of any other public MQTT broker.

5.1.2 Implementation Details of Web Application

Since we have used Java servlet technology for building the web applications, both the web applications are separate servlets themselves namely Account.java and Billing.java. Account.java is requester while Billing.java is responser. In the prototype, we have always moved responser web application i.e. Billing servlet.

5.1.2.1 Requestor

Listings 5.1 and 5.2 shows *doGet()* and *doPost()* methods implementation for handling GET and POST method requests from REST Client. The GET method is designed to handle single parameter named *t1*, passed through URL. In Listing 5.1, on line 4 we are storing the parameter value in variable *str1*. Lines 6 to 9 forms the HTTP connection to the responser Biiling and send the data through URL "*http://localhost:8080/WebApp2/Billing?t1=<value-of-str1>*". Lines 11 to 25 captures the response from Billing.java. Finally line 27 displays that response on the REST Client. Similarly, POST method is implemented as shown in Listing 5.2. Lines 7 to 11 forms the HTTP connection to Billing.java through the URL "*http://localhost:8080/WebApp2/Billing*". The data from the REST Client is read and stored in *postbody* variable on lines 13 and 14. On lines 15 and 16, the DataOutput-Stream is opened onto the connection and data is sent to Billing Web application. Lines 19 to 31 captures the response for this sent data and finally line 32 displays that response on REST Client.

5.1.2.2 Responser

Listings 5.3 and 5.4 shows implementation of GET and POST request methods of responser Billing.java. The main purpose of Billing Web application is to capitalize the string data sent by the requester. On line 5 of Listing 5.3 parameter *t1* is extracted from the URL and stored in variable *str1*. On line 6 it is capitalized and sent back as response to requester on lines 8 to 13. The *doPost()* is implemented similarly in Listing 5.4.

Listing 5.1 doGet() of Account.java

```
1  public void doGet(HttpServletRequest req, HttpServletResponse res) throws
       ServletException, IOException
2    {
3    try {
4       String str1 = req.getParameter("t1");
5       //SENDING DATA VIA GET METHOD AND RECEIVING RESPONSE
6       URL url = new URL("http://localhost:8080/WebApp2/Billing" + "?t1=" + str1);
7       HttpURLConnection connection = (HttpURLConnection) url.openConnection();
8       connection.setRequestMethod("GET");
9       connection.setConnectTimeout(0);
10
11      InputStream is = connection.getInputStream();
12      BufferedReader rd = new BufferedReader(new InputStreamReader(is));
13
14      String line;
15      StringBuffer response = new StringBuffer();
16
17      while ((line = rd.readLine()) != null)
18         {
19            response.append(line);
20            response.append('\r');
21         }
22      rd.close();
23
24      res.setContentType("text/html");
25      PrintWriter out = res.getWriter();
26
27      out.println("Response from Billing "+response.toString());
28      out.close();
29      }
30      catch (Exception e)
31      {
32         e.printStackTrace();
33      }
34   }
```

Listing 5.2 doPost() of Account.java

```
1  public void doPost(HttpServletRequest req, HttpServletResponse res) throws
       ServletException, IOException
2     {
3     try {
4        //SENDING DATA VIA POST METHOD
5        res.setContentType("text/html");
6
7        URL url = new URL("http://localhost:8080/WebApp2/Billing");
8        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
9        connection.setRequestMethod("POST");
10       connection.setConnectTimeout(0);
11       connection.setDoOutput(true);
12
13       BufferedReader in=new BufferedReader(req.getReader());
14       String postbody = in.readLine();
15       DataOutputStream wr = new DataOutputStream(connection.getOutputStream());
16       wr.writeBytes(postbody);
17       wr.flush();
18
19       InputStream is = connection.getInputStream();
20       BufferedReader rd = new BufferedReader(new InputStreamReader(is));
21
22       String line;
23       StringBuffer response = new StringBuffer();
24       while ((line = rd.readLine()) != null)
25          {
26             response.append(line);
27             response.append('\r');
28          }
29       wr.close();
30       rd.close();
31       PrintWriter out = res.getWriter();
32       out.println("Response from Billing "+response.toString());
33       }
34       catch (Exception e)
35       {
36          e.printStackTrace();
37       }
38    }
```

Listing 5.3 doGet() of Billing.java

```
1  public void doGet(HttpServletRequest req, HttpServletResponse res) throws
       ServletException, IOException
2     {
3     try{
4        //RECEIVING DATA VIA GET METHOD OF ACCOUNT
5        String str1 = req.getParameter("t1");
6        str1=str1.toUpperCase();
7
8        res.setContentType("text/html");
9        OutputStreamWriter writer = new OutputStreamWriter(res.getOutputStream());
10
11       writer.write(str1);
12       writer.flush();
13       writer.close();
14       }
15       catch (Exception e)
16       {
17          e.printStackTrace();
18       }
19    }
```

Listing 5.4 doPost() of Billing.java

```
1  public void doPost(HttpServletRequest req, HttpServletResponse res) throws
       ServletException, IOException
2     {
3     try{
4        //RECEIVING DATA VIA POST METHOD OF ACCOUNT
5        res.setContentType("text/html");
6
7        BufferedReader in=new BufferedReader(req.getReader());
8        String line = in.readLine();
9        line=line.toUpperCase();
10
11       res.setContentType("text/html");
12       OutputStreamWriter writer = new OutputStreamWriter(res.getOutputStream());
13
14       writer.write(line);
15       writer.flush();
16       writer.close();
17       }
18       catch (Exception e)
19       {
20          e.printStackTrace();
21       }
22    }
```

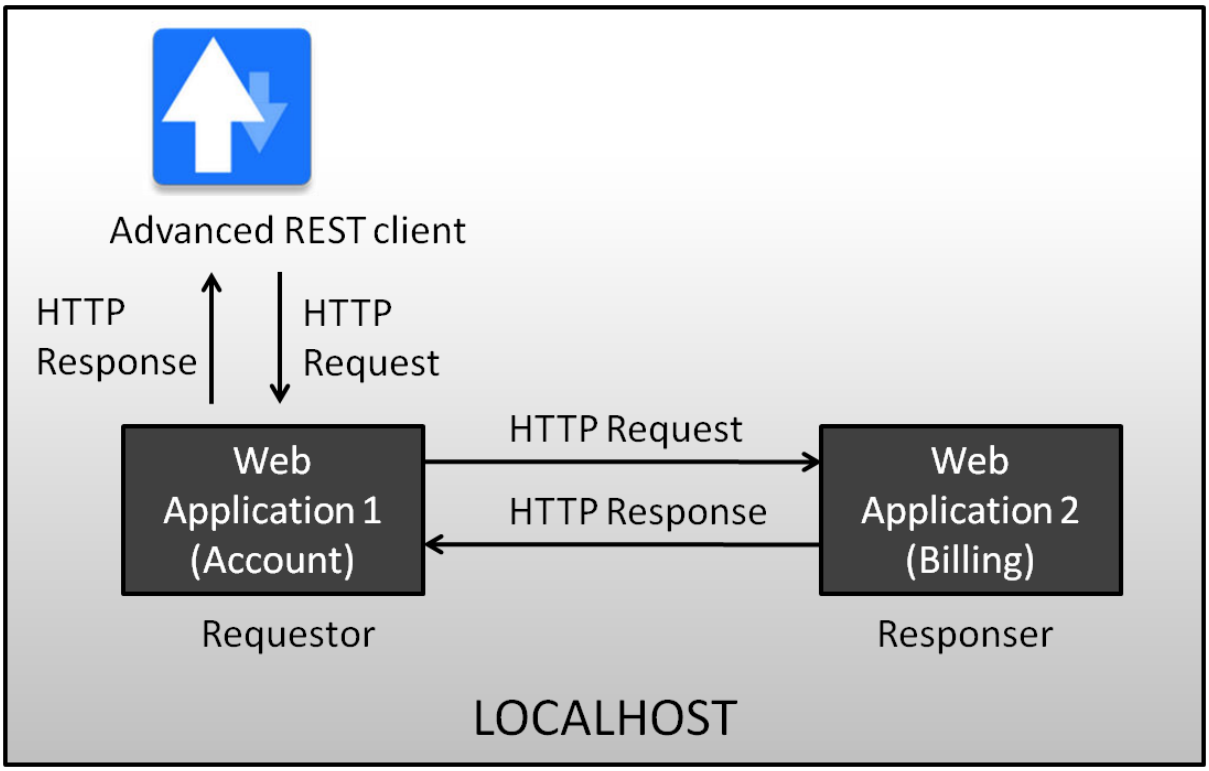

Figure 5.2: Web applications communicating over HTTP before redistribution

5.2 Implementation Details for Scenario 1: HTTP Communication

Figure 5.2 shows setup when two web applications Account and Billing are residing on same host i.e. localhost and Figure 5.3 shows setup when web application Billing is moved to EC2. From Figure 5.3 one can see that proxy for Billing has been introduced by PG on localhost.

5.2.1 Implementation Details of Proxy

The generated code skeleton for the proxy is shown in Listing 5.7. It is a servlet named Proxy.java. This proxy will intercept all the request coming from Account web application. The web.xml file, which is the configuration file for Proxy.java is shown in Listing 5.5. On line 10 of Listing 5.5, we can see that all the requests are accepted by this proxy. Proxy.java will analyze whether the request is of type GET or POST and corresponding code will be executed. On lines 7 and 23 of Listing 5.7, we can see that new HTTP connection is opened to Billing. The URL consists of new host name and new

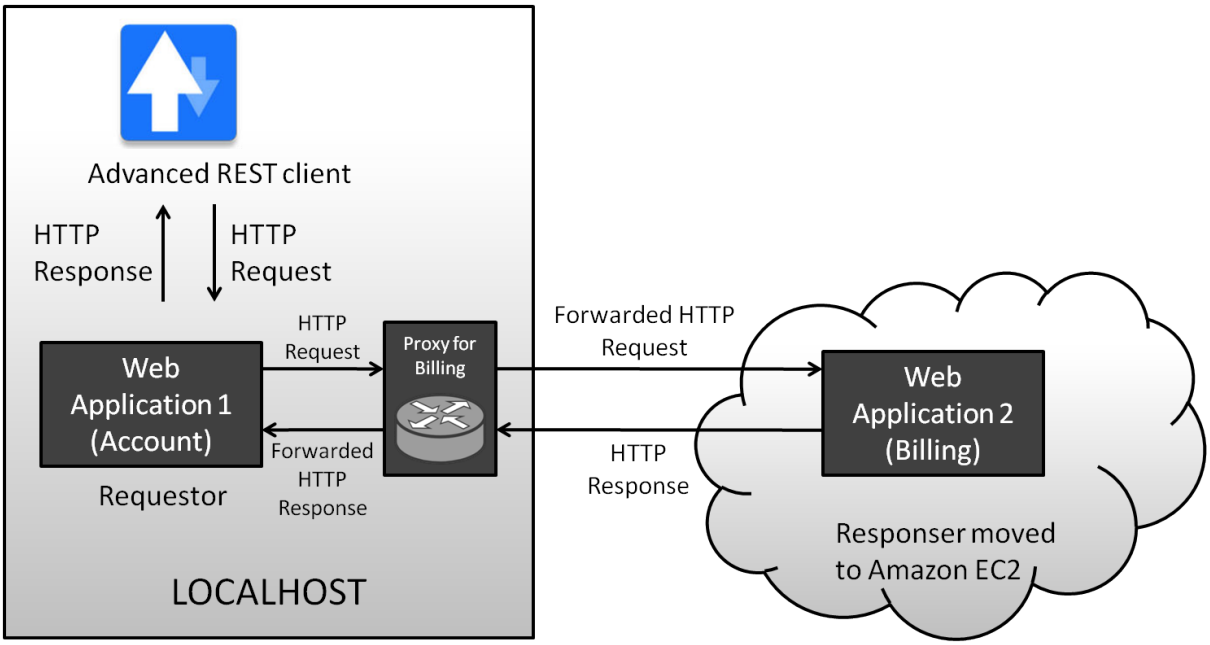

Figure 5.3: Web applications communicating through proxy over HTTP after redistribution

port number which is given as an input to PG. Lines 10 to 16 and 31 to 37 will capture the response from Billing web application and lines 18 to 19 and 39 to 40 will send this captured response back to Account web application. Finally, Account sends the received response back to REST Client.

Listing 5.5 web.xml file for Proxy.java

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app id="WebApp_ID" version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3     <display-name>WebApp2</display-name>
4     <servlet>
5    <servlet-name>proxy</servlet-name>
6    <servlet-class>Proxy</servlet-class>
7  </servlet>
8  <servlet-mapping>
9    <servlet-name>proxy</servlet-name>
10   <url-pattern>/*</url-pattern>
11 </servlet-mapping>
12 </web-app>
```

Listing 5.6 pom.xml file for Proxy.java

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" .......>
2     <dependencies>
3        <dependency>
4           .....
5        </dependency>
6     </dependencies>
7     <build>
8        <finalName>WebApp2</finalName>
9     </build>
10 </project>
```

The proxy code generated out of PG is basically a maven web application project which has pom.xml file shown in Listing 5.6. The project consists of WAR (Web application ARchive) file as an output which is then put into the web apps directory of Apache Tomcat folder on localhost. The name of WAR file is set as "WebApp2" as shown in Listing 5.6 on line 8. The same name is provided to PG as an input.

5.3 Implementation Details for Scenario 2: HTTP to MQTT Communication

In this scenario, the underlying communication protocol is changed to MQTT. The web applications Account and Billing are unchanged with respect to their implementations as shown in Listings 5.1, 5.2, 5.3 and 5.4. Figure 5.4 shows setup when both the web applications are residing on localhost but the underlying communication protocol is changed to MQTT. In this case, we have separated the workspaces of Account and Billing by setting up a Virtual Machine on localhost. The reason for such set up is cleared at the end of this section. Figure 5.5 shows setup when Billing is moved to EC2 instance. Due to change of protocol, proxies consisting of Translator Tx and Adapter Ar are injected by the PG as suggested in the previous chapter. The generated requester proxy is a servlet named RequestorProxy.java and the responser proxy is named as ResponserProxy.java. Both the proxies act as a client to MQTT Mosquitto broker. Requestor proxy subscribes itself to response topic upon initialization while responser proxy subscribes itself to request topic upon initialization. Similar to Listing 5.5, a web.xml file for both the proxies are generated as shown in Listing 5.8 and 5.9 respectively. On line 7 of both the files, we can see that these servlets are loaded on startup, therefore, subscribing to their topics on the start.

Listing 5.7 Proxy.java

```
1  protected void service(HttpServletRequest req, HttpServletResponse res) throws
       ServletException, IOException
2  {
3        String URI=req.getRequestURI();
4        String method = req.getMethod();
5        if (method.equals("GET"))
6        {
7           URL url = new URL("http://52.57.206.31:8080"+URI+"?"+queryparamline);
8           //..
9
10          InputStream is = connection.getInputStream();
11          BufferedReader rd = new BufferedReader(new InputStreamReader(is));
12
13          String line;
14          StringBuffer response = new StringBuffer();
15          while ((line = rd.readLine()) != null)
16             {response.append(line);}
17
18          OutputStreamWriter writer = new OutputStreamWriter(res.getOutputStream());
19          writer.write(response.toString());
20       }
21       if (method.equals("POST"))
22       {
23          URL url = new URL("http://52.57.206.31:8080"+URI);
24          //..
25
26          BufferedReader in=new BufferedReader(req.getReader());
27          String str1 = in.readLine();
28          DataOutputStream wr = new DataOutputStream(connection.getOutputStream());
29          wr.writeBytes(str1);
30
31          InputStream is = connection.getInputStream();
32          BufferedReader rd = new BufferedReader(new InputStreamReader(is));
33
34          String line;
35          StringBuffer response = new StringBuffer();
36          while ((line = rd.readLine()) != null)
37             {response.append(line);}
38
39          OutputStreamWriter writer = new OutputStreamWriter(res.getOutputStream());
40          writer.write(response.toString());
41          }
42       }
43    }
```

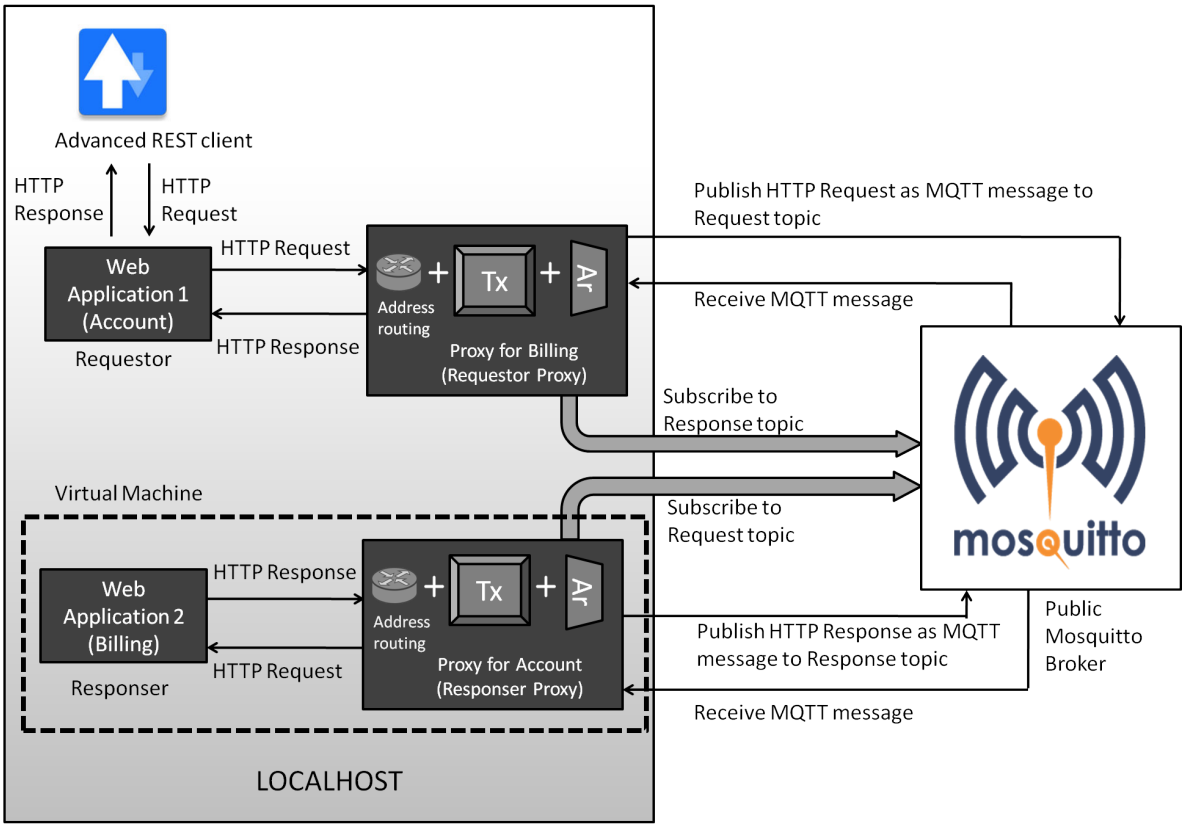

Figure 5.4: Web applications communicating over MQTT before redistribution

Listing 5.8 web.xml file for RequestorProxy.java

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app id="WebApp_ID" version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3      <display-name>Wapp2</display-name>
4      <servlet>
5         <servlet-name>proxy</servlet-name>
6         <servlet-class>com.proxi.RequestorProxy</servlet-class>
7         <load-on-startup>1</load-on-startup>
8      </servlet>
9      <servlet-mapping>
10        <servlet-name>proxy</servlet-name>
11        <url-pattern>/*</url-pattern>
12     </servlet-mapping>
13 </web-app>
```

Listing 5.9 web.xml file for ResponserProxy.java

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app id="WebApp_ID" version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3     <display-name>Wapp1</display-name>
4     <servlet>
5        <servlet-name>proxy</servlet-name>
6        <servlet-class>com.proxi.ResponserProxy</servlet-class>
7        <load-on-startup>1</load-on-startup>
8     </servlet>
9     <servlet-mapping>
10       <servlet-name>proxy</servlet-name>
11       <url-pattern>/*</url-pattern>
12    </servlet-mapping>
13 </web-app>
```

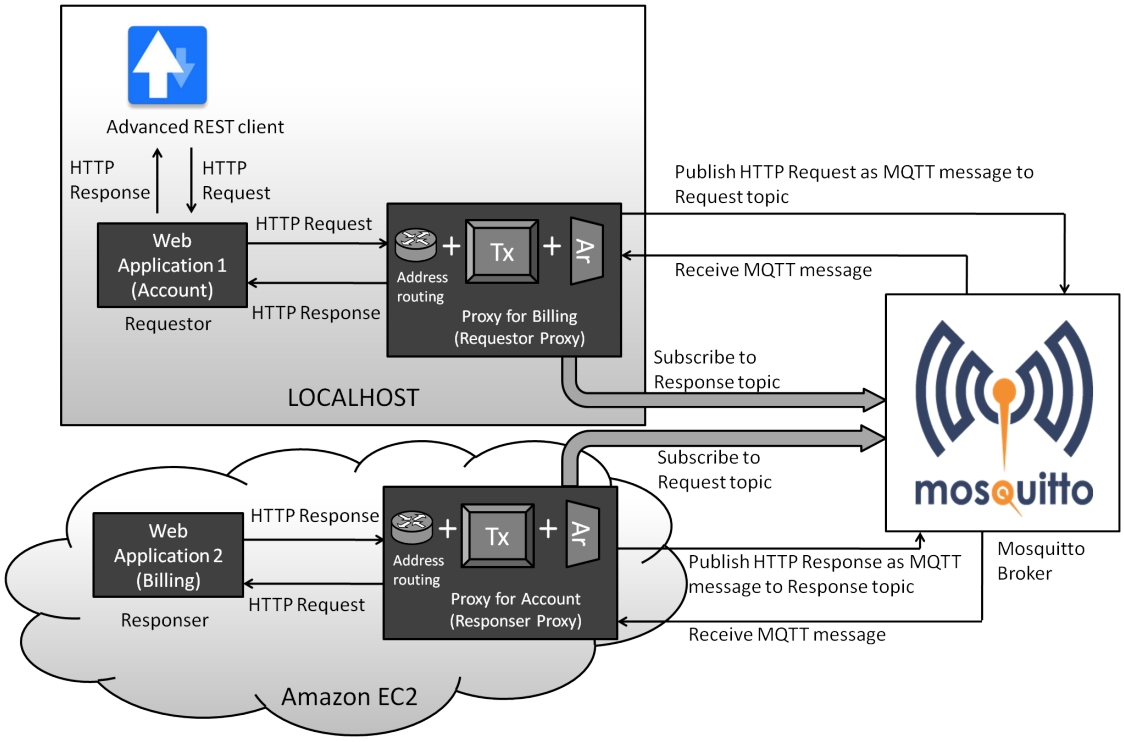

Figure 5.5: Web applications communicating over MQTT after redistribution

5.3.1 Implementation Details of Requestor Proxy

RequestorProxy.java contains *service()* as shown in Listing 5.10 for receiving the requests from Account. If the request is of type GET, then we are forming a messageString of format "method,URI,queryparamline" in which method means GET, URI is part of the URL after hostname and port number and queryparamline contains query parameters passed via URL. Similarly, if the request is of type POST then, messageString is of format "method,URI,body" where body is actual data passed in body of POST request method. Line 20 act as a address routing mechanism in which it is routing the data to broker on port 1883. Line 21 act as an adapter code that connects this proxy to broker. Line 22 and 23 act as a translator code that converts HTTP data to MQTT message type and finally on line 24 MQTT message is published to the broker on "*req_topic*". Lines 26 to 41 belongs to synchronized block of code that waits till it get notification from *messageArrived()*. This synchronized block waits to forward the response to Account until response message is received in *messageArrived()*. Listing 5.12 shows *messageArrived()* that receives the response message from broker. Once the message is received, it notifies the *service()* on line 29 of Listing 5.10. RequestorProxy.java contains *init()* method as shown in Listing 5.11 which serves the functionality of subscribing to response topic named "*res_topic*" upon initialization.

5.3.2 Implementation Details of Responser Proxy

The ResponserProxy.java contains *init()* similar to RequestorProxy.java which subscribes itself to request topic named "*req_topic*" as shown in Listing 5.13 on line 10. ResponserProxy.java has *messageArrived()* shown in Listing 5.14 which includes code for address routing, translator and adapter. The HTTP request in form of MQTT message is captured on line 3. Then the message is analyzed to check whether it is of type GET or POST, according to which the corresponding code is run. If it is of type GET, then it opens new HTTP connection to Billing via constructing the URL by inserting query parameters in it and making the GET request to Billing on line 8. This serves as an address routing mechanism. The host name and the port number are given to PG as input by the user. The response from Billing is then captured between lines 11 and 12. Line 15 appends the method type i.e. GET to the response received. Line 19 act as an adapter code that connects to MQTT broker. This response is then converted to MQTT message type by translator code (lines 21 and 22). Finally, the response message is published to MQTT broker on line 23. Similarly, the POST method requests are handled in lines 26 to 45.

Listing 5.10 service() of RequestorProxy.java

```
1  protected void service(HttpServletRequest req, HttpServletResponse res) throws
       ServletException, IOException
2    {
3          String messageString="";
4          if (method.equals("GET"))
5          {
6             //Read query parameters from GET request URL
7             //.....
8          messageString = method + "," + URI + "," + queryparamline;
9          }
10         else if (method.equals("POST"))
11         {
12            //Read from POST request
13            //.....
14
15            String body = in.readLine();
16            messageString = method + "," + URI + "," + body;
17         }
18
19         //Publish HTTP request as MQTT message to broker
20         client = new MqttClient("tcp://iot.eclipse.org:1883",
               MqttClient.generateClientId());
21         client.connect();
22         MqttMessage message = new MqttMessage();
23         message.setPayload(messageString.getBytes());
24         client.publish("req_topic", message);
25         client.disconnect();
26         synchronized(lock)
27         {
28            //service method has to wait until message is received by messageArrived()
29            lock.wait();
30            try
31            {
32               OutputStreamWriter writer = new OutputStreamWriter(res.getOutputStream());
33               writer.write(receivedMsg);
34               writer.flush();
35               writer.close();
36            }
37            catch (Exception e)
38            {
39               e.printStackTrace();
40            }
41         }
42   }
```

Listing 5.11 init() of RequestorProxy.java

```
1  public void init()
2     {
3     // defining MQTT subscriber for capturing the HTTP response as MQTT message
4        try
5        {
6           System.out.println("SUBSCRIBED TO res_topic");
7           client = new MqttClient("tcp://iot.eclipse.org:1883",
                MqttClient.generateClientId());
8           client.setCallback(new RequestorProxy());
9           client.connect();
10          client.subscribe("res_topic");
11       }
12       catch (MqttException e)
13       {
14          e.printStackTrace();
15       }
16    }
```

Listing 5.12 messageArrived() of RequestorProxy.java

```
1  public void messageArrived(String s, MqttMessage mqttMessage) throws Exception
2     {
3        synchronized(lock)
4        {
5           receivedMsg = new String(mqttMessage.getPayload());
6           //Remove method String from received message
7           receivedMsg=receivedMsg.substring(receivedMsg.lastIndexOf(",")+1,
                receivedMsg.length());
8           System.out.println("Message received: " + receivedMsg);
9           //Notify the service method that message has been received from broker.
10          lock.notify();
11       }
12    }
```

5.3.3 Explanation for Setting Up VM

We need to separate the working space of Account and Billing web applications via setting up a VM because for a setup in which both the web applications are in same Tomcat directory on localhost, Account can simply make HTTP request to Billing over HTTP which is not desirable for this scenario as communication must happen over MQTT protocol.

Listing 5.13 init() of ResponserProxy.java

```
1  public void init()
2     {
3        //defining MQTT subscriber for capturing the HTTP request as MQTT message
4        try
5        {
6           System.out.println("SUBSCRIBED TO req_topic");
7           client = new MqttClient("tcp://iot.eclipse.org:1883",
                MqttClient.generateClientId());
8           client.setCallback( new ResponserProxy() );
9           client.connect();
10          client.subscribe("req_topic");
11       }
12       catch (MqttException e)
13       {
14          e.printStackTrace();
15       }
16    }
```

5.4 Implementation Details of Proxy Generator (PG)

Proxy Generator is a tool developed in Java language which is capable of generating the proxies as described in previous sections. Figure 5.6 shows simple flowchart of how PG works.

5.4.1 Input to PG

In order to generate the proxies, a text file containing the input information is given to PG. The input file should specify certain details depending upon the type of proxy that needs to be generated.

5.4.1.1 Input to PG: HTTP Case

Text file containing following information is given as the input to PG for HTTP case:

- CommunicationProtocol:HTTP
- ResponserWebApplicationName:WebApp2
- Hostname:52.57.206.31
- Portnumber:8080

Listing 5.14 messageArrived() of ResponserProxy.java

```
1  public void messageArrived(String s, MqttMessage mqttMessage) throws Exception
2     {
3        String receivedMsg=new String(mqttMessage.getPayload());
4
5        if(receivedMsg.contains("GET"))
6        {
7           //Send GET request to Billing
8           URL url = new URL("http://52.57.206.31:8080"+URI+"?"+paramline);
9           //...
10
11          //Read the response from Billing
12          //...
13
14          String resMsg=response.toString();
15          resMsg="GET,"+resMsg;
16
17          //Publish response as MQTT message to broker
18          client = new MqttClient("tcp://iot.eclipse.org:1883",
                MqttClient.generateClientId());
19          client.connect();
20
21          MqttMessage message = new MqttMessage();
22          message.setPayload(resMsg.getBytes());
23          client.publish("res_topic", message);
24       }
25
26       if(receivedMsg.contains("POST"))
27       {
28          //Send POST request to Billing
29          URL url = new URL("http://52.57.206.31:8080"+URI);
30          //...
31
32          //Read the response from Billing
33          //...
34
35          String resMsg=response.toString();
36          resMsg="POST,"+resMsg;
37
38          //Publish response as MQTT message to broker
39          client = new MqttClient("tcp://iot.eclipse.org:1883",
                MqttClient.generateClientId());
40          client.connect();
41
42          MqttMessage message = new MqttMessage();
43          message.setPayload(resMsg.getBytes());
44          client.publish("res_topic", message);
45       }
46    }
```

For HTTP case, the text file must have a communication protocol as *HTTP*. The name of Responser Web application is set as *WebApp2* in our implemented scenario. The text file must have the new hostname and port number of responser web application.

5.4.1.2 Input to PG: MQTT Case

Text file containing following information is given as the input to PG for MQTT case:

- CommunicationProtocol:MQTT
- RequestorWebApplicationName:Wapp1
- ResponserWebApplicationName:Wapp2
- brokerId:tcp://iot.eclipse.org:1883
- Hostname:52.57.206.31
- Portnumber:8080

For MQTT case, the text file must have a communication protocol as *MQTT*. The name of Responser Web application and Requestor Web application is set as *WebApp2* and *WebApp1* respectively in our implemented scenario. The text file must have MQTT broker's ID. The text file must also have the new hostname and port number of responser web application.

5.4.2 Output of PG

As stated before the generated proxies are maven web application projects consisting of WAR (Web application ARchive) file. The name of the web application provided at the time of input in the text file will be the name of generated WAR file. These WAR files are to be put into the web apps directory of Tomcat installation folder.

5.4.2.1 Output of PG: HTTP Case

HTTPHandler class of PG is responsible for generating the proxy in case of HTTP protocol. Following is the sequence of steps followed for generating the proxy:

1. Initially it creates a skeleton of maven project of type web application.
2. It creates a servlet called *Proxy.java* within the project. This servlet contains the required functionality of proxy which is address routing mechanism.

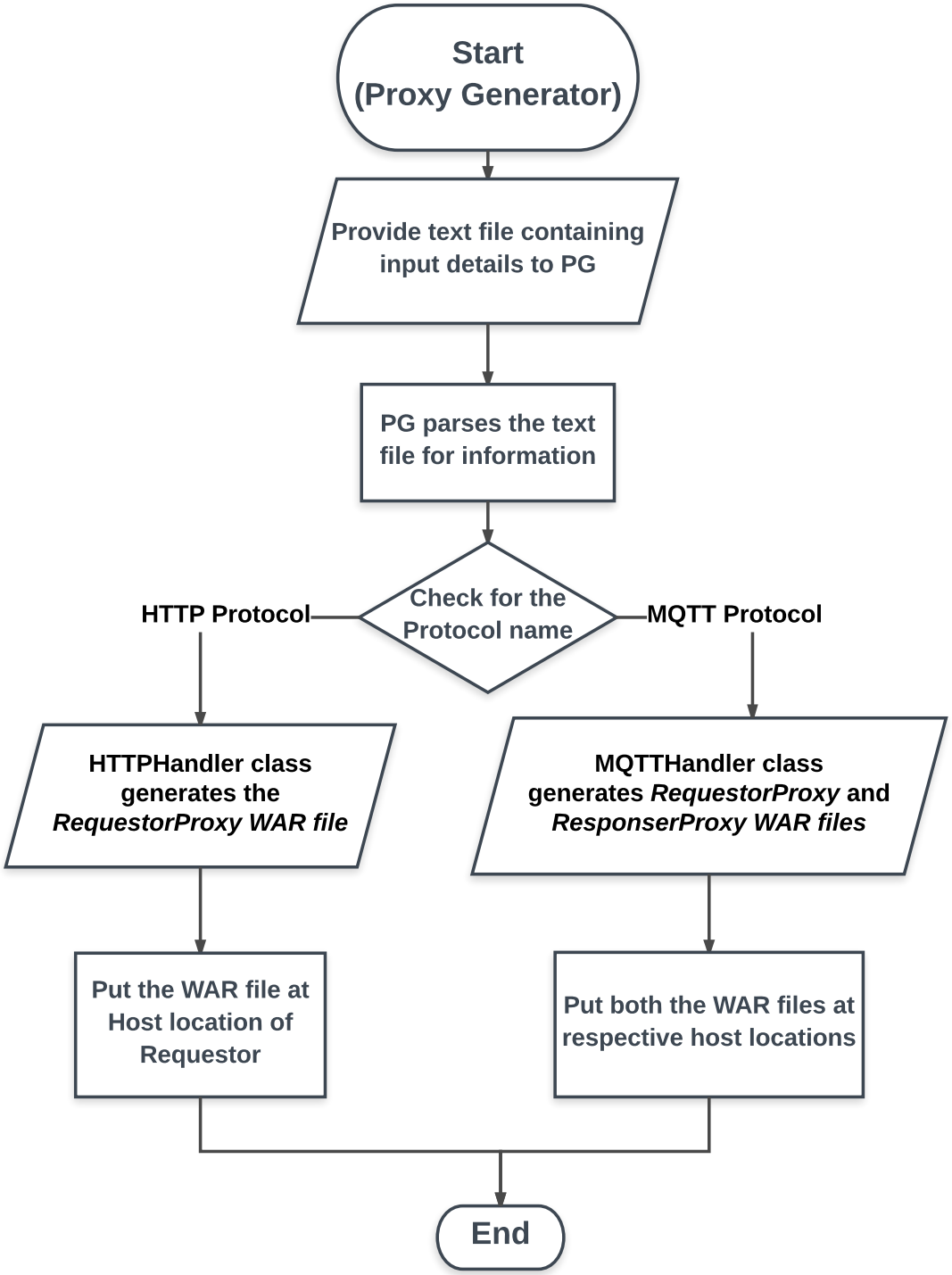

Figure 5.6: Flowchart of PG Working

3. It generates the *web.xml* file also known as deployment descriptor for the web project similar to Listing 5.5.
4. It generates *pom.xml* file required for building maven project as shown in Listing 5.6.
5. Finally, it builds the proxy as a WAR file.

5.4.2.2 Output of PG: MQTT Case

MQTTHandler class of PG is responsible for generating the proxy in case of MQTT protocol. Following is the sequence of steps followed for generating the proxy:

1. Initially it creates a skeleton of maven project of type web application for Requestor Proxy.
2. It creates a servlet called *RequestorProxy.java* within the project. This servlet contains the required functionality of proxy which is address routing mechanism, translation and adapter function.
3. It generates the deployment descriptor *web.xml* for the web project similar to Listing 5.8.
4. It generates *pom.xml* file required for building maven project for Requestor Proxy similar to Listing 5.6.
5. It builds WAR file for the Requestor Proxy.
6. It creates a skeleton of maven project of type web application for Responser Proxy.
7. It creates a servlet called *ResponserProxy.java* within the project. This servlet contains the required functionality of proxy which is address routing mechanism, translation and adapter function.
8. It generates the deployment descriptor *web.xml* for the web project similar to Listing 5.9.
9. It generates *pom.xml* file required for building maven project for Responser Proxy similar to Listing 5.6.
10. Finally, it builds the WAR file for Responser Proxy.

5.5 Summary

In this chapter, the implementation details of created prototype for proof of concept are discussed. The prototype is based on Java servlet technology. At the start of this chapter, some assumptions and test environment for the created prototype are mentioned. The prototype implements the two scenarios stated in Chapter 4. Internal design and working of PG including it's input and output form is also stated in this chapter. Created prototype support only translation between HTTP and MQTT protocol.

6 Validation

In this chapter, the built prototype for the two scenarios as mentioned in the previous chapter is validated. Google Chromes's Advance REST Client is used for making HTTP GET and POST requests to web application Account as shown in Figure 6.1 and Figure 6.2 respectively. Account web application forwards the request to Billing web application directly or via proxy and displays the response.

As we can see from Figure 6.1, a GET request is made to Account web application through URL "*http://localhost:8080/WebApp1/Account?t1=hello_world*". Here, the message *hello_world* is sent as query parameter via GET method. The received response

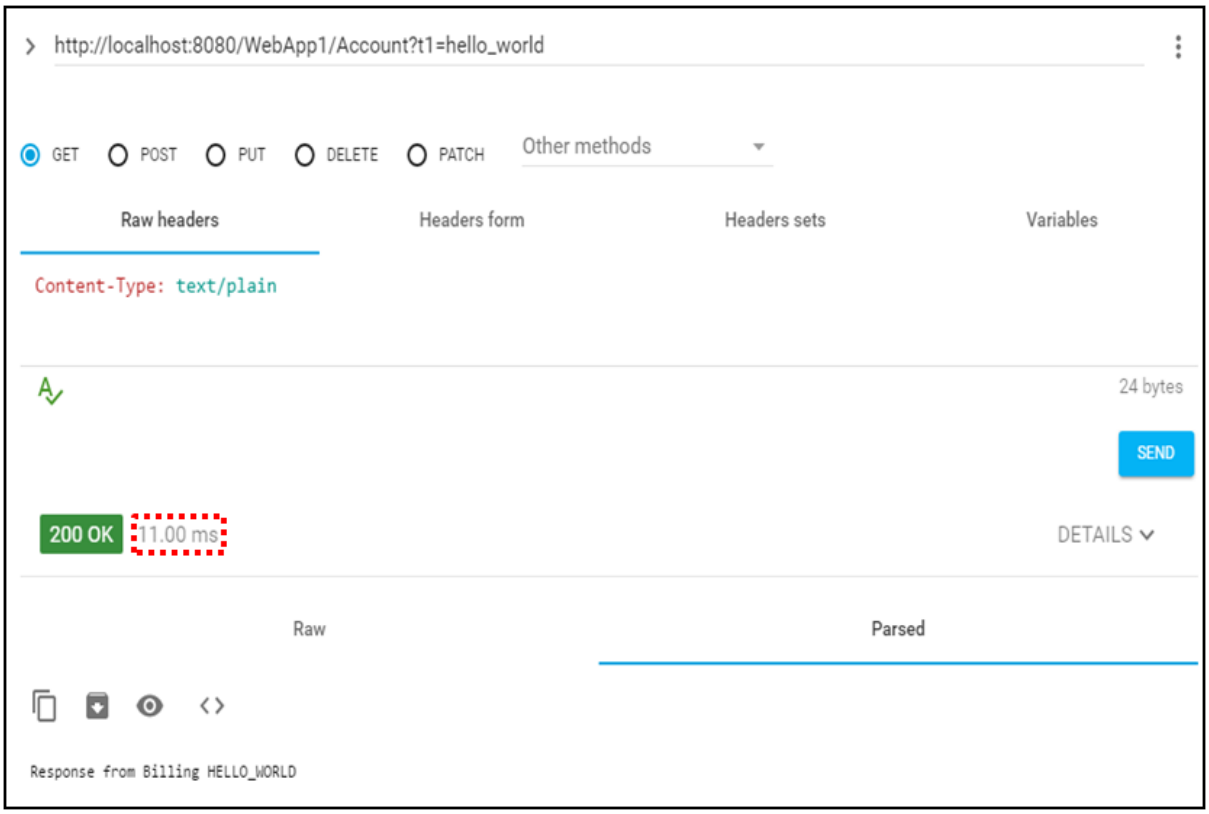

Figure 6.1: HTTP GET request using REST Client

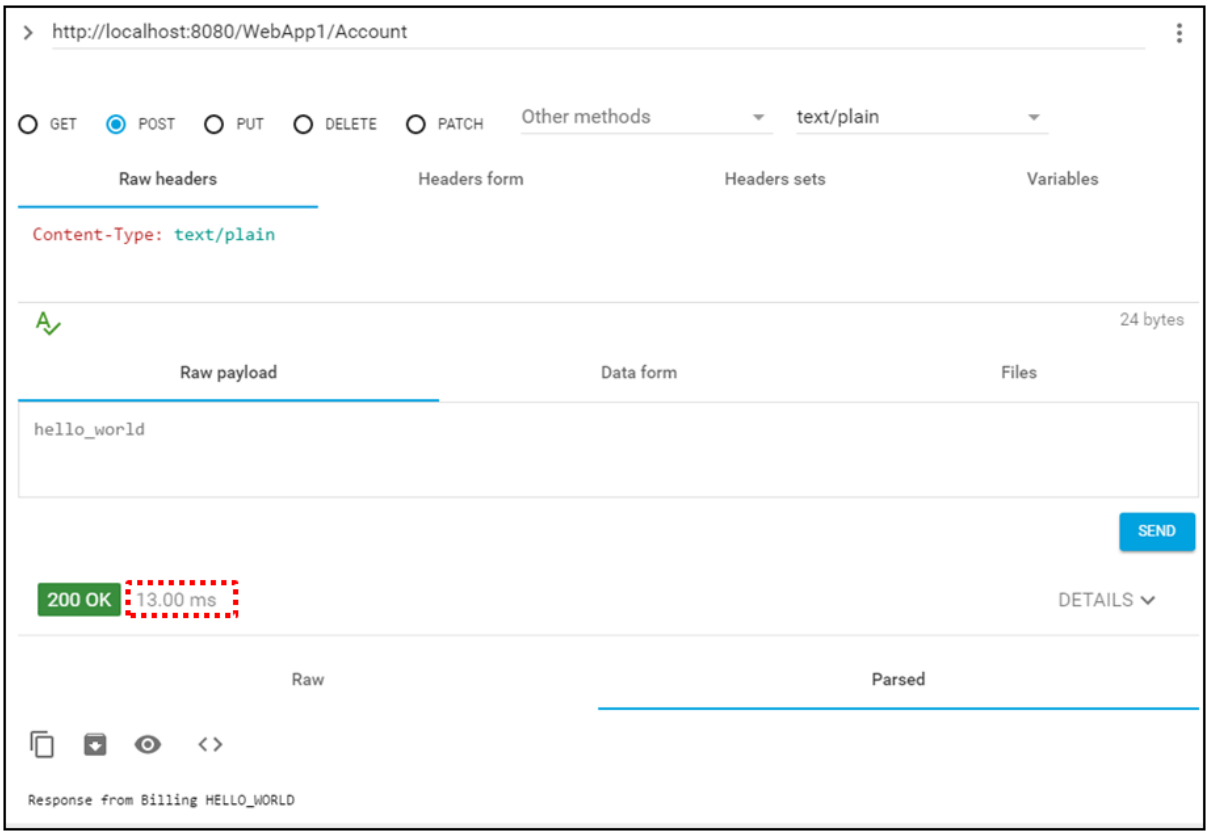

Figure 6.2: HTTP POST request using REST Client

which is a capitalized message *HELLO_WORLD* is displayed at the bottom. The response time is marked in the figure. Figure 6.2 shows POST method request to Account web application via URL "*http://localhost:8080/WebApp1/Account*". The message *hello_world* is sent through POST body. In both the cases 200 OK response was received indicating the successful processing of the requests.

6.1 Validation of Scenario 1: HTTP Case

In order to validate this scenario, firstly both the web applications, Account and Billing are kept on localhost as shown on Figure 5.2 and ten consecutive GET and POST requests are made to Account. The underlying communication protocol was set as HTTP. The time for receiving the response on REST Client is noted. The average time required for sending the message "hello_world" via GET request is 6.6 milliseconds while via POST method is 7.3 milliseconds. After recording the response time for this setup, the Billing web application was redistributed to EC2 instance and the proxy between Account and

Billing was introduced on localhost as shown in Figure 5.3. The response time in this scenario is recorded. The average response time for GET request was 15 milliseconds and for POST request was 25.1 milliseconds. From validation results, it is clear that the response time has been increased due to additional proxy component between the two web application but proxy has successfully forwarded the request to correct destination address of Billing and returned the response.

6.2 Validation of Scenario 2: MQTT Case

In order to validate this scenario, Account web application was hosted on localhost and Billing web application was hosted on VM which was set up on localhost. In order to access VM network, the port 9000 was forwarded from localhost to VM. For this scenario, the underlying communication protocol was set as MQTT and Mosquitto was used as the MQTT broker. Therefore, requestor proxy was injected on localhost while responser proxy was injected on VM as shown in Figure 5.4. The Same HTTP GET and POST method requests were made to Account. The average response time for GET request was noted as 770.2 milliseconds and for POST method it was observed to be 778.9 milliseconds. After recording the response time for this setup, Billing web application was redistributed to EC2 instance and responser proxy on EC2 was introduced as shown in Figure 5.5. In this case, average response time for GET request was noted as 775.5 milliseconds while for POST request it was observed to be 773.5 milliseconds. From validation results, we can conclude that the generated proxy components, translators and adapters work in a desirable way and returns the correct response.

6.3 Summary

In this chapter, the created prototype was validated by measuring the response times in two scenarios under discussion. The mapping between HTTP and MQTT is successfully achieved by the prototype.

7 Conclusion and Future Work

The aim of this thesis was to develop an approach to solve the problem of maintaining the same communication between applications after they have been redistributed. Also, the approach must support the change of communication protocol over which applications are communicating with each other without modification of their source code. In this thesis, the knowledge of some design patterns namely proxy pattern, translator pattern and adapter pattern was studied to solve the problem of redistribution. An approach of generating the proxy having address routing mechanism, translators and adapters was proposed to support maintaining communication between redistributed application along with support for underlying protocol change. An attempt has been made to automatically generate these proxies just by knowing the communication endpoint of redistributed application. A tool called Proxy Generator (PG) was developed to automatically generate these proxy components and inject into the system of communication. This approach primarily requires the information of hostname and port number of the redistributed application and intermediate channel details of new communication protocol, if at all communication protocol is changed. Based on this information which is given as an input to Proxy Generator, it produces the required proxy and injects into the system. The approach was demonstrated by creating the prototype for Proxy Generator. The prototype supported redistribution of web applications that are implemented in Java servlet technology and communicate over HTTP protocol. The prototype supports the conversion from HTTP to MQTT and vice-a-versa, to demonstrate the protocol change scenario.

The biggest advantage of such an approach is that existing application that needs to be redistributed to some other hosts need not be modified or one need not have an access to source code of such application. One needs to know only the host name and port number i.e. communication endpoint of redistributed application. Also suggested approach is generic in the sense that it should be able to translate from one communication protocol to other, thereby applications those are developed that rely on single communication protocol can be adapted to communicate if underlying protocol changes later in the future.

7.1 Future Work

Currently, the implemented prototype supports web applications developed in Java servlet technology but there are numerous technologies and programming languages that are used to develop applications. Therefore, the prototype can be extended to support several technologies and can be made generic in a true sense. Similarly, in the case of different communication protocol other than HTTP and MQTT, support can be added to the implemented Proxy Generator tool.

In this thesis, we have considered only single instance of redistributed application but in practice, there can be multiple instances of single application. For example, as MQTT protocol follows publish-subscribe model, it is quite possible to have multiple subscribers for a single topic. In that case, Proxy Generator might need more input information in terms of a number of instances of moved application. According to this information, one proxy component can be generated per instance that act as a subscriber to the request topic.

The thesis work focuses only on client-server architecture pattern. In practice, there is three tier architecture pattern involving database as the third tier. The approach can be extended in future to focus on this third tier as well.

Bibliography

[ABLS13] V. Andrikopoulos, T. Binz, F. Leymann, S. Strauch. “How to adapt applications for the Cloud environment.” In: *Computing* 95.6 (June 2013), pp. 493–535. ISSN: 1436-5057. DOI: 10.1007/s00607-012-0248-2. URL: https://doi.org/10.1007/s00607-012-0248-2 (cit. on p. 18).

[Ara07] K. Arai. *Distributed application layer protocol converter for communications network*. US Patent 7,280,559. Oct. 2007. URL: https://www.google.com/patents/US7280559 (cit. on p. 35).

[AYD+03] S. Ali, P. Yared, B. Daniels, R. Goldberg, Y. Kamen. *Transparent injection of intelligent proxies into existing distributed applications*. US Patent App. 09/997,927. June 2003. URL: https://www.google.com/patents/US20030105882 (cit. on p. 34).

[Bea] V. Beal. URL: http://www.webopedia.com/TERM/P/proxy_server.html (cit. on p. 20).

[Car13] J. L. Carlson. *Redis in Action*. Greenwich, CT, USA: Manning Publications Co., 2013. ISBN: 1617290858, 9781617290855 (cit. on p. 37).

[CBVC14] M. Collina, M. Bartolucci, A. Vanelli-Coralli, G. E. Corazza. “Internet of Things application layer protocol analysis over error and delay prone links.” In: *Advanced Satellite Multimedia Systems Conference and the 13th Signal Processing for Space Communications Workshop (ASMS/SPSC), 2014 7th*. IEEE. 2014, pp. 398–404 (cit. on p. 38).

[CCV12] M. Collina, G. E. Corazza, A. Vanelli-Coralli. “Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST.” In: *2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications - (PIMRC)*. Sept. 2012, pp. 36–41. DOI: 10.1109/PIMRC.2012.6362813 (cit. on pp. 36–38).

[com] mosquitto community. *Mosquitto Broker*. URL: https://mosquitto.org/ (cit. on pp. 27, 38).

[Cop14] J. O. Coplien. *Software Patterns*. 2014. URL: http://hillside.net/patterns/50-patterns-library/patterns/222-design-pattern-definition (cit. on p. 19).

[Cor96] M. Corporation. *DCOM Technical Overview*. Technical Report. Redmond, WA: Microsoft Corporation, Nov. 1996. URL: http://msdn2.microsoft.com/en-us/library/ms809340(d=printer).aspx (cit. on p. 33).

[Fie99] R. Fielding. 1999. URL: https://www.ietf.org/rfc/rfc2616.txt (cit. on p. 23).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Publishing Company, Incorporated, 2014. ISBN: 3709115671, 9783709115671 (cit. on p. 19).

[For02] B. A. Forouzan. *TCP/IP Protocol Suite*. 2nd ed. New York, NY, USA: McGraw-Hill, Inc., 2002. ISBN: 0071199624 (cit. on p. 22).

[Foua] E. Foundation. URL: https://eclipse.org/paho/clients/java/ (cit. on p. 52).

[Foub] T. A. S. Foundation. URL: http://camel.apache.org/ (cit. on p. 40).

[Fou13] E. Foundation. *Ponte Eclipse Project*. Dec. 2013. URL: http://www.eclipse.org/ponte/ (cit. on pp. 38, 39).

[FT00] R. T. Fielding, R. N. Taylor. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000 (cit. on p. 26).

[GAWM14] S. Gómez Sáez, V. Andrikopoulos, F. Wessling, C. C. Marquezan. "Cloud Adaptation and Application (Re-)Distribution: Bridging the Two Perspectives." In: *Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOCW), 2014 IEEE 18th International*. 2014, pp. 163–172. DOI: 10.1109/edocw.2014.33 (cit. on p. 18).

[Gro01] W. Grosso. *Java RMI*. Ed. by R. Eckstein. 1st. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2001. ISBN: 1565924525 (cit. on p. 33).

[Hiv17] HiveMQ. 2017. URL: http://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment (cit. on p. 27).

[Hoh16] G. Hohpe. *Enterprise Integration Patterns*. 2016. URL: http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageTranslator.html (cit. on pp. 21, 22).

[HW03] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683 (cit. on pp. 15, 19, 21, 22, 40).

[IBM] IBM. *Really Small Message Broker*. URL: http://ibm.co/GQ7vwr (cit. on p. 38).

[Jus07] A. Just. *Distributed application proxy generator*. US Patent 7,171,672. Jan. 2007. URL: https://www.google.com/patents/US7171672 (cit. on pp. 32, 33).

[KOY98] N. Kimura, T. Onodera, N. Yokoshi. *System and method for converting communication protocols*. US Patent 5,778,189. July 1998. URL: https://www.google.com/patents/US5778189 (cit. on p. 36).

[Ley09] F. Leymann. "Cloud Computing: The Next Revolution in IT." In: *Photogrammetric Week '09*. Wichmann Verlag, 2009, pp. 3–12 (cit. on p. 15).

[Lig17] R. A. Light. "Mosquitto: server and client implementation of the MQTT protocol." In: *The Journal of Open Source Software* 2.13 (May 2017). DOI: 10.21105/joss.00265. URL: https://doi.org/10.21105/joss.00265 (cit. on p. 27).

[Mul] I. MuleSoft. URL: https://www.mulesoft.com/platform/soa/mule-esb-open-source-esb (cit. on p. 40).

[Net17] M. D. Network. 2017. URL: https://msdn.microsoft.com/en-us/library/aa267045(v=vs.60).aspx (cit. on p. 15).

[OAS] OASIS. *MQTT*. URL: http://mqtt.org/ (cit. on p. 27).

[RBM01] W. A. Ruh, W. J. Brown, F. X. Maginnis. *Enterprise Application Integration: A Wiley Tech Brief*. New York, NY, USA: John Wiley & Sons, Inc., 2001. ISBN: 1590615441 (cit. on p. 39).

[SALS14] S. G. Sáez, V. Andrikopoulos, F. Leymann, S. Strauch. "Design Support for Performance Aware Dynamic Application (Re-)Distribution in the Cloud." English. In: *IEEE Transactions on Service Computing* (Dec. 2014), pp. 1–14 (cit. on p. 18).

[Sha86] M. Shapiro. "Structure and Encapsulation in Distributed Systems: the Proxy Principle." In: *Int. Conf. on Distr. Comp. Sys. (ICDCS)*. Int. Conf. on Distr. Comp. Sys. (ICDCS). IEEE. Cambridge, MA, USA, United States, 1986, pp. 198–204. URL: https://hal.inria.fr/inria-00444651 (cit. on pp. 31, 32).

[SHB14] Z. Shelby, K. Hartke, C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014. DOI: 10.17487/RFC7252. URL: https://rfc-editor.org/rfc/rfc7252.txt (cit. on p. 38).

[Sof] P. Software. URL: https://projects.spring.io/spring-integration/ (cit. on p. 40).

[TS06] A. S. Tanenbaum, M. v. Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN: 0132392275 (cit. on p. 17).

[VD98] A. Vogel, K. Duddy. *Java Programming with CORBA: Advanced Techniques for Building Distributed Applications*. 2nd ed. New York: Wiley, 1998. ISBN: 978-0-471-24765-4 (cit. on p. 33).

[WBB+13] J. Wettinger, M. Behrendt, T. Binz, U. Breitenbücher, G. Breiter, F. Leymann, S. Moser, I. Schwertle, T. Spatzier. "Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA." In: *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013, 8-10 May 2013, Aachen, Germany*. SciTePress, 2013, pp. 437–446 (cit. on p. 15).

[YS16] T. Yokotani, Y. Sasaki. "Comparison with HTTP and MQTT on required network resources for IoT." In: *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*. Sept. 2016, pp. 1–6. DOI: 10.1109/ICCEREC.2016.7814989 (cit. on p. 27).

[ZBL17] M. Zimmermann, U. Breitenbücher, F. Leymann. "A TOSCA-based Programming Model for Interacting Components of Automatically Deployed Cloud and IoT Applications." In: *Proceedings of the 19th International Conference on Enterprise Information Systems*. SciTePress, 2017 (cit. on p. 15).

[Zim88] H. Zimmermann. "Innovations in Internetworking." In: ed. by C. Partridge. Norwood, MA, USA: Artech House, Inc., 1988. Chap. OSI Reference Model&Mdash;The ISO Model of Architecture for Open Systems Interconnection, pp. 2–9. ISBN: 0-89006-337-0. URL: http://dl.acm.org/citation.cfm?id=59309.59310 (cit. on p. 22).

All links were last followed on August 25, 2017.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature