Utilization of Jyaguchi Architecture for development of Jini Based Service Cloud

Bishnu Prasad Gautam, Shree Krishna Shrestha and Dambar Raj Paudel

● **Abstract**

本論文では、分散アプリケーションを開発するために不可欠なアーキテクチャースタイルを提案し、さらに次世代サービス cloud の構築を強化するためにこのスタイルを使用可能とする方法を提示した。この取り組みは、ドメイン固有のアーキテクチャースタイルではなく、ハイブリッド型のアーキテクチャースタイルを利用するという発想から始まった。本研究で採用したアーキテクチャースタイルは、サービス及び空間ベースのハイブリッド型のアーキテクチャモデルである。研究成果として、既存のアーキテクチャースタイルを分析し、ハイブリッド型のアーキテクチャースタイルに基づいた Jyaguchi アプリケーションの開発に成功した。

● **Key words**

Jyaguchi Architecture
Service Clouds
Universal Browser

1. Introduction

Computing resources have passed the series of evolution which started from a tiny program executing in the single computer to the large virtual world so as to forming the cloud computing resources as of today. This evolution of computing infrastructure implies that the increasing tendency of resource utilization often change the generally accepted notion of resource sharing and resource allocation. Cloud computing, a new software development paradigm for solving complex and large-scale problems, is getting diverse attention from varied fields of science and technology recently. Though computational cloud like services widely known as Grid [2] services are already being used to solve large-scale problems in science and engineering, most of them have focused on defining low-level services.

In Software Architectural front, the development history has started from 1940s with no architectural concept, if not, with very minimal architectural concept mainly targeted to solve the numerical problem of military project and the solutions implemented at that time were relatively simple compared to the current large-scale systems. Solution in those earlier days with less implication of architecture was possible as the components involved in earlier software solutions were minimal. However, software of today needs more attention while developing large scale application. This research defines an architectural style of Jyaguchi [1] to deepen the understanding of distributed application that can be extended up to the architecture for cloud services and demonstrates how this style can be used to enhance the architectural design of simple distributed application. A detailed literature review about existing architectural style was done during the whole research period and then an application based on JINI infrastructure was developed in order demonstrate the architectural style for the service clouds.

1.2 Problem Domain and Research Motivation

Good distributed application architecture [4] must consider lots of elements, components, connectors and other parameters of the system that directly or indirectly affect in the realization of the system. These elements are required to be analyzed and be sketched so as to depict the solution before implementation leading to the best design decision in order to reduce the total cost of the system finally. This research is motivated by the desire to understand and evaluate the architectural style of distributed application thereby demonstrating how such a style in relation with its architectural properties and constraint affect the realization of the Network-based and domain specific application. This paper focuses in the architectural style of distributed application as we realized that there have been very few research to objectively evaluate the architecture of distributed application.

1.3 Solution Approach

In order to perceive the essence of architectural styles adopted in the distributed application and understand its inherent problems, this research analyzes and evaluates the prevalent styles from an architectural perspective. For this purpose, we started to analyze the widely used architectural style[4] in the field of distributed application and further deepen our understandings of internal architecture of those applications.

In terms of evaluating hybrid architecture adopted in Jyaguchi, [1] we have developed some services which were developed emphasizing on the scalability of component and the reduction of dependencies among them. Furthermore, it maintains the principle of encapsulating legacy system by providing the simple methodology of interfacing the underlying software components and the way of enhancing them to be the well defined service. This well defined service is required to retain and be within development principles which were practiced during the development of Jyaguchi.

2. Jyaguchi Architectural Style

2.1 Client With State Full Server

Our first constraint of our system was adding client-server style into the architecture. Applying client-server style, we could separate the service provider and service consumer component. Our other constraint is to maintain the state of service in order to administer the service consumption by the client as our services will be published in the registry for the usage of human client. Developers are required to apply client server model thereby retaining the service state while publishing the service in Jyaguchi service repository.

2.2 Uniform Interface

The most important feature of the Jyaguchi architecture is that our service interface must maintain uniformity for each service implemented. The uniformity implies that each interface has equal number of methods for the given service. These interfaces are designed for large grain object transfer over network. Uniformity provides certain merits in the architecture in production level of the software, one of which is that by fixing the numbers of methods and the name of service, productivity will increase during implementation.

This means that the way the client interacts with each service is the same across all services which lead to increase architectural ability of creating reusable code for service consumption. Further, such reusable artifacts need a relatively less time from the developer perspective. We also set other constraint in our programming style for interface, which are discussed in the Jyaguchi implementation section in the coming chapter.

2.3 Spaced Based and Self Sufficiency

Further improvement to our architecture can be achieved by appending space based architectural node [6] or element in our hybrid architecture style. Spaced based style supports architecture to be composed of shared memory space by providing self-sufficient node which increases the scalability of the system. Further, we virtualize these layers by grouping the application logic and business logic together into a single computational unit that is supported with space based architecture thereby virtualizing the underlying tires. Scaling can be achieved by running multiple instances of such units on multiple machines.

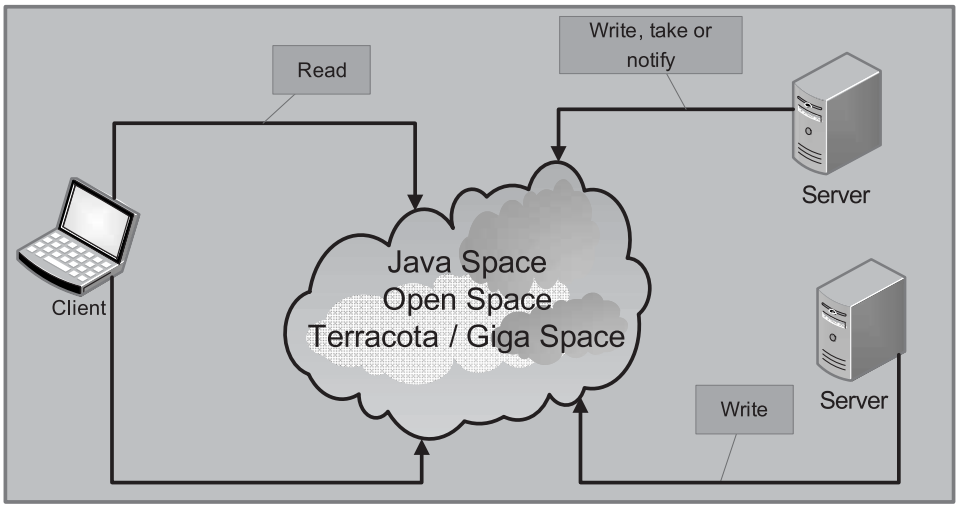

Figure 1: Space Based Model

2.4 Dynamic Discovery: Knowledge Of Service Residency is Not Required

Another constraint in the architecture is to manipulate our service so that discovery process can occur dynamically. Dynamic discovery in protocol level implies that discovery process either follow a broadcast or multicast protocol. Multicast discovery which is based on UDP multicast involves communication with a subnet or multicast group, in order to locate a lookup service. This enables the dynamic discovery property of the service and hence the knowledge of the location of the service is not required.

2.5 Replication and Loose Coupling

Multicast discovery completely removes the dependency on any pre-informed physical location information encoded in a service description (as in WSDL), and any single point of failure issues (as with UDDI). It is trivial, even encouraged, to have multiple service registrars on the network. Services automatically register with multiple registrars, and clients can easily find and query multiple registrars.

2.6 Code on Demand (COD)

Code on demand is the core style applied in Jyaguchi architectural style. It allows the services running in server side which are able to dynamically be downloaded in the client computer and executed in client machine. This style reduces the code required for client as the client might not need full implementation of the program and those codes are served on demand basis from the server. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility.

2.7 Relation Tree of Jyaguchi Style

A complete architecture of Jyaguchi consists of a set of coordinated properties and constraints impacted in the final architectural artifact. The whole tree shown above depicts the relation between these constrains and show

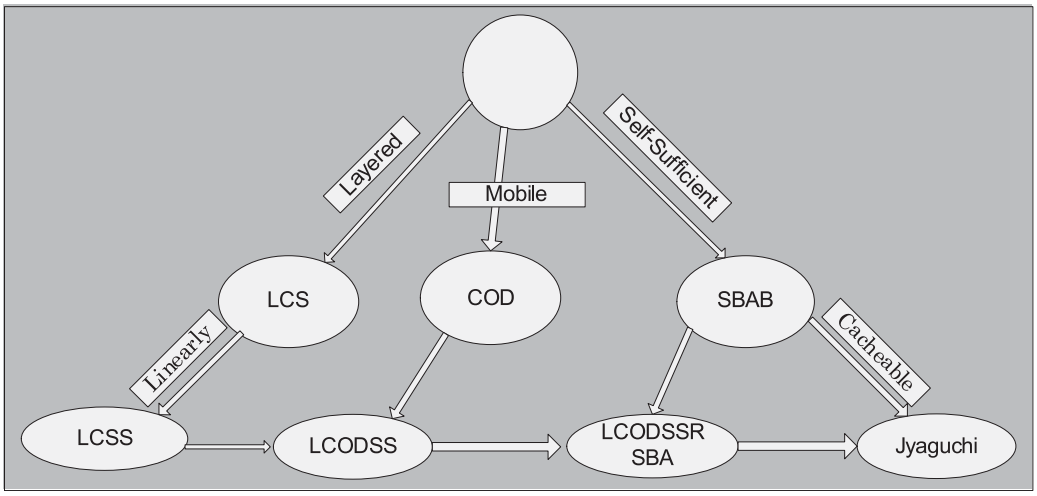

Figure 2: Style relation tree

how the additional constraint affect the relation of the architecture thereby forming a new style. This addition of the style finally would be able to exploit the best architectural properties and result to the hybrid architecture.

3 Designing Jyaguchi

The most significant motivation for Jyaguchi architecture is guided by building block approach, where a set of solutions are implemented as required, from low profile initial capabilities to high profile so as to meet the properties of Jyaguchi architecture. Jyaguchi does not dictate the prior knowledge of this need. However, the ultimate goal of virtualization and resource sharing is always considered and hence does not mis-guide the service development process during the implementation. Jyaguchi cloud assumes that the adaptability, flexibility and robustness are required in the architecture.

3.1 Jyaguchi Architectural Elements

The primary architectural elements in Jyaguchi are Services, Repository and Clients. Services are software resources which are the entities contained in the service repositories provided by Jyaguchi infrastructure. Jyaguchi assumes that the services deployed in the service repository must be available to the service consumer either through uni-cast or through multicast protocol.

3.1.1 Service Repository or Container

Service repository in Jyaguchi architecture has one or two way relationships to other resources that publish its service for the service consumer. The one way relationship of the service is much similar with pure client-server model. In pure client-server model, there are clear distinction between client and server however in the case of Jyaguchi architecture this distinction is minimalistic or almost null as the client also could serve a particular service so as to maintain the two way relationship. Services developed on the basis of Jyaguchi architectural style can be published in the service repository. We choose to publish in the Jini Look Up service as service repository of Jyaguchi to utilize uni-cast, broadcast and multicast protocol provided by Jini Lookup service.

3.1.2 Services

A general concept of service in network application is defined as an entity that serves some sort of job requested by the client. In order to fulfill the task requested by the client, the server side program is required to implement certain operations. These operations are either implemented within the single program or in the multiple programs as per the nature of implemented task.

Services in the context of Jyaguchi refers to as an entity that can be used by the client of Jyaguchi that serves some sort of operation such as calculation, storage or so on for the particular purpose within the software system. Jyaguchi recognizes service which has specific objective in order to fulfill certain responsibility. Thus, services are always waiting to get request from the client in order to fulfill those responsibilities that are expected in the software system.

3.2 Foundation Services

3.2.1 Web Server Unit and Codebase

During the research, we have implemented Jyaguchi services in Java and utilized the features provided by Jini infrastructure to develop model application. Service built upon Jini infrastructure has a proxy component which is exported to look up service and will finally be transferred to the client. The full footprint of the service of course stays in server side; however, the copy of the object is necessary to be downloaded to the client. How would be this achieved? The copy of service will be serialized to the client and finally reconstructed in the client side however this is impossible without the class file. Those class files required for client are necessary to be visible from client side too; otherwise the client program will be unable to utilize the service. These implementation of the classes can be exported either using by HTTP or FTP. However, the service provider must indicate the location of the class file using codebase property. The philosophy behind Java codebase is that when the sender of an object serializes that object for another machine it annotates the serialized stream of bytes in codebase. This will tell the receiver where the implementation of this object can be found. The serialization [5] mechanism facilitates the translation of object state data to bytes and vice versa. Serialized objects do not contain the definitions of the associated classes and interfaces, and thus, when de-serializing an object, it is necessary to either have the relevant type definitions preloaded, or to know the location from which to load the classes and have the capability to do dynamic class loading [10]. This is the place where the code-base plays a significant role.

In Java platform, JVM looks the class at the path set in the CLASSPATH environment which directs the system class loader to load the class from the indicated path. However once the class loader fetch the information of codebase it will load the class from the path given in the code-base.

3.2.2 Database Unit

Database server is another foundational service for the Jyaguchi services built during the research. In fact data base server plays a vital role for the management of user information and to track how long the service was consumed by the client. Jyaguchi services are built by differentiating client, service and data access layer. All

sort of data such as business data, users related data are retain in the database which are accessed only through data base related packages. These packages are accessed only by server classes. This allows client program not to know the location of their data sources. By using this principle, we can re-align the database by changing schema without affecting application. Our architectural style suggests that the client never fire SQL queries directly to the database, however, it can send requests to services if such requirement arises. The service receives the request from a client and sends a message to the middle tier. The middle tier will send an SQL call on behalf of server to the database. This approach means the client does not need to manipulate the database layer directly which will further increase the decoupling of the different tiers federating the overall architecture.

3.3 Client

This element in Jyaguchi application directly involved with end user. In order to utilize the service in Jyagchi style, service providers are required to provide a connector interface. This connector interface works as a protocol for the communication between services and clients. The client component in Jyaguchi architecture presents an abstract interface which is for component communication, enhancing simplicity by providing separation between client and server. This interface in XML based web service architecture are called Web Service Description Language (WSDL) document.

4. Methodology and Implementation of Jyaguchi

4.1 Design Pattern and Remote Service Invocation

The importance of architectural pattern in the case of Jyaguchi architecture is vital. The architectural pattern is relatively simple that we followed Service Oriented Architecture for the entire system. The pattern applied during the development follows the normal course of evolution starts at a service's inception, during which a service emerges as a concept, continues through its analysis and design phases on the basis of architectural constraint discussed above, and finally evolves as a solution service that is executable within Jyaguchi space. This pattern is followed for all created services.

This research was also motivated by programming pattern and therefore programming pattern was applied in the cloud service development during the research. Discussing the importance of design pattern is out of scope of this paper; however, we would like to note the facade pattern with some modification that was adopted during the development. We would like to call this pattern still as facade because the resulted pattern maintains the goal of simplicity. The pattern is adopted during the design of service program for which we have reduced the number of methods in interface in order to minimize the complexity of interface. Jyaguchi service followed facade pattern by implementing the interface which contains single method and all the complex operation are implemented by the Facade class. Facade pattern does not dictate to contain just a single method in the interface rather it was decided during the development of Jyaguchi in order to simplify the client code while making the remote service invocation.

-Write interface

-Write Façade Class

-Write Client Code

In the case of Jyaguchi Service let's describe the facade pattern by using Calculator service example. This Calculator has numbers of operations such as addition, subtraction, divide, multiply and providing the graph of given equation. All of these services are packed in a GUI thus playing a facade for all other modules. This facade class is called by an aggregator class which implements the interface. This aggregator class had wider visibility among the subsystems and all the required parts for the services are aggregated finally with this class. The good part of merging facade pattern with single method interface pattern is that the client program does not require calling of the operations so as to reduce the number of call in the network thus reducing the server overload. Let discuss in the following sections how the facade, aggregator class and single method interface pattern is adopted in Jyaguchi service.

4.2 Writing an interface (Single Method Interface)

```
public interface CalculatorServerInterface extends CommonContainerInterface {
public void Login (String serviceName, int userId) throws RemoteException;
}
```

The above code snippet shows that there is just a single method inside the interface. However, please note that this interface extends the CommonContainerInterface. We have reduced the number of methods by shifting the required methods to the CommonContainerInterface. We can add the number of methods as per the requirement inside CommonContainerInterface instead of CalcultorServerInterface in order to simplify the interface as stated in our earlier discussion. We extended this pattern to be the basic requirement in the case of developing Jyaguchi service and this is very essential part to develop the service in this pattern. Jyaguchi services are invoked and downloaded to the client machine after calling this single method defined in the interface. Using a Singly Method Pattern in an interface has following rewards for the developers.

1. It reduces the unnecessary invocations from the client program. This means overhead of server machine can be reduced. As Jyaguchi services are located in different machine, it is suggested to reduce the number of call from client machine.
2. Reducing the number of methods in interface leads to decrease the dependency between client and server and hence they can be kept loosely coupled. As a result, client program can be implemented easily.
3. It reduces the role of server and provides the mechanism to implement fat or smart proxy

4.3 Writing an Implementation Class as Facade of the subclasses

Following the facade pattern lead us that when clients have to interact with several classes or subsystems, we can generally provide a single class as an aggregator class which acts as a buffer between the user class and the sub-systems which provide expected result. In the case of above discussion, all the operations such as divide,

multiply, add and packing all GUI required for the class are implemented in the subclasses. In our case Calculator class has implemented all sorts of operation. The client of this facade class is not required to know all the operations of the sub-classes. In our case all facade classes are appended with "impl". The class postfix with "impl" not only implements the interface of the facades but also provide the operation required for client program by utilizing the functionalities of other classes.

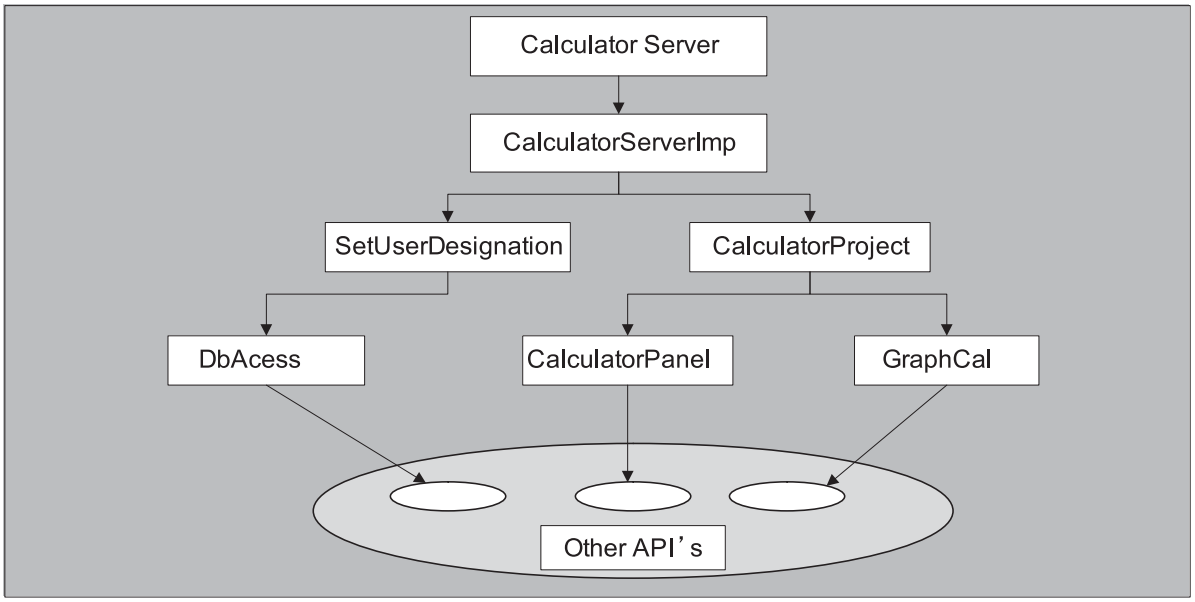

Figure 3: Class Hierarchy based on facade pattern

4.4 Programming Model -Remote Service Invocation

The application built on Jyaguchi architecture exploits the maximum usage of Jini programming model which utilize most of the features of Java RMI (Remote Method Invocation). Our prototype application which is based on this model also exploited most of the lower level features of RMI. But there are few differences between the approach of RMI and RSI. RSI is a model or a new architectural pattern of the services that participate in the Jyaguchi cloud environments. We can see the difference between RMI and RSI in the diagram below. In the RMI system, interface must be pre-wired in the client program. This made the RMI system tightly coupled, which is obviously not so favorable in distributed applications. Whereas, in RSI system this requirement is avoided, instead, the client program will require some information about the server program but those information are not necessarily be pre-wired. That information about the server program will be transferred later on in semi-permeable way. This trick made the RSI system more loosely coupled [7].

RSI programming model suggest the procedure mentioned below during the service development.

1. Write a single method in the interface
2. Follow the facade pattern while defining the services
3. Invoke the method of interface from client side in order to get the service

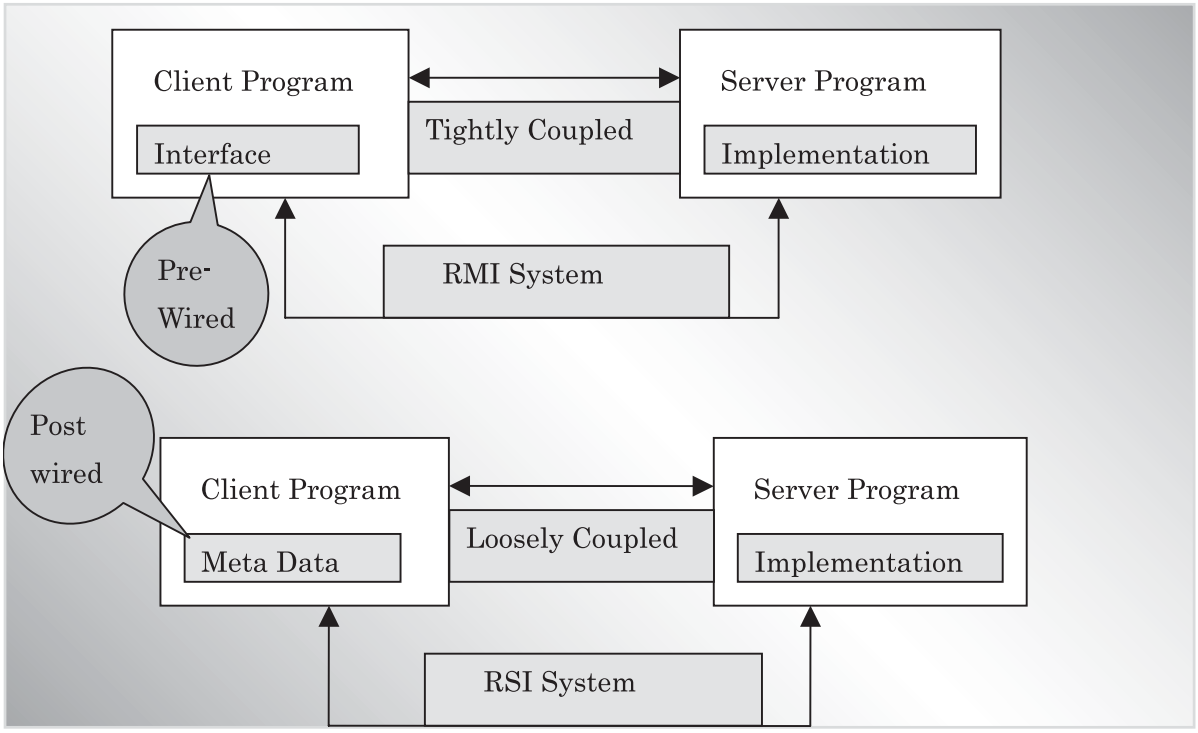

Figure 4: Remote Service Invocation

Resolving a service request within a single method has number of advantages in distributed application. The most notable advantage is reduction of complexity of the service invocation. As we witnessed in SOAP where documents have their own protocol of procedure calls and which are layered over HTTP. This increases the complexity of SOAP procedure calls compared to simple document fetches using GET and POST [9]. In the case of Web service too, interface can be written in WSDL document however to generate simple interface, it requires complex definition of XML elements. We have achieved simplicity by reducing the number of methods in the interface.

4.5 System Scenario

The overall flow of the service invocation and utilization process is depicted in the following sequence diagram. The minor details of sequences are intentionally omitted in order to simply the overall scenario. In order to make Jyaguchi services act as utility services, the service provider needs to create an instance of the exportable service object, register this and keep the lease alive. To fulfill our architectural requirement and implement our services, we have utilized the API provided by Jini 2.1.

The sequence diagram starts from the service registration process. All services implemented in our applications follow the same process. Service registration process follows two simple phase, first of which starts from finding the service registry. Once the service registry is found, the second phase of service registration starts for which it not only registers the service but also adds it to a resource manager, to keep the lease alive and be able to run for long time. Resource manager is a LeaseRenewalManager class provided by Jini API that renews and monitors the entire leasing activities which runs its own threads to keep re-registering the leases. In

case the service is terminated by service provider lease will fail to renew and the exported service will be discarded from the service registry.

Whereas in the client side, our system requires proper authentication of the user in order to properly consume the resource. Once the user entered his/her information and upon successful authentication, client will be able to send a request to search the service registry in the network. To find the registry, we have tested both unicast and multicast discovery. In the both processes, client gets the proxy of service registrar by which client will be able to find the service that it wants to consume finally.

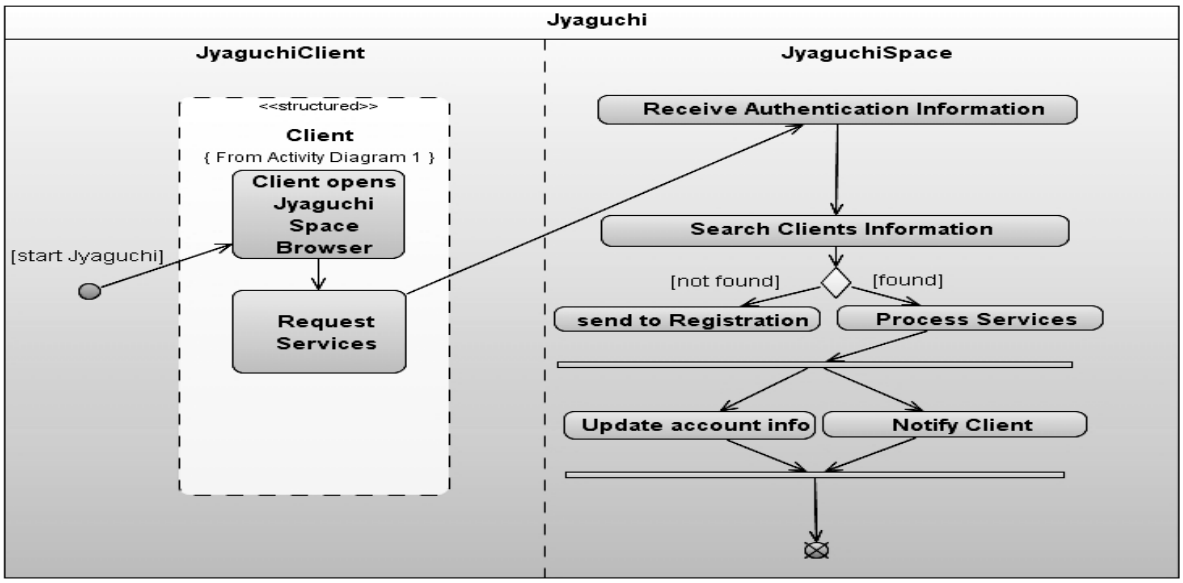

Figure 5: Activity Diagram

As shown in the above activity digram, the user of Jyaguchi can utilize service once the service discovery process is successful. Client will receive the service and be able to utilize those services. A brief introduction of the services is given in the next sections.

4.6 Experimented Services and Post Notification

The overall service usage scenario has been described in the earlier sections by using sequence and activity diagram. In the following diagram, the real scenario of the service usage has been depicted. As shown in the left most part, we can see the service registry which store the services registered in it. A list of service registry in the network can be depicted within this tool thereby creating a virtual network of services. In the very next part, the registered services are listed. These services are enabled with post notify capabilities which implies that new services can be seen in the browser and dynamically appended in the list.

Post notification and dynamic deliver of the service omit the requirement of complex procedure of software installation for the client. Post notification implies that services deployed in the registry are notified to the end-user without human interference. Thus, end-users do not need to worry about new installation and update of the software.

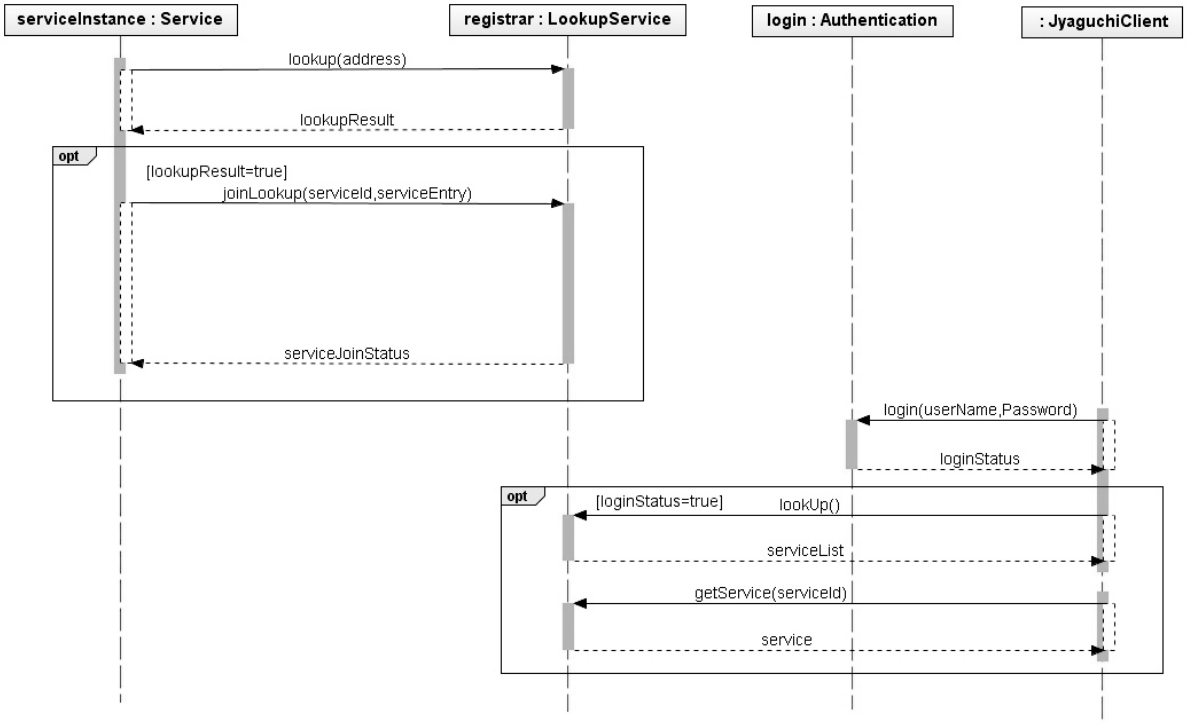

Figure 6: Simplified sequence diagram of Jyaguchi application

Service providers can update the services in the server without interfering the usage of the service by the client so that client can utilize the updated service immediately.

4.7 Universal Browser and Single Point of Connection

We witnessed that internet is becoming an essential information resource for all walks of people. Yet millions of users are unable to access this important resource due to inaccessible web sites, documents and the resources on the web. There are large numbers of services or resources in the web if allowed to use with simplicity, or from the same web browser that will add another level of satisfaction to the end-users. Our concept of universal browser provides a single point of connectedness with the resource which is distributed across the web.

Single point of connectedness refers from the end-user perspective that end-users do not like to face the hurdles of software installation, update or removal. These sorts of actions require a bit of technical knowledge and relatively higher level of literacy. However, if the services are provided without the requirement of such technical knowledge, the usability of web resources increase more than ever.

Universal browser shown in the figure 8 provides the single point of connectedness to the end-users. We have tested Editor, Web Browser, Presentation and other few numbers of services which can be viewed in this browser and provides the single point of connectedness. End-users are not required to install these services as these services can be provided upon double click and all are connected in the same browser which we called universal browser.

Figure 7: Jyaguchi Application Interface

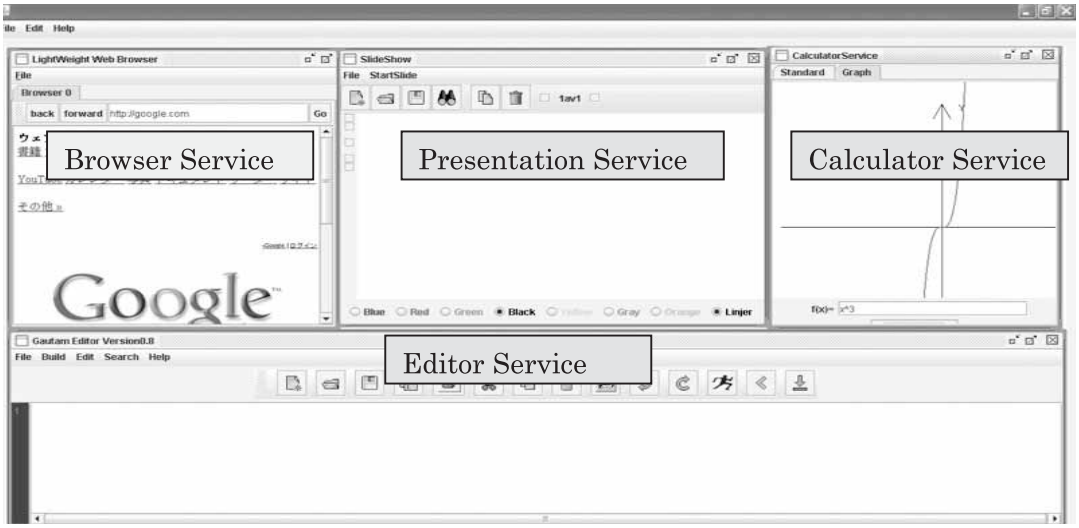

Figure 8: Prototype of Universal Browser

5. Evaluation, Conclusion and Future Works

We have developed the Jyaguchi application after analyzing the architectural constraint of distributed application and induce hybrid style into the architecture. We have used reverse engineering technique by utilizing the reflection API of Java in order to extract architectural information from executing services in the network. It is quite possible to visualize this information at run time and regenerate the process view of the underlying system after implementation. This technique can give an insight even to generate the overall architectural representation and evaluate the architectural representation of the system incurred at design phase.

This in fact led to better understanding of the role of architectures in the application life-cycle process.

5.1 Conclusion

One of the prominent objectives of building services in Jyaguchi architecture is retaining the quality of SOA [3] and exploits benefits from hybrid nature of architecture through utilizing the key architectural principles which finally may lead to improvements in building of distributed applications. Distributed application include various styles of architecture, however, Jyaguchi approach of implementation elaborates the ones which are considered essential for distributed services so as to virtualize the services and render the services to the end-users in a minimal human intervention. This type of delivering of the services can offer the dynamic resource allocation to the end-user thereby providing the services in pay per use basis as depicted in earlier sections.

IT enterprises will have no attention shall the services are not modeled as per the business need of the Enterprise. We witnessed that in a world new technology surface and fade away without impacting the society. We believe that Jyaguchi architecture can be applied to develop services that can be extended to be the services of cloud computing technology.

5.2 Future Work

There are two main directions for future work in Jyaguchi application, one of which is at the application level and the other is at the architecture level. While evaluating our service prototype, we identified a number of issues for further research. For instance, we have to limit the size of service and identify what sort of service suit to utilize the services provided by this system. We also would like to develop a workload portioning packages for Jyaguchi services with more precise partitioning function. We also would like to work on the security layer of the application. The current implementation utilizes the security provided by java security packages and also the policy based security feature by which we can restrict the unauthorized service to participate in the cloud. However, in order to incorporate the architecture into the business, substantial work to support the security of the system is vital.

● References

[1] Bishnu Prasad Gautam, *An Architectural Model for Time Based Resource Utilization and Optimized Resource Allocation in a Jini Based Service Cloud.* Master Thesis, Shinshu University, Nagano, Japan

[2] Ian Foster, *There's Grid in them thar Clouds,* http://ianfoster.typepad.com/blog/2008/01/theres-grid-in.html

[3] Rob High, Jr. et. all IBM's SOA Foundation, *An Architectural Introduction and Overview, Version 1.0*

[4] Roy Thomas Fielding, Representational State Transfer http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[5] Tero Hasu, *Implementing Jini Servers without Object Serialization Support,* http://www.tml.tkk.fi/Studies/T-110.557/2002/-papers/tero_hasu.pdf

[6] Nati Shalom, GigaSpaces White Paper, *The Scalability Revolution:From Dead End to Open Road An SBA Concept Paper*, 2007

[7] Bishnu Prasad Gautam, *JINI Based Service Grid: Remote Service Invocation Model for Time Based Resource Allocation*,

Undergraduate Thesis, Wakkanai Hokusei Gakuen University, Japan

●英文タイトル

Utilization of Jyaguchi Architecture for development of Jini Based Service Cloud

●要約

This thesis proposes an architectural style for the development of distributed application that can be extended up to the architecture for cloud services and demonstrates how this style can be used to enhance the architectural design of next generation service cloud. The motivating factor to undertake such initiatives is associated with utilizing the hybrid architectural style rather than sticking in domain specific architectural style. The architectural style that we adopted is the hybrid of service and spaced based architectural model. A detailed literature review about existing architectural style was done during the whole research period and then an application based on hybrid architectural style was developed and named as Jyaguchi [1].

●キーワード

Jyaguchi Architecture
Service Clouds
Universal Browser