April 2012

Resource Location Transparency in Clouds

Khanh-Nhan P. Nguyen
Worcester Polytechnic Institute

Latiff Zaaliembike Seruwagi
Worcester Polytechnic Institute

Linhai Zhu
Worcester Polytechnic Institute

Project Number: MXC-0360

Resource Location Transparency in Clouds

A Major Qualifying Project Report:

Submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

Latiff Seruwagi

Linhai Zhu

Khanh-Nhan Nguyen

Date: April 25, 2012

Professor Michael J. Ciaraldi, Major Advisor

Sponsored by Oracle

Abstract

Cloud systems have become a ubiquitous way of harnessing the power of large numbers of networked computers. One of the important functionalities of these systems is the ability to access a resource seamlessly regardless of where in the cloud it is stored. In this MQP, we provide a design that ensures transparency of a resource using a message-oriented cloud system. We then created a simple implementation of this design that demonstrates how information is transmitted over the cloud.

Acknowledgements

We would like to thank Oracle for sponsoring our project and giving us advice and professor Ciaraldi for advising us.

Special thanks go to Mike Voorhis of the Computer Science department for setting up our virtual machine testbed and offering us his expertise.

Table of Contents

Table of Figures

Executive Summary

Cloud systems have become a widely-used technology for distributed services. Their capabilities vary from remote file access and editing to performing complex calculations and simulations. The configuration of cloud services is often complex and a resource may be provided by multiple load-balanced servers. Even with such complexity, cloud systems make resources seemingly transparent to the client. Because of the cloud's often dynamic nature, it is important that a client is able to access a given service transparently, without relying on a static list of resource-providing remote servers. The goal of our project is to achieve resource location transparency. Specifically, we will provide a seamless way of maintaining access to a resource even when the location of a resource changes.

We looked at several existing systems whose designs could be useful in solving the problem. These were Oracle Coherence, Peer-To-Peer systems in general and message-oriented middle-ware. We also considered dynamic DNS as an initial solution. Of these systems we eventually chose to use message-oriented middle-ware because it could more easily be used to implement a resource-transparent design.

Peer-to-peer systems allow direct connections between hosts in a network. In a peer-to-peer system every host in the network behaves as both a client and a server. This means that every host provides both a service and can use the services of other hosts. The equality of all hosts in the network gives these systems their strength but makes them inappropriate for cloud systems which depend on a client-server relationship. Oracle Coherence uses a peer-to-peer

system of servers to create clusters that can service a client's resource requests. However, it is built to provide file-related services.

Our solution is designed to use message-oriented middle-ware (MOM) to send data between a client and server. The server provides resources that the client can access. In this sense the system consisting of the server and the message-oriented middle-ware behaves as a cloud to the client, which can transparently request from it whichever resource it requires. The message-oriented middle-ware consists of a set of brokers configured in what is called high-availability cluster. A high-availability cluster provides a way of ensuring redundancy in message storage in such a way that any broker in the cluster can replace any other in case of broker failure.

We designed a solution consisting of a virtual client, a virtual server, and an MOM cluster. The virtual client maintains a direct connection to a resource. The virtual server maintains a direct connection to the client. Both of these programs maintain a connection to the MOM cluster and use it to send messages that might have otherwise been sent directly from the client to a server that provides the resource. This configuration provides transparency by creating a constant intermediary (the cluster) which can be connected to by the virtual server and the virtual client. The redundancy in the cluster provides reliable communication. The use of virtual servers and clients provides a general framework that can operate with whatever resource that is being used.

We then created a simple implementation of this solution as a proof of concept. The implementation was programmed to handle web page requests. It was able to handle multiple client requests as well as network interface failures. This basic implementation can be built upon to provide for greater flexibility in terms of the resources handled by the virtual client as well as

the ability of the virtual client to handle more complicated resource configurations which may require balancing connections to different servers.

1. Introduction

1.1. Directions From Oracle

Oracle's instruction: "Develop a solution that supports resource location transparency in non-static environments like Cloud - an example of this would be a database that is launched on a virtual machine on DHCP address - A solution needs to be developed that would support mobility of an entity in a dynamic environment such as virtualization in such a way that client processes can react to changes in their network location transparently."

1.2. Usefulness of Transparent Clouds

The cloud is often marketed as a general solution to consolidate resources while reducing complexity. With the "cloud" one need not be concerned with some of the specifics of resource location and management. The cloud takes the management work away from the user while still providing robust access to resources. Like many other computing concepts, the cloud serves as a convenient abstraction that simplifies computer or resource usage. The benefit from this is modularization and specialization of tasks; and therefore greater efficiency among individual components.

For example, in a file-based cloud the users need not know the exact location of the requested file. Requesting from the cloud through what is usually an Application Programming Interface would reduce the software complexity as well as the amount of data tracking and synchronization. With traditional file hosting, users would often have their own partitions or even servers, which both limits scalability and expansion and leads to unused space and

redundant infrastructure. In contrast to traditional network file systems, clouds eliminate the need to remember and update a large list of network shares.

However, the same cannot be said about resources themselves. Although cloud systems are optimized for lots of moving resource consumers, the same cannot be said about servers holding the resources. Not all cloud systems are designed to withstand the arbitrary loss or relocation of servers without interruption of transactions. That is, although clouds are universally adept at linking an end user to the right resource, clouds are not nearly robust at handling servers that move or disconnect intentionally or unintentionally without the end user noticing.

Transparency, in this case, is defined as the ability for the cloud to handle both network events on the user side as well as the server side seamlessly.

1.3. Problem statement

Design a general system where resources can automatically be routed to the right users. This system should be capable of handling network interruptions, reconnections, and relocation of resources without user maintenance.

1.4. Project Roadmap

We planned to gather general information about distributed systems in order to understand existing solutions and their handling of resource discovery and routing. Armed with that information we would then create a solution that incorporates useful designs from existing solutions as well as our own fixes in order to fulfill the request by Oracle and the problem statement.

1.5. Project members' responsibilities

Linhai Zhu: RabbitMQ setup and evaluation, existing distributed systems research

Latiff Seruwagi: setup of middle-ware, solution design, concept implementation

Khanh Nhan Nguyen: sketch the solution design, evaluate and implement the concept

All: Final report, solution design

2. Background

2.1. Distributed Systems

A distributed computing system is a network of autonomous computers used for a single purpose or goal. The use of these systems is necessitated by the performance limitations of a single machine. And as large-scale computing rose in demand, decentralization became indispensable. However, decentralization raises an entirely new set of issues that need to be tackled. As clusters of computer increase in number and size, management becomes an issue. Manual control of each individual computer is both inefficient and impractical given the complexity of large systems. Self-managing systems were created as a response, and are providing the prospect of computer clusters that can handle operations independently.

2.2. Resource Transparency

An important concept of modern computing is abstraction, where the complex details are hidden away leaving behind a clear and simple interface. One example of this is the Flat Memory Model, where memory is offered to the programmer as a single array of bytes while hiding the complex paging and caching system. Even more recently, a new generation of garbage-collection languages and environments seek to free the programmer from manual memory management and allow them to focus on algorithms.

In this project, resource transparency is going to abstract away resource locations, especially in the case of relocation and reconnection. This way, the end user or software would no longer have to be concerned with keeping track of and updating locations and backup locations of all the resources in what could potentially be an extremely large cloud.

2.3. CAP theorem

The CAP theorem, created by Eric Brewer of the University of California at Berkeley, describes a limitation of distributed computing systems where a maximum of two out of three major desired qualities can exist at the same time. These qualities are Consistency, Availability, and Partition tolerance, hence the acronym "CAP".

Consistency in a distributed environment means the consistency of data or state across autonomous machines within said environment. This requires timely synchronization of data and states between the machines within the distributed environment. For example, in a large cluster of replicated databases, consistency means that any edit in one instance of the database would be quickly propagated across all machines. The term "quickly" isn't precisely defined, but we can assume that the system as a whole manages changes to the resources so that different users need not worry about out of date information or colliding updates.

Availability here is not the constant availability of the resource (although that would be desirable as well), but rather the system's ability to return feedback to any requests. Timely feedback is also part of availability since extraordinarily long timeouts are indistinguishable from lost messages and waiting indefinitely for a message while keeping a state for all of them is inefficient. Timely responses, then, would be our interpretation of availability, and this is not just our opinion. Time to live of messages or packet communications are common in distributed communications.

Partition tolerance is perhaps the most important aspect of our solution. The term here means that a system would be able to tolerate network events and even partial failure without

catastrophic breakdown. This is the most important quality of our solution, where the ability to reroute communications in the event of relocation or partial failure is part of Oracle's explicit instruction. (Sathupadi, 2011) (Seth Gilbert)

Overall, our solution's relationship with the CAP theorem will be one of tradeoffs. Partition tolerance will be a hard requirement as it is the core of our project while consistency and availability are swapped in different areas of the cloud. Although this will not provide a perfect solution, (and according to CAP, it does not exist) we will try to compensate as much as possible through the solution design.

3. Existing Solutions

3.1. Oracle Coherence

Coherence is Oracle's proprietary solution for high-availability file management and caching. It is designed as a scalable peer-to-peer system that can tolerate and recover from failure as well as changes in the network. This system can handle the addition and loss of peers and take care of data replication automatically. Some additional features include data backup, cluster analysis, data querying, and change notification. However, it only handles data and no other resources. (Oracle, 2011) In short: it is a file cloud with replication as well as high-availability features. Overall, Coherence reflects the solution to the problem proposed, albeit for files only.

Although implementation details aren't publicly available, it did give us an impression on some of Oracle's offerings in the cloud area as well as what features are likely to be required. It also provided precedence for enterprise cloud solutions that are based on peer-to-peer systems.

3.2. Peer-to-peer (P2P) File Sharing

3.2.1. About P2P

One well-known type of distributed system is Peer-to-Peer (P2P) file sharing, more specifically, the BitTorrent protocol. Although the protocol focuses heavily on P2P file transfers, the subject of interest is the method of which the distributed task is accomplished. We are interested in how a new computer joins the P2P network and how it handles continuous resource transfer in a network with high churn.

3.2.2. How BitTorrent works

In a BitTorrent system, there are two major entities: peers and trackers. A peer is simply an end user that receives and sends parts of files to other peers. Since most file-sharers are users on home networks and generally shut down the file sharing application after downloading a file, there is high turnover. A tracker does not partake in file exchange, and is generally hosted on dedicated servers. Unlike peers, trackers keep track of what peers are currently within the peer-to-peer network and are updated constantly for consistency.

Before downloading, a peer must know which trackers are keeping track of the network holding the relevant files. Where the peer gets the tracker address is something that can vary greatly and is not relevant here. However, the tracker is almost always a server that exhibits high persistence and availability. A user can use another tracker in case of tracker failure. A newly connected peer can start downloading from any of the active peers in the network. If a connection becomes unusable, the peer will simply connect to other computers without user intervention, giving the impression of a continuous transfer. A resource can also be relocated in this system easily. Upon relocation, its old address will be removed by the tracker once it no longer responds and its new address will be added to the list once it comes online again, ready to provide data. (Atkinson, 2011)

3.2.3. What we gathered from P2P

What makes this system so interesting to us is its ability to manage and distribute a resource across a large and extremely dynamic network. The BitTorrent protocol solves the

issue of transient connections by having a single persistent and well-known server manage peer membership. And since the tracker can manage the network and resources, a peer only needs to be configured with tracker information to handle all future reconnections and relocations transparently. The success of BitTorrent has proven that this multi-tiered system is fully capable of transparent resource location and we adopted this model.

3.3. Messaging Systems

3.3.1. About messaging systems

Sometimes referred to as message-oriented middle-ware, messaging systems allow processes on different computers to pass messages to each other. Messages here can be either pre-defined data structures or opaque collections of bits. Since they are usually separated from resource providers and maintainers, these systems can be part of a larger modular system.

In the message systems we have encountered, there are two major levels - message producers/consumers, and message exchanges/brokers. Producers and consumers are simply end nodes that communicate back and forth with messages. All messages must be sent through a broker that routes them through a queue. For many message systems, the broker provides services such as backup, queuing, and subscriptions. (Korhonen)

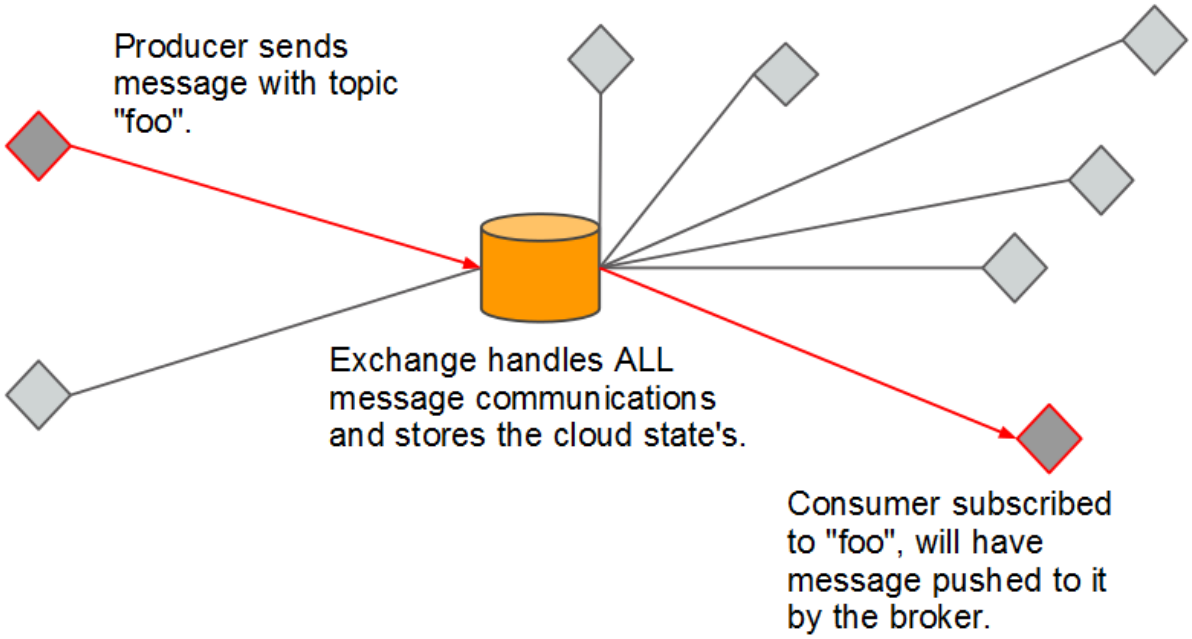

Figure 1: Simplified view of message queue architecture

3.3.2. About Java Messaging Service (JMS) 1.1

Java Messaging Service (JMS) is a messaging system interface released by Oracle that was last updated in 2003. As opposed to a communications specification, JMS only provides a list of interfaces and the outcome of using them. Although behavior is defined by JMS, implementation is not. Programmers are free to implement JMS however they want as long as the interface is functional and adheres to specification.

3.3.3. OpenMQ

OpenMQ is Sun's Java-based, open-source messaging software. It implements JMS 1.1 and was last updated around 2008. Although OpenMQ fully implements JMS, the JMS specification itself lacks certain features that we deem useful. Most of these features will be in JMS 2.0. (Oracle)

3.3.4. JMS 2.0

JMS 2.0 is Oracle's next proposed JMS specification, as of now it is still in draft after initial development, which stated around March 2011. 2.0 aims to update the JMS standard to match updated Java Enterprise Edition and to make it useful to the "cloud" as well as making various behaviors more standardized.

A more complete list of potentially relevant features includes:

- Improved Exception system with clearer descriptions of the errors that occurred.
    - We need this to deal with network interruptions/relocations/dropped messages.
- Making shared subscriptions easier to use.
    - This is similar to the work queue pattern in RabbitMQ, and can be used to delegate messages to a pool of available servers.
- Other useful features similar to those we saw in RabbitMQ, including.
    - Delivery delay - Specify a minimum timeout before sending the message.
    - Asynchronous ACK - Don't block in order to wait for acknowledgement, resulting in speedier responses and less wasted time.
    - Mandate delivery count – We will know how many times a message has been delivered, which is useful for knowing about duplicates.
    - Topic hierarchy - Provides more flexibility than simpler pattern matching in the previous JMS.
    - Multiple consumers - This provides a built-in pattern to facilitate broadcasting behavior.
    - Batch delivery - messages sent in batches asynchronously.

Note that JMS 2.0 is still JMS, and therefore is more of a specification than a concrete implementation or even protocol. It does not cover aspects such as load balancing, fault recovery, and storage and backup of messages. (Deakin, JMS Version 2.0 (Early Draft)) (Deakin, What's probably coming in Java Message Service 2.0) (Nigel Deakin)

3.3.5. AMQP

Advanced Message Queuing Protocol (AMQP) is a protocol that defines both the messaging interface as well as a concrete protocol that all implementations must follow. Unlike JMS, it goes a step further by defining what kind of bit exchanges occur during the actual transaction with a "wire protocol". The advantage of this is a very high level of inter-compatibility between messaging implementations. (OASIS)

3.3.6. RabbitMQ

RabbitMQ is a messaging system that follows the AMQP protocol. It is written in Erlang and is an open source project sponsored by a collection of companies.

3.3.7. Why we like RabbitMQ

In the testbed, we are using RabbitMQ in place of other systems because we felt that it is well-documented and easy to use. Like any other message-oriented middle-ware, it can be used as an intermediate layer that can be used for resource discovery and requests, freeing the client from having to know where each individual resource is.

This way, RabbitMQ will act as a service that will package any resource request into the appropriate message, send it to the exchange so it can be forwarded to the appropriate resource holders, and see if any resource holders have the available resource.

For large systems, RabbitMQ provide patterns such as Work Queues, High Availability Clusters, and reception confirmations for load balancing, transaction acknowledgement, and duplication of messages in case of failure on one machine.

3.3.8. Structure of RabbitMQ

The messaging model RabbitMQ uses provides Exchanges separate from the message Producers and Consumers. These Exchanges are message hubs where all requests and responses are routed and filtered through. In our interpretation of resource transparency, a message is either a request for a resource or a response to a request. Message topics can indicate the type for resources needed and the message payload can be used to contain additional data about the request. All Exchanges do their job by filtering and routing messages to the right consumers based on Message Topics. Resource holders can opt for messages with topics that indicate requests for certain types of resources while ignoring ones that are not relevant. RabbitMQ Exchanges also offer high availability clustering for, delivery confirmation, and various built-in messaging patterns that give it the flexibility to adapt to various resource properties. (VMware, Inc)

3.3.9. Usefulness of Messaging Systems

Recommended by Oracle at the beginning of the project, we found that messaging systems can dynamic, flexible, and very reliable. It has existing robust implementations such as

RabbitMQ, and future versions of JMS 2.0 promise more messaging systems to come. Its characteristics, along with many advanced features, make it the best-suited platform for us to build our solution on.

4. Solution Requirements

4.1. General needs of the solution.

From Oracle's request we derived three aspects of the problem that we should focus on. These are resource transparency, scalability, and availability. The core of these is resource transparency. We interpreted resource transparency as meaning the ability of the client to connect to a resource seamlessly regardless of any problems that occurred with resource delivery. At the most basic level it should be able to handle problems arising from a change of the network address of the resource-holding server.

Additionally, the solution needed to be scalable to larger systems so that it can fit any purpose. Because we intended to design a general system that could function for whichever resources were being serviced, our system should not restrict itself in terms of scale. Whichever system maintained a connection to the resource must be able to handle requests from several clients as well as information about several resources. This requirement implies that a solution designed for a specific type of resource should be avoided.

And, lastly, the solution needs to be available enough so that it can actually implement resource transparency. The availability condition was crucial since all the solutions we considered used some sort of middle-ware. This middle-ware served as a constant server that could keep track of the underlying changes in the resource. Therefore if the middle-ware was unreliable, the transparency of the resource could not be ensured.

5. Dynamic Domain Name System (DDNS) Solution

We first considered a solution involving a dynamic domain name system (DDNS). In this solution we envisioned a broker cluster as the resource to be made transparent. Any server could send messages to this cluster to a chosen queue and have its messages stored transparently. A client could retrieve these messages by connecting to the broker cluster and requesting messages from the server's assigned queue.

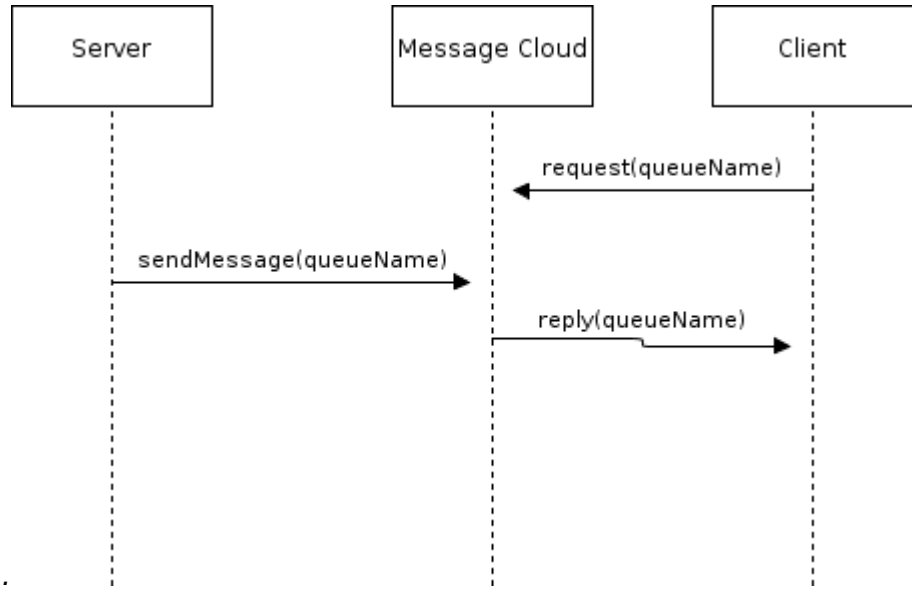

Figure 2: This figure demonstrates how the server and client interact with the cloud. The cloud stores the resource as a message on a queue. The server sends messages to the cloud and the user can get access to those messages by requesting them from the cloud.

In this solution any broker in the cluster could possibly have changed its network location. Changes in the network location would be detected through dynamic DNS. A server running on the broker machine would detect changes in its location and inform the DNS server of them. The middle-ware would detect these changes in the DNS configuration of its brokers and update its connection to them.

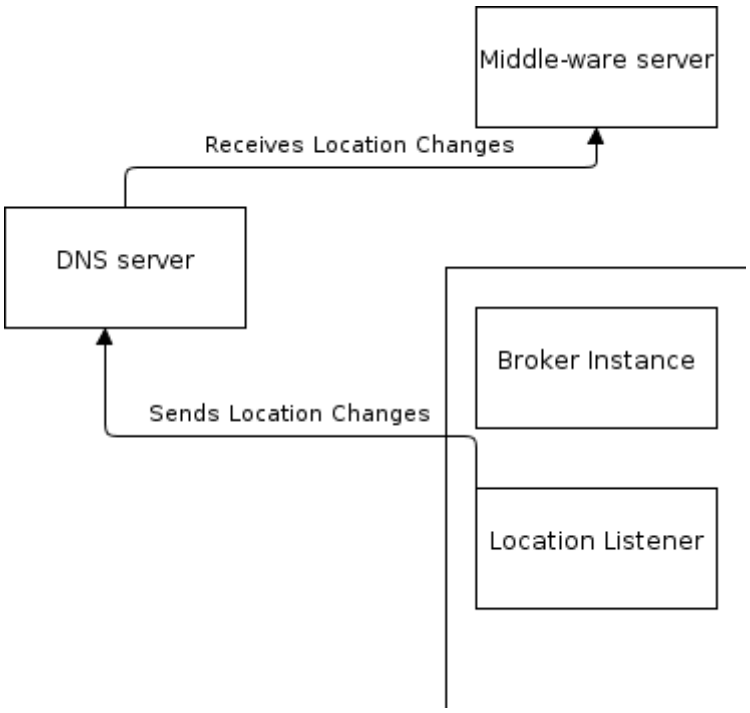

Figure 3: This figure shows how the location listener, DNS server and middle-ware interact.

We abandoned this solution for several reasons. The first was that Oracle preferred a more middle-ware oriented solution that did not rely on other services. Secondly, since the design required for the brokers themselves to be made transparent, it was necessary that any message-oriented middle-ware being used to implement the solution would have to be modified to handle clusters in which the brokers could change location. Thirdly, system was not design to allow for interactions between the server and the client in real-time. It is also limited in the number of possible connection errors it can handle transparently. And, lastly, it was more natural to interpret the servers that were providing a service as the resource to be made transparent.

6. Message-Oriented Middle-ware (MOM) Solution

6.1. Why we chose Message-Oriented Middle-ware

Message-oriented middle-ware was first brought to our attention during a meeting with Oracle. After examining its features and design, we felt that incorporating an existing and mature communication system into resource transparency would allow us to adapt its features and apply it as a general transparency solution without re-inventing the wheel.

With their various message routing systems, MOMs proved themselves to be flexible enough to provide transparency to various kinds of resources, which is critical to a general solution. Since there are already plenty of existing solutions for all kinds of replicated resources, the middle-ware would only have to provide a means to re-establish a connection with the resource in the cloud. This approach will relieve complexity from our own product as well as provide a channel for established products to be included into our resource-transparent cloud.

6.2. Test bed configuration and setup

Our test bed setup was provided and configured by the WPI Computer Science Department. The test bed was a group of six KVM virtualized servers running on a single host. The host configuration had an Intel Core2Duo clocked at 2.66 GHz, with 8 GB of memory, and 136 GB of disk space. Each of the vitalized machines was configured with one virtual core, 512 Mb of ram, and 4 GB of disk space. Although the host has access to the outside network, the virtualized instances were connected via a virtual network on the host and can only be accessed from the host alone for security reasons. The virtual network automatically assigns IPs to each of the virtual machines.

All machines had up-to-date versions of OpenJDK installed as well as RabbitMQ.

The low computing power afforded by the setup forced us forgo physical access and graphical interfaces and opt for remote terminal access only. The source code used for the testing will be included in a separate archive and will be explained further in the evaluation sections.

6.3. Evaluation of OpenMQ

We first attempted to test Message systems with OpenMQ. We wrote two simple programs, one that produces a message and another that receives messages from an OpenMQ broker. The broker was setup using OpenMQ utilities. After sending some messages we attempted to create a high-availability cluster using OpenMQ.

OpenMQ uses an MySQL database for storing messages in a high-availability cluster. This proved to be quite difficult and OpenMQ's documentation was not particularly helpful in resolving errors that we experienced while configuring the MySQL database. So we looked for another system.

6.4. Evaluation of RabbitMQ

The second system we evaluated was RabbitMQ, an open source message queuing software developed in Erlang. We found that the installation usage was relatively pleasant thanks in part to both the ample documentation available and its inclusion in the official Ubuntu repository, although newer versions of the RabbitMQ software package can be found on its

official website. Since we are working in Java for this project, we also downloaded the Java client, which allowed us to interface with RabbitMQ via Java.

We found RabbitMQ to be generally more manageable and easier to use. Basic messaging patterns were already well-documented on its website and the open-source examples made the test bed easy to set up, configure, and test. The overall design of RabbitMQ seems to focus on ease of use and deployment, which we appreciated greatly. The pattern that we found to be the most useful is the high-availability cluster, which replicates the state and information of a message exchange so that in the event of an exchange failure, others will be able to quickly take over and prevent serious data loss.

7. Solution Design

7.1. Solution description

Based on the initial testing with the virtual machines and different message queuing frameworks, we decided that the middle-oriented framework is a fairly mature middle ground and well-suited to our intention of location transparency.

The intended design is to build a thin layout over the message-oriented middle-ware to support resource location transparency while making use of the existing features of the middle-ware such as high-availability, scalability and reliability. This overlay layer consists of two main interfaces which are VirtualServer and VirtualClient. These will communicate with the real client and real resource server, respectively, as if they were their real counterparts. Sitting between the two virtual interfaces are the MOM-brokers cloud dynamically managed by the middle-ware.

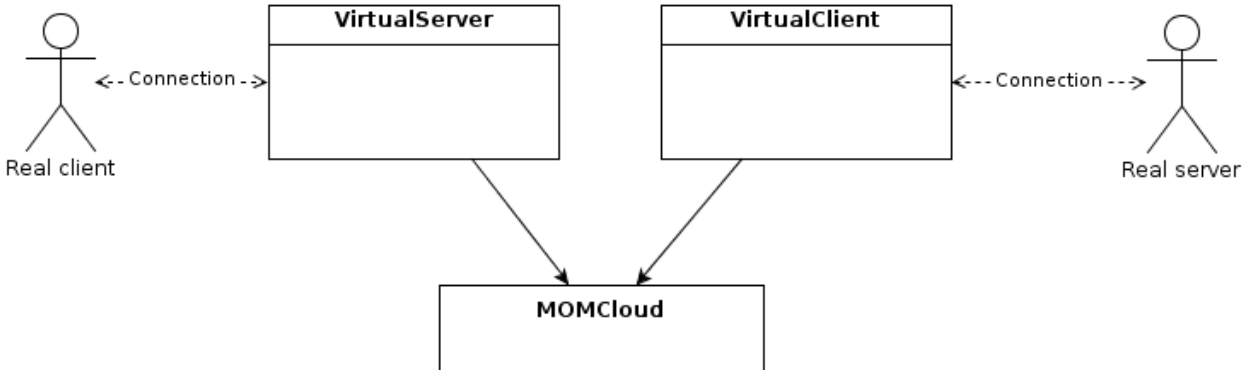

Figure 4: Shows the relationships between the real client, virtual server, virtual client, cloud and real server. The real server and client send messages to each other through the cloud using the virtual server and virtual client

7.1.1. Connection data flow

Instead of talking directly with the real server, the client will initiate and communicate with the VirtualServer via a network socket as if it were the real server. The VirtualServer will, in turn, forward the application-layer messages it has received from the client to the broker cloud. The messages were then reliably transferred to the VirtualClient through the cloud using the middle-ware messaging queues. Finally, the VirtualClient forwards the request and receives the response from the real server using a network socket.

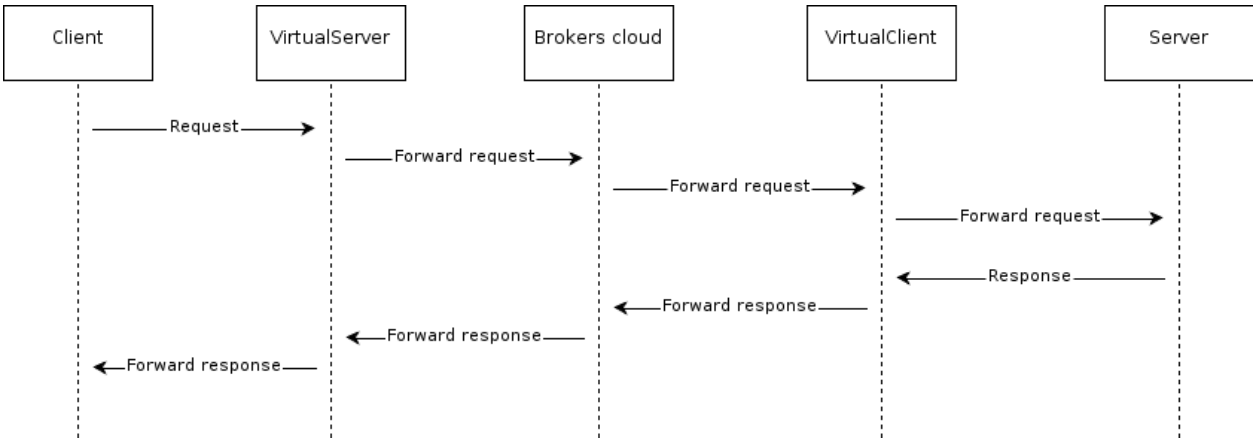

Figure 5: Sequence diagram showing the data flow for a single request-response from client

7.2. Scenarios, features and limitations

7.2.1. Multiple clients support

If there are multiple clients requesting for the resource at the same time, there should be a reliable way to separate or classify messages to/from different clients. Fortunately, this can be solved using one of the most notable features of the message-oriented middle-ware: the ability to create and destroy queues dynamically. More specifically, every time a client establishes a socket connection with the virtual server, two separate queues must be established in the middle-ware: one routing the data upstream and the other downstream. Additionally, the virtual

client needs to know about the newly established queues (because it is on the other side and have not known the new client yet); therefore, a global control queue is also needed for such announcements.

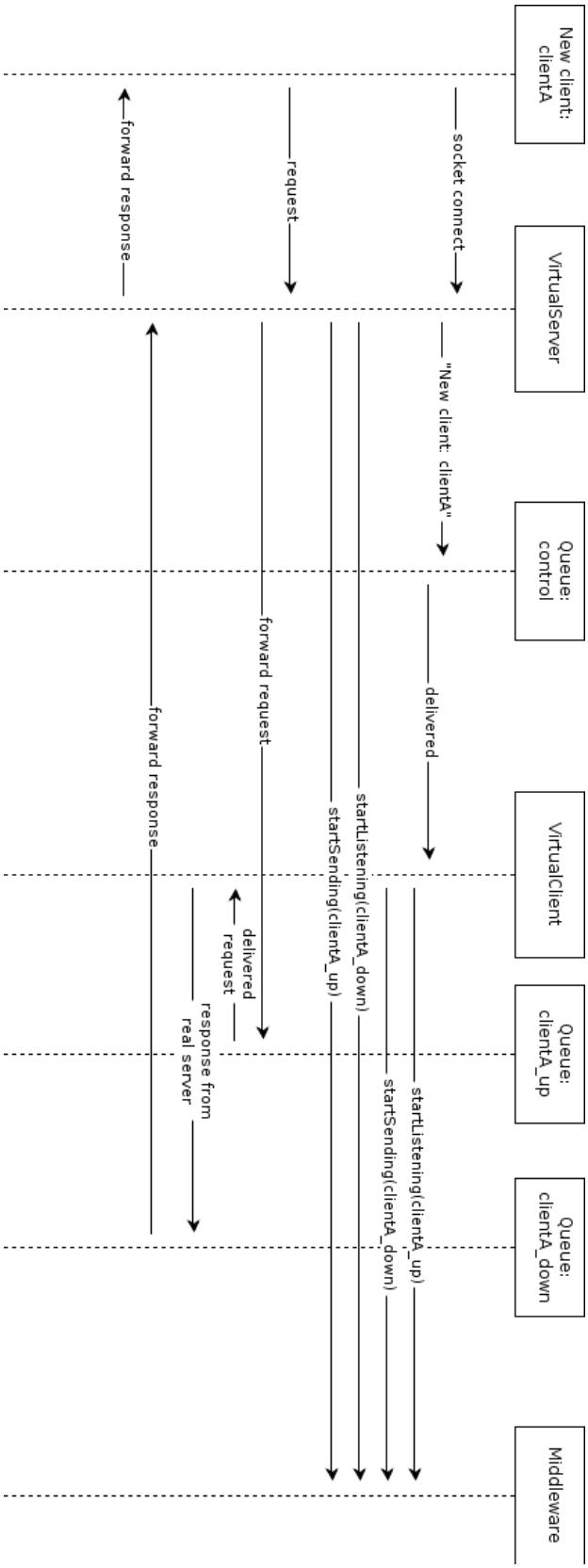

Figure 6: Multiple-client support sequence diagram

7.2.2. Client failures

There is no feasible solution for this scenario. Since a failure on the client-side means that they will not be able to receive responses from the server at all, it would be meaningless to build tolerance for it. When the client goes down, it should just try to set up the new connection with the system when it recovers.

7.2.3. VirtualServer/VirtualClient failures/redundancy

As the diagram demonstrated, the VirtualServer and VirtualClient are decoupled and not directly connected with each other but through the messaging queues instead. This implies that we can simply have multiple redundant VirtualServers or VirtualClients on each side so that failure of a single machine would not bring the whole system down.

On the other side, the real clients and servers are connecting with the virtual interfaces using socket connections, so a failure on a virtual server/client might be noticeable. However, this issue can be fixed by setting a single domain name for all redundant servers/clients in such a way that the next time the real client/server attempts to connect to the domain it would be paired with the next working redundant virtual server/client.

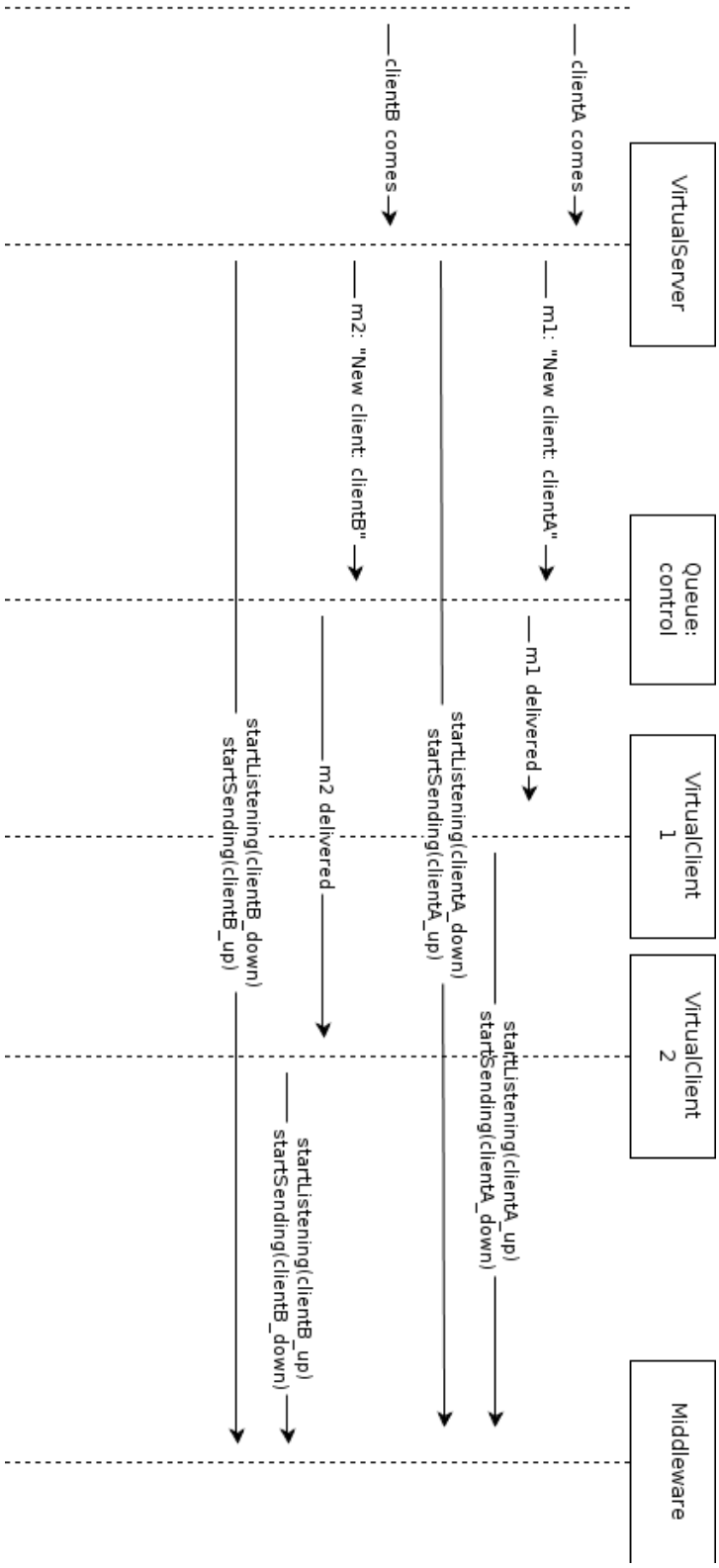

Figure 7: Sequence diagram for system with redundant virtual clients

7.2.4. Redundancy on server side

Redundancy on the server side is achieved by allowing the virtual client to be configured with the possibility of adding multiple servers. The default behavior of the virtual client is to instantiate a connection to a single server each time it receives a request for a connection from the cloud. With redundancy this is changed so that virtual client can forge a connection to any one of a list of servers depending on such factors as load-balancing or whether or not a server is running.

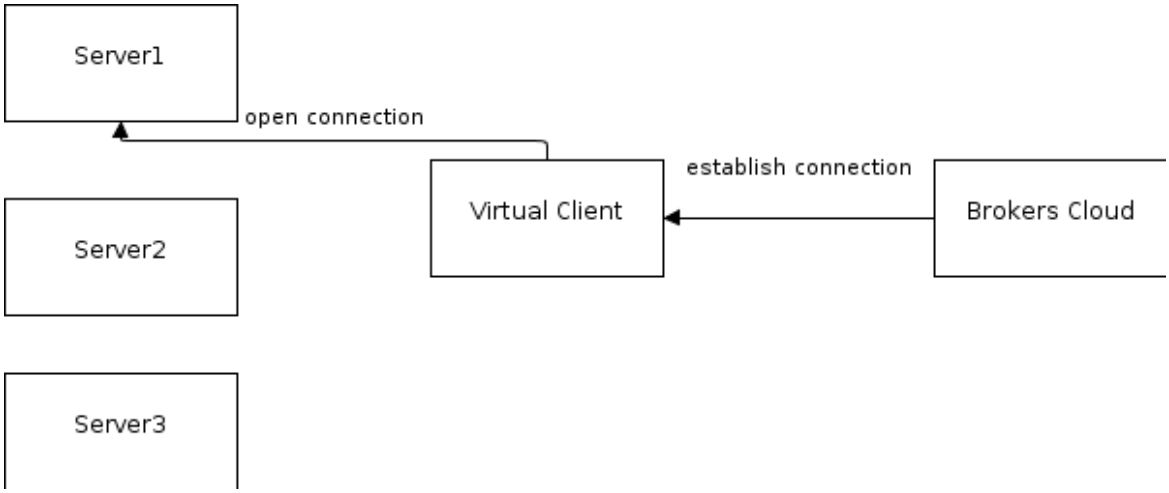

Figure 8: Virtual client getting first connection request and choosing to connect to server1

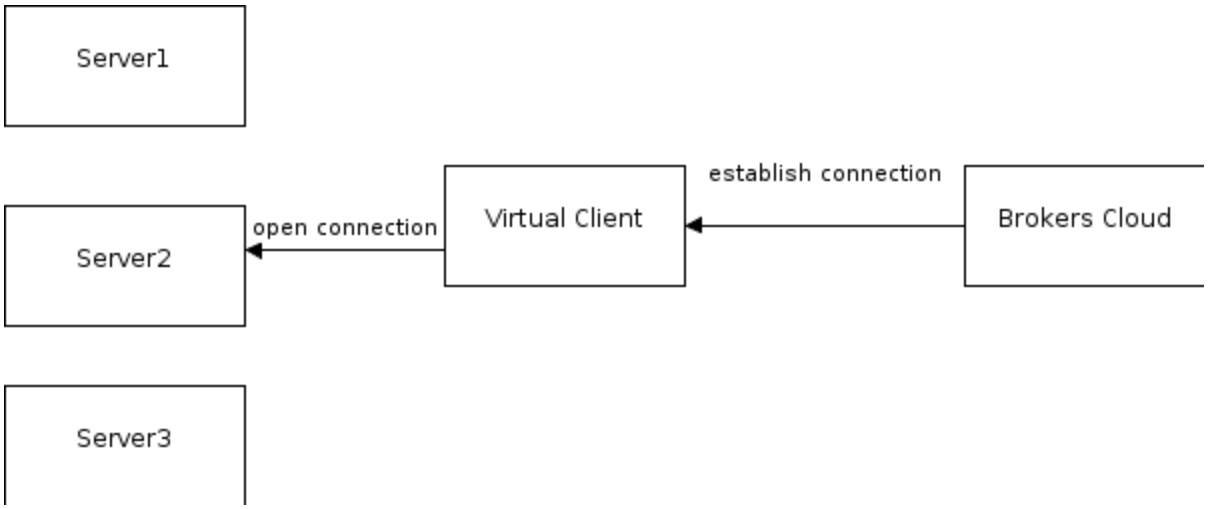

Figure 9: Shows virtual client getting second connection request and choosing to connect to server2

7.2.5. Redundancy in the brokers cloud

The middle-ware cloud already has some redundancy features built-in such as high-availability cluster, replicated brokers, and seamless synchronization. Also, recent versions of the different middle-ware systems include the ability to add/remove brokers dynamically at run time.

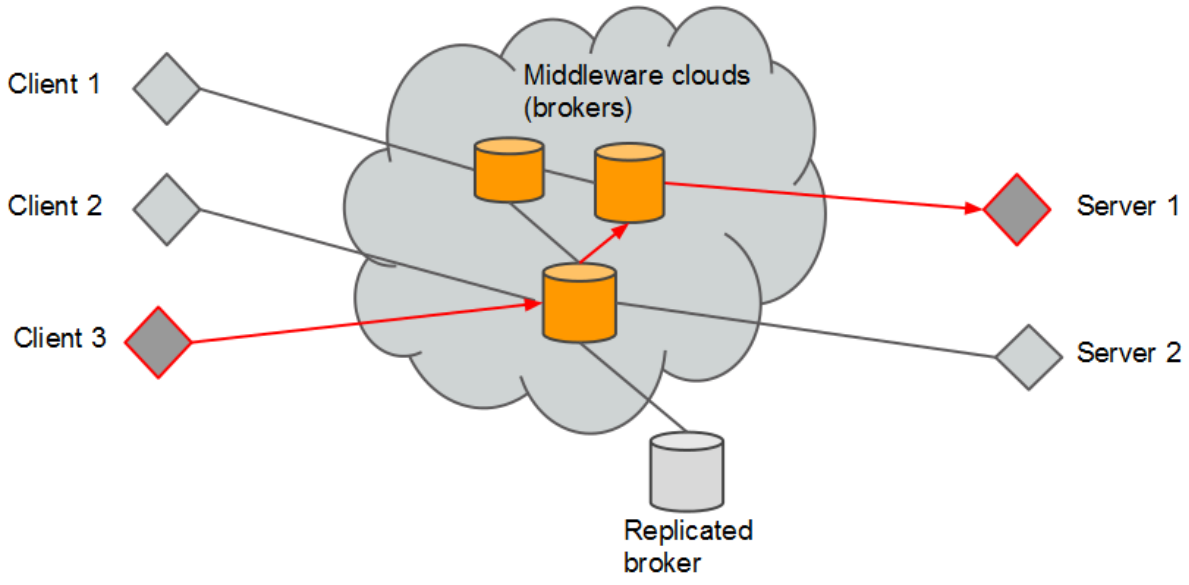

Figure 10: Simple model of replicated broker cluster

7.2.6. Transparency to change in resource location

Whenever there is a resource location change event, it can be split into two separate events: the removal of the old resource server from the system, and the introduction of the new one. As servers can be added and removed dynamically very easily as discussed in section 7.2.4 above, there should be no difficulty at all moving the server around, although there might be some minor glitch during the transition (some connections might be lost and the client needs to reconnect to get to the new resource server).The virtual client would have to maintain the connection it has to the cloud and reset it if it is disconnected. It detects changes to the server configuration and recreates the connections to the server depending on the type of change in the server configuration.

From the client point of view, it never talks with the real server, nor knows the location of the real resource. When the resource changes its location, the virtual client will route the client's request to the new destination automatically.

7.2.7. Multiple services

Different services such as HTTP, DNS, database, and file servers can be virtually hosted on the same virtual server by having the server connect to a different port depending on the type of message it received from the virtual client for establishing a new connection. Their message queues share the same middle-ware cloud and are differentiated by using different queue base names.

7.3. Summary: the benefits

Failure-tolerance: As previously described, this method does offer a significant level of redundancy and failure tolerance thanks to the decoupling nature of the underlying middle-ware. Redundancy machines can be set up easily on both the client and server side, as well as in the brokers cloud.

Scalability: Load-balancing can be achieved in the system using redundancy machines. Also, the configuration is dynamic, so that machines can be added/removed if necessary on the fly, with no interruption of service.

Resource location transparency: In this approach, the entity that the real client communicates with is the virtual server; therefore, any location change from the real server is transparent to the client. The client also needs to know only the location of the virtual server to use different services as discussed above, therefore the real locations of different resources are well hidden behind the middle-ware, which also contributes to the location transparency concept.

7.4. The limitations

There might be some overhead in the data flow, as we are introducing additional layers in the middle to achieve transparency. In order words, if looking into the CAP theorem, we are actually trading availability of resource (longer response time) for better partition tolerance.

8. Implementation of Message-Oriented Middle-ware Solution

We programmed a simple implementation of the concept of our solution. Our main goal for it was to test whether it is a feasible approach, and to make sure that it does not have any significant limitations that we may have overlooked.

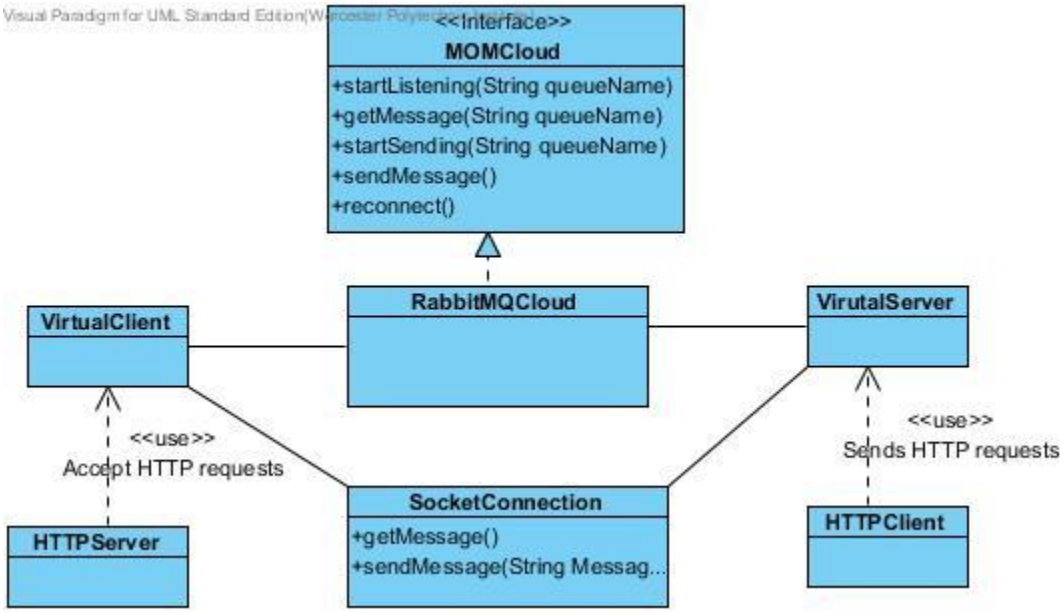

Figure 11: The interface of the MOM cloud solution

We have set up one virtual server and one virtual client connected by the middle cloud run by RabbitMQ. Although the solution may work with different services, during the scope of this project, we decided to test the system with the HTTP protocol by emulate a remote webserver (like google.com) on the virtual server.

The program has two main routines, one that implements VirtualClient and another that implements VirtualServer. A VirtualServer listens on a port specified by the user and when connected to by a browser establishes a connection to the http server by sending a message to a

VirtualClient through a RabbitMQCloud instance. Replies for the http server are sent to a VirtualClient and a VirtualClient sends them to the cloud. Replies from the http client are sent to a VirtualServer which then sends them to cloud.

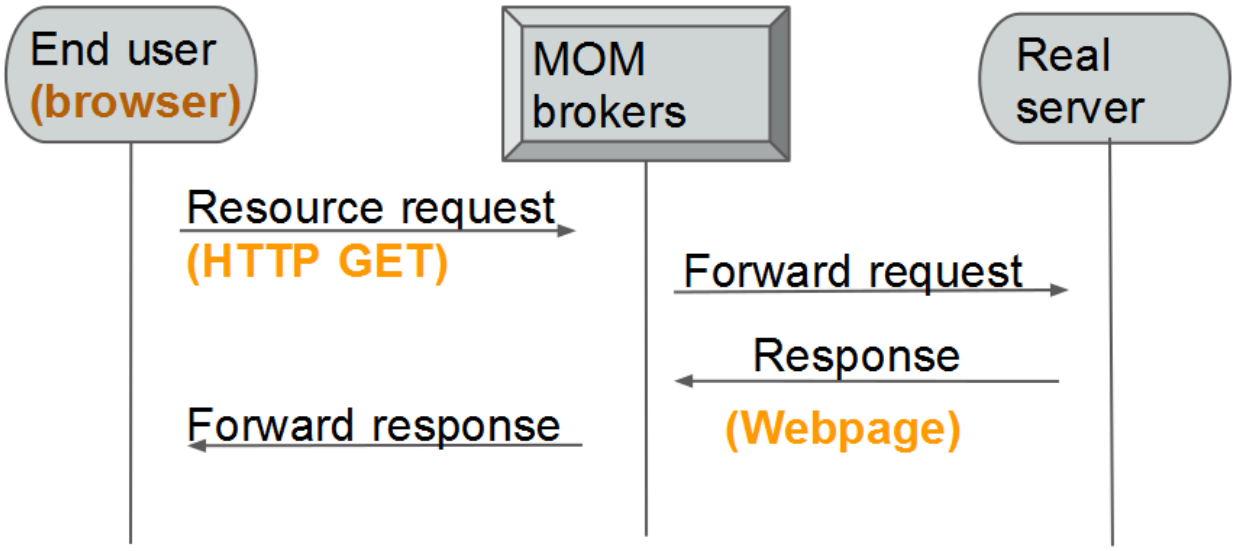

Figure 12: Interaction between a client (browser), cloud, and server (HTTP server).

The implementation handles the connections to the HTTP server and the client by using the SocketConnection class. The getMessage and sendMessage functions in this class are used to send and receive messages from the connections maintained by both a VirtualClient and a VirtualServer.

9. Testing Results

The package was packaged as a zip file, which includes the source files and several RabbitMQ jar files that need to be included in the classpath when compiling or running the implementation. The zip file also contains scripts required to compile and run the implementation.

In this test the virtual server was set to listen on port 80 in order to get requests from a browser that was run locally on the machine. The RabbitMQ cloud was a simple setup involving only one broker. In order to simplify the testing the virtual server, virtual client, the RabbitMQ middle-ware and the RabbitMQ broker were all run on the same machine. After running both the virtual server and client we ran the browser. We used the *Links* web browser to connect to the virtual server. The command used to do this can be seen in Figure 13-16.

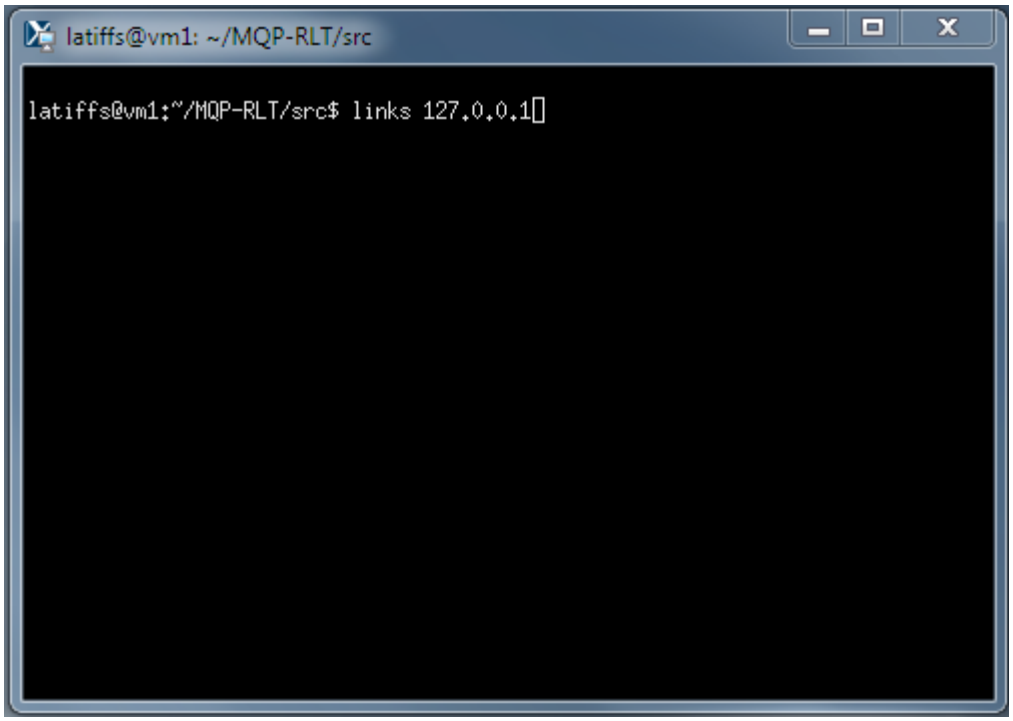

Figure 13: Connecting to Server using Links

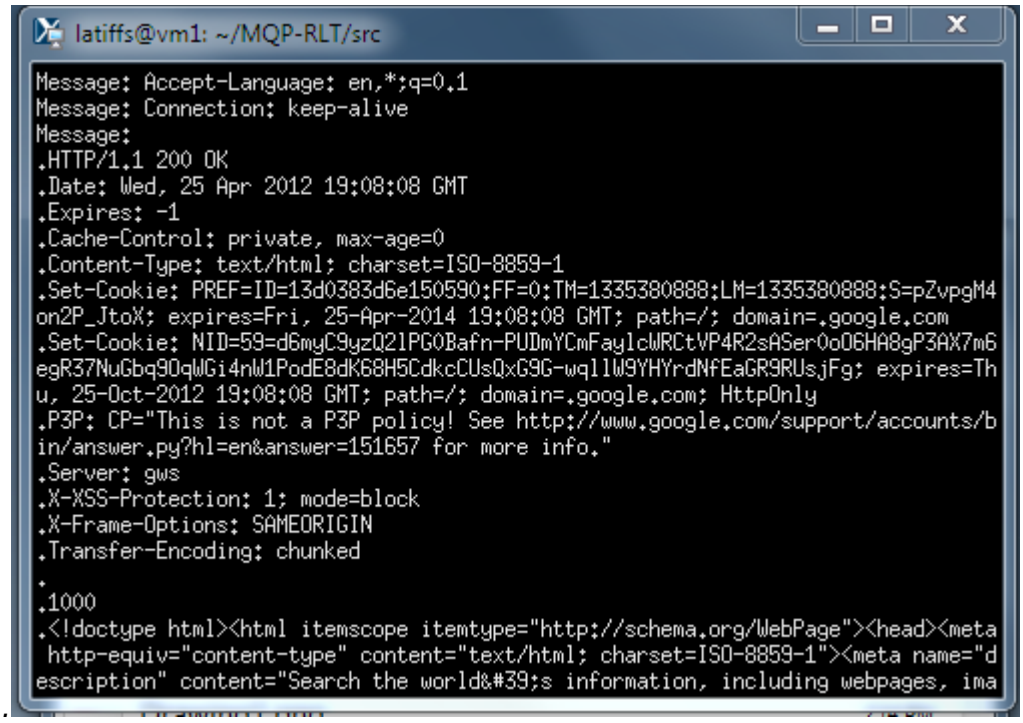

Figure 14: This screen capture shows the response from the virtual client which it received from the Google web server

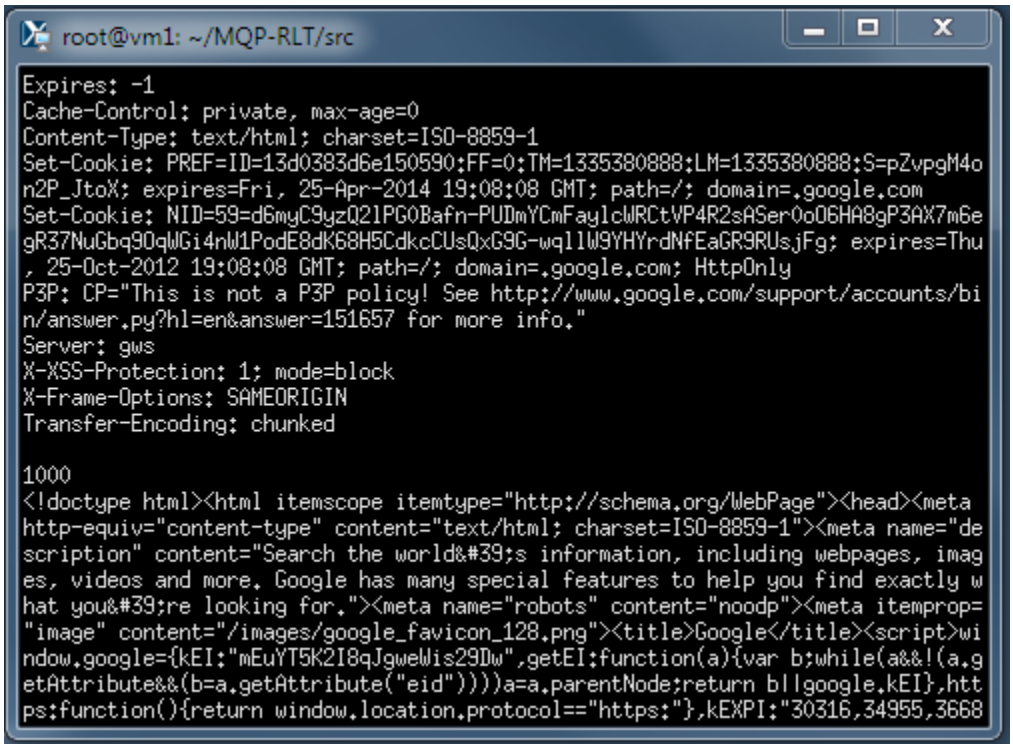

Figure 15: This capture shows the response that the virtual server received from the cloud

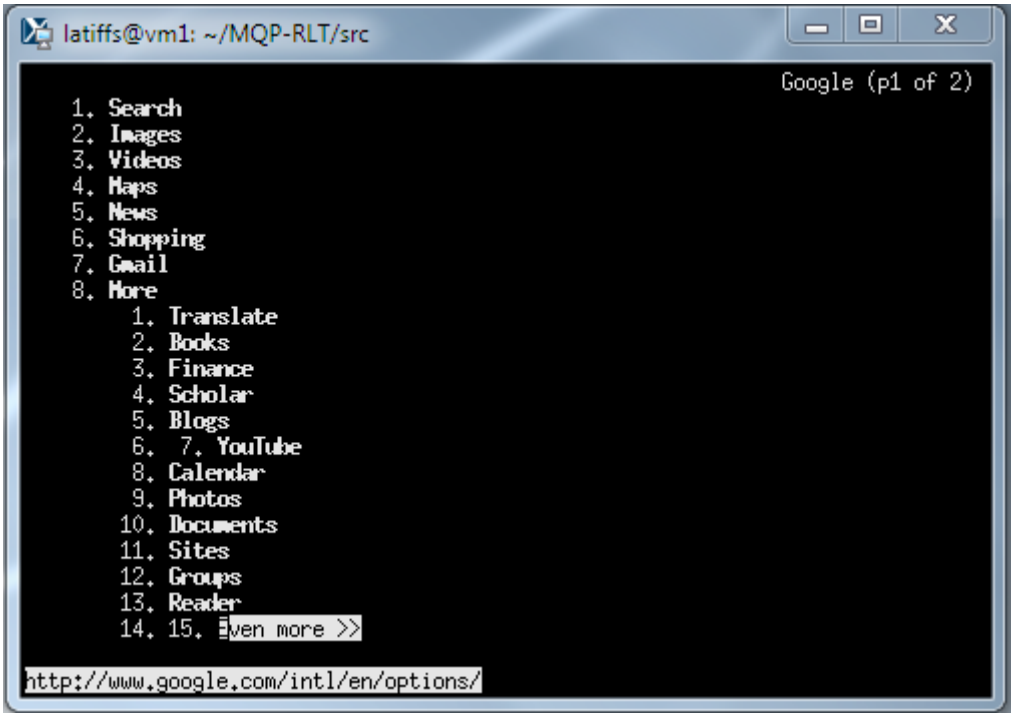

Figure 16: This screen capture shows the links browser rendering the page that it received from the virtual server

Also, it should be noted that opening up a web page like google.com in a browser would initiate multiple connections to the server. Therefore this experiment also demonstrated that our implementation works in a multiple-client with multiple connections environment. (See figure 17)

Figure 17: This screen capture shows that our implementation supports multiple clients/connections and shows the virtual client receiving a request from a second instance Links

Lastly, we also integrated a transparency feature in such a way that, whenever a network broker lost a connection with the MOM-cloud, it would automatically attempt to reconnect shortly after and restore the initial connection state. The feature was implemented using the shutdown protocol provided by RabbitMQ, which was an advanced feature not supported yet by the current release of OpenMQ (1.0).

To test this feature, vm1 was set up as the VirtualServer and vm4 as VirtualClient. vm4 network card was then shut down for 10 seconds. After the restart and reconnection of vm4, vm1 could still be able to serve the content of the webpage as if nothing had happened. (See figure 18-19)

Figure 18: This screen capture shows the vm4 network connection being shut down

Figure 19: This screen capture shows the vm4 network being restarted

10. Conclusions, Future Work, and Recommendations

Our solution consisted mainly in creating a design that can be implemented fully later by another group. We tested some of the features of this design but some features are yet to be tested. Some future work might include using redundancy on the server side by building in load-balancing. Handling resource transparency in this context would require running a local service on each server machine that sends the virtual server information about location changes. Another interesting extension of our work would be using a different messaging system. RabbitMQ has the flexibility we needed to create a dynamic high-availability cluster, but Oracle is developing a new version of OpenMQ which will provide the same flexibility with ease of use. Modifying our implementation to function with OpenMQ would be an affirmation of generality of our design.

Further investigations can be made on designing an effective way of handling timing between cloud messages and the virtual objects (virtual server and client). During some tests with our basic implementation we experienced problems with delivering requests over the cloud. These problems were due to problems in timing requests between the virtual objects and their connections to the real objects with their connections to the cloud.

Works Cited

Atkinson, A. (2011, 4 25). *What is BitTorrent's architecture?* Retrieved 4 21, 2012, from Quora: http://www.quora.com/What-is-BitTorrents-architecture

Deakin, N. (n.d.). *JMS Version 2.0 (Early Draft).* Retrieved 4 21, 2012, from http://download.oracle.com/otn-pub/jcp/jms-2_0-edr-spec/jms-2_0-edr-spec.pdf?AuthParam=1335390883_f7bfe901fd7818474026ec33cf71b4a9

Deakin, N. (n.d.). *What's probably coming in Java Message Service 2.0.* Retrieved 4 21, 2012, from http://java.net/projects/jms-spec/downloads/download/javaone2011-jms20.pdf

Nigel Deakin, C. S. (n.d.). *JSR 343: What's Coming in Java Message Service 2.0.* Retrieved 4 21, 2012, from http://java.net/downloads/jms-spec/javaone2011-jms20.pdf

OASIS. (n.d.). *Developer FAQ's*. Retrieved 4 21, 2012, from AMQP: http://www.amqp.org/resources/developer-faqs

Oracle. (2011). ORACLE COHERENCE 3.7.

Oracle. (n.d.). *Open Message Queue*. Retrieved 4 21, 2012, from Java.net: http://mq.java.net/overview.html

Sathupadi, K. (2011, 8 18). *A plain english introduction to CAP Theorem*. Retrieved 4 21, 2012, from http://ksat.me/a-plain-english-introduction-to-cap-theorem/

Seth Gilbert, N. L. (n.d.). *Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.*

VMware, Inc. (n.d.). *RabbitMQ - Documentation*. Retrieved 4 21, 2012, from RabbitMQ: http://www.rabbitmq.com/documentation.html