Autoscaling Web Services on Amazon EC2

By: M. Afkham Azeez (Index# 078268D)

This dissertation was submitted to the Department of Computer Science and Engineering of the University of Moratuwa in partial fulfillment of the requirements for the Degree of MSc in Computer Science specializing in Software Architecture

Department of Computer Science & Engineering
University of Moratuwa, Sri Lanka
21st February 2010
Declaration

The work included in this report was done by me, and only by me, and the work has not been submitted for any other academic qualification at any institution.

.................................................. .................................................. ..................................................

Afkham Azeez Date

I certify that the declaration above by the candidate is true to the best of my knowledge and that this report is acceptable for evaluation for the CS6999 M.Sc. Research Project.

.................................................. ..................................................

Sanjiva Weerawarana (PhD) Date
ABSTRACT

Fault tolerance, high availability & scalability are essential prerequisites for any enterprise application deployment. One of the major concerns of enterprise application architects is avoiding single points of failure. There is a high cost associated with achieving high availability & scalability. We will look at an economical approach towards automatically scaling Web service applications while maintaining the availability & scalability guarantees at an optimum economical cost. This approach, involving the Amazon EC2 cloud computing infrastructure, makes it unnecessary to invest in safety-net capacity & unnecessary redundancy. The Web service application developer should only need to write the application once, and simply deploy it on the cloud. The scalability & availability guarantees should be provided automatically by the underlying infrastructure. Auto-scaling refers to the behavior where the system scales up when the load increases & scales down when the load decreases. Auto-healing refers to an approach where a specified minimum deployment configuration is maintained even in the event of failures. Such an approach is essential for cloud deployments such as Amazon EC2 where the charge is based on the actual computing power consumed. Ideally, from the clients' point of view, in an autoscaling system, the response time should be constant and the overall throughput of the system should increase. We will describe in detail an economical approach towards building auto-scaling Apache Axis2 Web services on Amazon EC2. In the course of this article, we will introduce well-known address (WKA) based membership discovery for clustering deployments where multicast-based membership discovery is an impossibility. We will also introduce an approach towards dynamic load balancing, where the load balancer itself uses group communication & group membership mechanisms to discover the domains across which the load is distributed. In a traditional setup, a single load balancer fronts a group of application nodes. In such a scenario, the load balancer can be a single point of failure. Traditionally, techniques such as Linux HA have been used to overcome this. However, such traditional schemes have quite a bit of overhead and also require the backup system to be in close proximity to the primary system. In case of catastrophic situations, this approach can result in complete failure of the system. We will introduce an auto-healing scheme in case of load balancer failure using Amazon Elastic IP addresses & a load balancer group, which can overcome these shortcomings.
ACKNOWLEDGMENTS

I would like to express profound gratitude to my advisor, Dr. Sanjiva Weerawarana, for his invaluable support, encouragement, supervision and useful suggestions throughout this research work. His continuous guidance enabled me to complete my work successfully.

I am grateful for the support & assistance provided by the management of WSO2, who provided me the research facilities. This work may not have been possible without the Amazon Web services R&D account that was provided by WSO2. This account was extensively used during this research. This work would not have been possible without the support & assistance I received from Filip Hanik, author of the Apache Tribes group communication framework, which is extensively used in my work. I would also like to thank Asankha Perera, software architect at WSO2 & a lead architect of Apache Synapse, who provided advice on architecture & design. I would also like to thank Ruwan Linton, a lead developer of Apache Synapse, and Paul Fremantle, project lead of Apache Synapse, who provided me with design ideas related to dynamic load balancing & load analysis. I would also like to thank my colleague & good friend Amila Suriarachchi for reviewing my work and providing valuable feedback & suggestions for improvements.

I would like to thank Chinthana Wilamuna, the scripting languages expert at WSO2, for providing his expert advice on various aspects, as well as Deependra Ariyadewa & Chamith Kumarage, Linux experts, for providing their expertise.

I am grateful to the open source communities backing the Apache Axis2 & Apache Synapse projects. Their contributions were the foundation upon which this work was built.

I am as ever, especially indebted to my parents for their love and support throughout my life. I also wish to thank my wife, who supported me throughout my work. Finally, I wish to express my gratitude to all my colleagues at WSO2. Many ideas related to this project have come to my mind when having technical & even non-technical discussions with this group of intellectuals.
TABLE OF CONTENTS

ABSTRACT ........................................................................................................................................... iii
ACKNOWLEDGMENTS ......................................................................................................................... iv
TABLE OF CONTENTS ........................................................................................................................... v
LIST OF ABBREVIATIONS ................................................................................................................... ix
Chapter 1 Introduction ......................................................................................................................... 1
  1.1 Amazon Elastic Compute Cloud (EC2) ......................................................................................... 3
  1.2 EC2 Features for Building Failure Resilient Applications ......................................................... 4
  1.3 Apache Axis2 ................................................................................................................................ 5
  1.4 Apache Synapse Dynamic Load Balancing .................................................................................. 6
  1.5 The Problem ............................................................................................................................... 6
  1.6 Objectives ................................................................................................................................. 7
  1.7 Prior Work ................................................................................................................................... 8
Chapter 2 Literature Review ............................................................................................................. 10
  2.1 Terminology ............................................................................................................................. 11
  2.2 Theoretical Aspects .................................................................................................................... 12
    2.2.1 Failure Detection .................................................................................................................. 12
    2.2.2 Overcoming Distributed System Failures ............................................................................ 14
    2.2.3 High Availability ................................................................................................................ 15
    2.2.4 Group Membership Service (GMS) .................................................................................. 16
    2.2.5 Data Replication Models ................................................................................................... 16
    2.2.6 Virtual Synchrony .............................................................................................................. 17
    2.2.7 Membership ...................................................................................................................... 18
    2.2.8 Load Balancing .................................................................................................................. 18
    2.2.9 Reliability & Web services ............................................................................................... 19
    2.2.10 Group Communication Frameworks (GCF) .................................................................... 20
  2.3 Amazon EC2 ............................................................................................................................... 20
    2.3.1 Using EC2 ......................................................................................................................... 20
    2.3.2 Instance Types ................................................................................................................... 21
    2.3.3 Features for Building Failure Resilient Applications ......................................................... 21
    2.3.4 Data Transfer Charges ...................................................................................................... 22
  2.4 Giga Spaces ............................................................................................................................... 23
Chapter 3 Methodology ................................................................................................................... 24
  3.1 Introduction ............................................................................................................................... 25
  3.2 Well-known Address (WKA) based membership ....................................................................... 26
  3.3 Fault Resilient Dynamic Load Balancing .................................................................................. 28
  3.4 Dynamic Load Balancing with Load Analysis .......................................................................... 34
LIST OF FIGURES

Figure 1: Scale-up when the system load increases ........................................................................ 7
Figure 2: Scale-down when the system load decreases ................................................................. 8
Figure 3: Passive Replication ........................................................................................................ 16
Figure 4: Multicast from client ...................................................................................................... 17
Figure 5: Replies from replicas ..................................................................................................... 17
Figure 6: Member joins group ......................................................................................................... 27
Figure 7: Well-known member rejoins after crashing ................................................................. 27
Figure 8: Load balancer & application groups ............................................................................. 29
Figure 9: Active-passive load balancers with Elastic IP .............................................................. 29
Figure 10: Membership channel architecture ............................................................................... 30
Figure 11: Initialization channel architecture .............................................................................. 31
Figure 12: Application member joins. A load balancer is also a well-known member ............... 32
Figure 13: A non-WK load balancer joins .................................................................................... 33
Figure 14: A well-known load balancer rejoins after crashing .................................................. 34
Figure 15: Deployment on EC2 .................................................................................................... 36
Figure 16: Normal message flow ................................................................................................. 40
Figure 17: Error flow .................................................................................................................... 42
Figure 18: synapse.xml ................................................................................................................. 43
Figure 19: LoadAnalyzerTask configuration in synapse.xml ...................................................... 45
Figure 20: axis2.xml in application node ..................................................................................... 49
Figure 21: autoscale-init.sh script ............................................................................................... 51
Figure 22: Axis2 instance bootup .................................................................................................. 52
Figure 23: Synapse load balancer instance startup ....................................................................... 53
Figure 24: Bootstrapping the system ............................................................................................ 54
Figure 25: Performance Testing Scenario 1. Non-autoscaling single worker instance .......... 60
Figure 26: Performance Testing Scenario 2. Autoscaling system ............................................. 61
Figure 27: Response Time Variation ............................................................................................. 61
Figure 28: Throughput Variation ................................................................................................ 62
LIST OF TABLES

Table 1: EC2 Instance Types ...................................................................................................................... 21
Table 2: LoadAnalyzerTask configuration parameters........................................................................... 45
**LIST OF ABBREVIATIONS**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMI</td>
<td>Amazon Machine Image</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>EC2</td>
<td>(Amazon) Elastic Compute Cloud</td>
</tr>
<tr>
<td>ESB</td>
<td>Enterprise Service Bus</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>GCF</td>
<td>Group Communication Framework</td>
</tr>
<tr>
<td>GMP</td>
<td>Group Membership Protocol</td>
</tr>
<tr>
<td>GMS</td>
<td>Group Membership Service</td>
</tr>
<tr>
<td>HA</td>
<td>High Availability</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SOAP</td>
<td>Historically, Simple Object Access Protocol. Now simply SOAP</td>
</tr>
<tr>
<td>S3</td>
<td>(Amazon) Simple Storage Service</td>
</tr>
<tr>
<td>WKA</td>
<td>Well-known Address, Well-known Addressing</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction
Cloud computing is a field which is gaining popularity. Cloud computing is an alternative to having local servers or personal devices handling applications. Essentially, it is an idea that the technological capabilities should "hover" over the infrastructure and be available whenever a user wants. Instead of utilizing the power of one or a few devices, it is much more economical and convenient to utilize the computing power of a cloud configuration. Companies such as Amazon [1] & Google [2] have started to provide services for hosting applications in their respective clouds. The infrastructure does not have to be maintained by the companies hosting the applications and also there are other value additions such as security. This is a very appealing idea for mostly small & medium companies and even for large companies. Generally, they only have to pay for the computing power consumed and the network bandwidth utilized. This idea has started to appeal not only for small & medium companies but even for large and well established companies. According to [3] some of the biggest customers of Amazon Web Services in both number and amount of computing resources consumed are divisions of banks, pharmaceuticals companies and other large corporations.

Fault tolerance, high availability & scalability are essential prerequisites for any enterprise application deployment. The traditional approach towards achieving scalability has been anticipating the peak load, and then purchasing and setting up the infrastructure that can handle this peak load. However, the system will not operate at peak load most of the time, hence it is obvious that this approach is not the most economical choice. It is intuitive that the most economical approach would be to scale-up the system as the load increases and scale-down the system when the load decreases. This is called auto-scaling. However, such an approach will not be possible without virtualization. This is where cloud computing becomes very useful. Amazon Elastic Compute Cloud (Amazon EC2) [1] is a Web service that provides resizable compute capacity in a cloud computing environment. It is designed to make web-scale computing easier for developers. Since Amazon provides guarantees of security & takes over the responsibility of infrastructure maintenance, it is very economical for mostly small & medium scale companies, and even some large scale companies to host their applications on EC2 instead of taking on this burden.
Web services technology has become the de-facto standard for integrating autonomous entities. Therefore, the availability & scalability requirements have also become essential for Web services.

Apache Axis2 [4] [5] is a middleware platform which enables hosting of Web service applications and supports some of the major Web services standards. Multiple EC2 instances will need to host Axis2 nodes to provide these guarantees. When the load increases, we should be able to start up new nodes to satisfy the scalability guarantees. However, when the load decreases, the idle nodes should be shutdown. This is because the fees charged by Amazon is based on the number of compute units consumed. A compute unit is equivalent to running a CPU with a particular computing power for one hour. We can achieve an optimal balance between the cost and performance using this autoscaling & auto-healing approach. Autoscaling refers to scaling up or down depending on the load whereas auto-healing refers to automatically recovering the system to a specified minimum deployment configuration. This will be a very appealing idea from a business perspective. We will be looking in detail how Apache Axis2 Web services can be autoscaled on Amazon EC2.

1.1 Amazon Elastic Compute Cloud (EC2)

Amazon EC2 [1] is a Web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. Amazon EC2 provides a simple Web service interface & tooling that allows one to obtain and configure capacity with ease & to build failure resilient applications. Amazon EC2 reduces the time required to start new server instances to minutes, allowing you to quickly scale capacity, both up and down, as the computing requirements change. One only needs to pay for the computing capacity actually used. This is a small amount compared to the significant up-front expenditures traditionally required to purchase and maintain hardware. This frees the user from many of the complexities of capacity planning and transforms what are commonly large fixed costs into much smaller variable costs, and removes the need to over-buy "safety net" capacity to handle periodic load spikes. The user has full control over the setup, including firewall
configuration. EC2 provides a better economic model for large-scale applications; the owner of the application only pays for usage, while benefiting from the reliability of the Amazon infrastructure. There are different types of EC2 instances [6] which can cater to varying computing & memory capacity requirements.

1.2 EC2 Features for Building Failure Resilient Applications

Recently Amazon announced two new features; Availability Zones & Elastic IP Addresses, which make it much more possible to and easier to build fault resilient applications.

**Availability Zones** [7] - One of the key requirements of highly available & highly assured systems is that the application should be available despite the failure of a few components. Clustered deployments is one of the common solutions to address these requirements. It is also common for a cluster of nodes to be deployed at a single location such as a data center. What if this data center is destroyed by fire or floods? What if the entire data center looses power? How can we satisfy the availability guarantees in case of such a scenario? To ensure availability despite these catastrophic failures, the replicas of the application should be located on separate infrastructure and preferably on separate physical locations.

Amazon has already thought about this and recently announced *Availability Zones*. The point of availability zones is the following: if we launch a server in zone A and a second server in zone B, then the probability that both go down at the same time due to an external event is extremely small. This simple property allows us to construct highly reliable Web services by placing servers into multiple zones such that the failure of one zone doesn’t disrupt the service or at the very least, allows us to rapidly reconstruct the service in the second zone. Availability Zones allow the customer to specify in which location to launch a new EC2 instance. The world is divided up into Regions and a Region can hold multiple Availability Zones. These Zones are distinct locations within a region that are engineered to be insulated from failures in other Availability Zones and provide inexpensive, low latency network connectivity to other Availability Zones in the same region. By launching instances in separate Availability Zones, applications can be protected from failure of a single location. Each
availability zone runs on its own physically distinct, independent infrastructure, and is engineered to be highly reliable. Common points of failures like generators and cooling equipment are not shared across availability zones, and availability zones are designed to be independent with failure modes like fires and flooding. For inter-availability zone data transfers, there is no charge, whereas for intra-availability zone data transfers, there is a $0.01 per GB in/out charge.

**Elastic IP Addresses** [8] - Elastic IP addresses are static IP addresses designed for dynamic cloud computing. Elastic IP addresses are associated with a customer account and allow the customer to do its own dynamic mapping of IP address to instance. Using this dynamic mapping, applications can remain reachable even in the presence of failures. An Elastic IP address is associated with your account, not a particular instance, and you control that address until you choose to explicitly release it. Unlike traditional static IP addresses, however, Elastic IP addresses allow you to mask instance or availability zone failures by programmatically remapping your public IP addresses to any instance associated with your account. However, if the Elastic IP address is used $0.01 per GB in/out will be charged. Therefore, when communicating within an availability zone, it is always best to use the private IP.

### 1.3 Apache Axis2

Axis2 [4] [5] is a middleware platform which enables hosting of Web service applications and supports some of the major Web services. Apache Axis2 supports clustering for high availability & scalability. In fact, clustering is a major value addition provided by Axis2 for an enterprise deployment. Fault tolerance, high availability & scalability are essential features for such an enterprise deployment. The state stored in the context hierarchy [5] of a node can be replicated to the other members & the cluster of nodes can be centrally managed. The default clustering implementation is based on Apache Tribes [9] which is a robust Group Communication Framework (GCF) [10] [11]. However, this clustering implementation does not work on Amazon EC2 since this implementation discovers the group members based on membership multicast. Multicasting has been disabled on EC2 and also there is no guarantee that the Amazon Machine Images (AMI) [1] instances will belong to the same multicast domain. Hence we have developed a different membership discovery mechanism which can be plugged into Tribes. This is explained in detail in the section titled “Well-known Address (WKA) Based Membership”.

- 5 -
Axis2 also supports plugging-in different clustering implementations which can be based on different Group Communication Frameworks (GCFs). Stateful as well as stateless services can be deployed on Axis2. In the case of stateless services, it helps to have multiple processing nodes to provide scalability guarantees, whereas in the case of stateful services, multiple replicas are necessary to provide high availability guarantees.

1.4 Apache Synapse Dynamic Load Balancing

Apache Synapse is designed to be a simple, lightweight and high performance Enterprise Service Bus (ESB) [12]. It has been designed to be simple to configure, very fast [13] [14] [15], and effective at solving many integration and gatewaying problems. Load balancing is one of the gatewaying problems addressed by Synapse. There are several modes in which load balancing can be carried out using Synapse. However, in this project, we will be only using the dynamic load balancing capabilities of Synapse. In a typical load balancer, the load balancer fronts a cluster of application nodes. The application nodes are the ones which carry out the work. In such a setup, traditionally, connection information of each node in the application cluster has to be provided using a configuration file. However, in dynamic load balancing, such static configuration is not carried out, and membership of the application cluster is dynamically discovered.

1.5 The Problem

Hosting applications on EC2 is a very appealing idea. In a typical clustered configuration, a group will consist of a set of nodes fronted by a load balancer. In most setups, a single load balancer will front two worker nodes. However, when the load increases, it would be best if the load is distributed over a larger number of worker nodes. The traditional solution is to gather some metrics on the peak load, and then compute the number of nodes required to handle this peak load. These nodes will then be arranged according to the deployment architecture in a data center. However, the system will be under peak load only for a fraction of the time of operation, hence most of the resources will be idling. A similar deployment can be setup on a cloud where the number of instances in service is fixed. However, the cloud is inherently
elastic, hence it does not make sense to have such a deployment. Also having many nodes means that network bandwidth is consumed since membership management messages will be sent back and forth between the nodes, even if they are not doing any useful work. It is going to be uneconomical on EC2 since the payment is dependent on the number of computing units & network bandwidth consumed, and more live instances means a higher payment. Hence the user preference would be to start new nodes when the load increases & to shutdown idling nodes when the load drops. Ideally, the number of live nodes should be proportional to the current load. This way, we can get optimal performance as well as optimize the price paid to Amazon. This idea should be quite appealing for enterprise customers since it gives the best of both worlds, i.e. performance as well as price. In addition, we need to provide all the fault tolerance guarantees that are available for such a deployment.

1.6 Objectives

The objectives of this project include building a framework which will autoscale i.e. scale up or down, the number of Axis2 nodes on Amazon EC2, depending on the load. The system load is represented by the length of a list of messages in flight.

![Diagram of Scale-up](image)

**Figure 1: Scale-up when the system load increases**

If the load increases, the length of the message list will be relatively large, at which point the system will be scaled-up by starting one or more EC2 instances which will join the application group as shown in Figure 1.
If the load increases, the length of the message list will be relatively small, at which point the system will be scaled-down by terminating some EC2 instance as shown in Figure 2.

1.7 Prior Work

GigaSpaces [16] is a solution for running stateful applications in the Amazon EC2 environment. The idea they propose is quite similar. A GigaSpaces application should follow a specified architecture to make it possible to deploy scalable applications on EC2. The main drawback in a GigaSpaces application is that the developers have to program to the GigaSpaces APIs, and are bound to those APIs. However, in the approach suggested in this thesis, there is no coupling to any special APIs as far as the Web service developers are concerned. They would develop the services as usual using the usual tools, test them, and deploy them on the cloud middleware we provide, without any change to their code. Our middleware will take care of automatically scaling those services, and that is completely transparent to the service developer.

There has not been much prior work in the area of high availability for Web services. In fact, distributed middleware technologies such as CORBA address high availability & scalability at the specification level, but Web services do not. This concern also been expressed by Birman [10]. According to Birman Web services Reliability standards do not talk about making services reliable, it defines rules for writing reliability requests down and attaching them to documents. In contrast, CORBA fault-tolerance standard tells how to make a CORBA service into a highly available
clustered service [10]. In other words, this is a major shortcoming in the area of Web services specifications. However, even though high availability aspects have not been addressed, newer standards such as WS-ReliableMessaging [17] take into account several aspects such as retrying, in-order, at-most-once delivery, and persistence. This work tries to address some of these shortcomings in this area.

The Apache Tribes [9] group communication framework (GCF) based clustering implementation for Axis2 also is a very recent development.
Chapter 2

Literature Review
2.1 Terminology

When discussing distributed systems, there are a number of terms which are heavily used. It would be helpful to define some of these terms at the beginning of this thesis so that the reader has a good understanding before going in deeper.

*Distributed System* – A set of processes that communicate by message passing and carry out some desired functions over a period of time [10]

*Fault Tolerance* – The ability of a distributed system to recover from failures without performing incorrect actions. The system remains correct in spite of failures. [11]

*High Availability* – A system that is designed for continuous operation in the event of a failure of one or more components. However, the system may display some degradation of service, but will continue to perform correctly. The proportion of time during which the service is accessible with reasonable response times should be close to 100%. [11]

*High Assurance* - Highly assured systems provide guarantees such as rapid restart of components or rapid rollover from failed components to healthy ones while continuing to provide services, and behaving correctly [10]

*Scalability* – The ability of a system to continue to operate correctly even when it is scaled to a larger size. In the context of this project, the system should continue to operate correctly even when the system scales up or scales down. [10]

*Recoverability* – The system can repair itself after certain components have crashed. This can be generally done by restarting failed components. This is not the same as High Availability. [10]

*Replication* – In a set of identical nodes that belong to a single group, changes in the state of a particular node will be reflected on the other nodes in the group. Server replication can refer to the replication of a server instance [11]

*Fail over* - Transfer of operation from a failed node to a similar, redundant node to ensure uninterrupted operation. This is generally transparent to the user. [11]
**Reliable Process Group Architecture** – This is an architecture where a process group is defined by a series of “views” of its membership. All members see the same sequence of view changes. A view change occurs when a member either joins the group, leaves the group, or is deemed to have left the group by the group membership service. Such an architecture is essential for providing high availability. [10]

**Cloud Computing** – Cloud computing is a general term for anything that involves delivering hosted services over the Internet. These services are broadly divided into three categories; Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS). A cloud service has three distinct characteristics that differentiate it from traditional hosting. It is sold on demand, it is elastic i.e., a user can have as much or as little of a service as they want at any given time, and the service is fully managed by the service provider.

**Web services** – The World Wide Web Consortium (W3C) definition of Web services is "a software system designed to support interoperable Machine to Machine interaction over a network" [18]. Birman specifies a Web service as “software components described via WSDL which are capable of being accessed via standard network protocols such as SOAP over HTTP.” [10]

**Middleware** - Software systems that sit in the middle between end-user applications & the communication infrastructure for basic message passing and support for reliable channels. Axis2 is such a middleware system.

### 2.2 Theoretical Aspects

In this section, we will take a look at some of the theoretical aspects of process failures and some of the techniques employed to overcome these failures.

#### 2.2.1 Failure Detection

Reliably detecting failures in an asynchronous set up is a theoretical impossibility [11]. This is mainly because we cannot set an upper bound on time in such an environment. In [19] Vogels argues that we do not assume that some person is dead just because he does not answer the phone, we will do some background investigation
like asking his landlord or neighbors, checking whether the electricity & water is being consumed and so on. The point is, timeouts at the transport layer are not always sufficient for failure detection. We can consult other data sources such as the Operating System where the process runs, use information about status of network routing nodes or can augment with application-specific solutions. In other words, some sort of external failure detection mechanism would generally provide better results. Anyway, this will not detect processes that seem to be UNSUSPECTED but are not working correctly. In other words, Byzantine failures cannot be detected [11] [10].

A robust Group Communication Framework can utilize these techniques to detect process failure with a higher degree of certainty which will suffice in all practical situations. However, such a setup will take longer to detect a failure due to the overhead involved [11]. Therefore, we may have to live with a trade off; *Unreliable Failure Detectors*. Such a failure detector will set the status of a process to SUSPECTED, if a heartbeat is not received from the process within a certain time interval. Also, if the process fails soon after sending the heartbeat, the failure detector will still continue to classify the process as UNSUSPECTED.

Failure detection models can be broadly classified as synchronous models and asynchronous models.

1. **Synchronous**

   These can be viewed as extreme cases of the spectrum. The *synchronous model* makes several, often impractical, assumptions. The value of this model is that if a problem cannot be solved using this model, there is a high probability that the same problem cannot be solved in a more realistic environment. So we can figure out the impossibility results.

2. **Asynchronous**

   The value of the asynchronous model is that it makes no assumptions on time. Furthermore, it is stripped down and simple so that if something can be done well in using this model, it can be solved at least as well in a more realistic environment. Thus we can determine the best and most correct solutions using this model. In the
synchronous models, when some thing is said to be “impossible” it generally means that “it is not possible in all cases”, hence is more mathematically correct.

2.2.2 Overcoming Distributed System Failures

The general approach for overcoming failures in a distributed system is to replace critical components with group of components that can each act on behalf of the original one. This is the approach we are following in the EC2 deployment. But the most difficult thing to accomplish in such a setup is keeping the states across the replicas consistent and getting all replica nodes to agree on the status. In fact, this is one of the major challenges we will be facing, and will need to investigate how different Group Communication Frameworks perform on EC2.

Redundancy is the widely adopted technique when it comes to fault tolerance. In fact, it is not wrong to claim that the only known technique of achieving fault tolerance is redundancy. Redundancy in terms of hardware, software, time or information are commonly used techniques. In this project, hardware and software redundancy will be employed. Multiple EC2 nodes correspond to separate hardware systems as well as multiple identical software instances. These multiple units can be employed for scalability, in which case, they will not be considered redundant. These units can also be used to achieve high availability and failover, in which case, some of the units can be considered redundant. In a fault tolerant system, critical components, critical data & critical system state are replicated for availability. However, there needs to be proper coordination when it comes to replicating data & state. Server side replication introduces many complexities. In some setups, the client needs to be aware of the fact that there are several replicas of the server and if a failure occurs, needs to direct its request to a different replica. Then again, how will the client know whether the original failed request was properly processed on the server before it crashed or not? The logical conclusion is to make these failures transparent to the client, and if a failure occurs, a replica will take over. Obviously, this transparency has its own overhead. In practice, we have found that it is not possible to make server side failures completely transparent to the client. In the current Axis2 implementation, after several retries, if the state cannot be replicated to all the member nodes on the server side, the operation is aborted and an error is reported to the client. The client is expected to retry when such an error occurs. In this project implementation, we have redundancy
at the load balancer level and will try to make failure as transparent as possible to the client, but when the primary load balancer fails the clients will recognize this failure and will have to retry.

### 2.2.3 High Availability

A highly available system remains operational during failure. Some of the components may have failed but the system will continue operate correctly even if there is a certain level of degradation in the quality of service provided. The system may be able to automatically heal itself by bringing into operation some other components that were kept standby. In such a system, the proportion of time during which the service is accessible with reasonable response times should be close to 100%. The main challenge in developing such a system is the replication of data needed for the continued operation. The greater challenge is while enabling replication, also maintaining acceptable performance when the number of replicas increases. Indeed this is going to be a major challenge on EC2, especially when the replicas are distributed across availability zones [7] on different regions. In most practical replication systems, Byzantine failures [11] have been ignored, since failure models for handling Byzantine failures are quite costly, since 3t+1 processes and t+1 protocol runs are needed to reach consensus [11], where t represents the number of traitors or number of failed processes. When a member is not contactable, it is deemed to have crashed and when a message is delivered to a member, the member is trusted to behave correctly [11].

In the default Axis2 clustering implementation, when a state replication message is sent, if a single correct process delivers the message and this message is properly processed, the initiator will deem the replication successful. Else, the initiator will retry several times and finally abort and report a failure to the client. The system provides the strongest guarantees. However, this will have a major impact on performance when the system scales up, hence we will have to investigate whether other mechanisms with weaker but acceptable guarantees exist.
2.2.4 Group Membership Service (GMS)

The GMS is an important component of a reliable process group architecture [10]. Internally, the GMS runs a Group Membership Protocol (GMP). Ideally, a GMS will have just a small number of members. This set of nodes will track membership for a large number of system processes. A Group Communication Protocol (GCP) is used amongst these nodes. But in GMSs like Apache Tribes [9], all nodes in the group become part of the GMS. The system membership list, which includes nodes which are not part of the GMS, is replicated list which is managed by GMS members. When membership changes, this list is updated using multicast. Since multicast is not possible on EC2, we will have to implement a point-to-multipoint GMS. To move from one view (say view $V_1$) to another (say view $V_2$), the GMS requires explicit acknowledgment by a majority of the processes in the current view (i.e. view $V_1$). A process group is defined by a series of views of its membership. All members see the same sequence of view changes. Events such as node failures & new nodes joining the group are notified by changing membership, which are sent as view change messages.

2.2.5 Data Replication Models

Replication is the key to providing High Availability and Fault Tolerance. In this section we will look at some widely used replication models.

Passive Replication

![Figure 3: Passive Replication](image)

This is also called primary-backup replication. The clients will communicate only with the primary. The primary executes the client requests and replicates the state changes to the backups. If the primary fails, a backup will take over the role of the primary. This is depicted in Figure 3. An election may take place to select the new
primary. The response to the client can be sent after all the backups have agreed with the change, or in order to improve performance, the primary can send periodic updates to the backups. In the latter case, there is a possibility of data loss. This situation may be acceptable for some applications, or may be handled at an upper layer. [14]

**Active Replication**

![Figure 4: Multicast from client](image)

![Figure 5: Replies from replicas](image)

As shown in Figure 4, the clients sends a totally ordered multicast the request to the group of replica managers. Each of the replica manager's replies back to the client, as shown in Figure 5. The client can do a comparison and take the majority result as the answer. In this model, Byzantine failures can be dealt with since the client can consider the majority result [11].

### 2.2.6 Virtual Synchrony

According to the principles of virtual synchrony, all multicasts take place between view changes. All multicasts that are in transit take place before the view change comes into effect. In this model, all processes see a set of events in the same order. There are two types of events; messages sent to the group and membership changes [10]. There are many real-world systems that use this model, and achieve extremely high performance.
2.2.7 Membership

1. Static Membership

In this scheme, only a defined set of members can be in a group. The GMS will detect members joining or leaving the group. External members cannot join the group. Each node may obtain the group member details from a central repository.

2. Dynamic Membership

In this scheme, the membership is not predefined. Members can join a group by specifying the appropriate group name, and also can leave the group. The GMS will detect new members joining or leaving the group. The group membership information may be obtained from the GMS.

3. Hybrid Membership

In this scheme, there is a set of well known members. We can consider these members as belonging to a static group. External members can join this group by notifying one of the well known members. These external members can then get the current group membership from this well known member. When new members join the group, the well known members will notify all other members. When members leave the group, the GMS can detect this event. This project will implement such a scheme.

2.2.8 Load Balancing

One of the most common cases for using a process group is for the purpose of load balancing or scaling out [10]. The group members share the load thereby maintaining acceptable response times even when the system load increases. However, all members may not share the load equally.

Several load balancing algorithms are employed in practice;

1. Multicasting a client request to all members

   The client request is multicast to all members and they all process the request. The server functionality should be deterministic for all members to be in the
same state. This more a case of availability rather than scalability since every node will have to process every request.

2. Client chooses the member to send the request to

The client is aware of the server side configuration and chooses a server to send the request to. This may not scale well since some clients may choose to invoke a particular node more often thereby overloading this node, while some other nodes may be idling. The client may also become more complex since it will have to implement a fail-over policy which it reissues a request if the original server node the request was sent to turns out to be faulty.

3. Carry out the processing on one member and replicate the server state to the others

This is the most widely used configuration in practice. Such a setup can support availability as well as scalability requirements. This is the default configuration used in Axis2 clustering.

4. Sticky configuration

All requests of a particular client for a particular session are always sent to the same node. This can be implemented using a sticky session implementation. The client is expected to send back the session ID with each request, and the load balancer keeps track of the mapping between the session ID and the node ID. This also may not be an effective load balancing mechanism since some clients may overwhelm their respective server side nodes, whereas some nodes may lie idle. However, managing the state for a particular client becomes much simpler.

2.2.9 Reliability & Web services

According to Birman Web services Reliability standards do not talk about making services reliable, it defines rules for writing reliability requests down and attaching them to documents. In contrast, CORBA fault-tolerance standard tells how to make a CORBA service into a highly available clustered service [10]. In other words, this is a major shortcoming in the area of Web services specifications. However, even though
high availability aspects have not been addressed, newer standards such as WS-ReliableMessaging take into account several aspects such retrying, in-order, at-most-once delivery, and persistence.

### 2.2.10 Group Communication Frameworks (GCF)

There are several popular GCFs and GCF toolkits which abstract the complexities of reliable group communication. However, many of the existing GCF do not scale too well. Birman reports that JGroups [20], Ensemble [21] and Spread [22] can scale up to a maximum of 50 to 75 members. After this threshold, the performance of the protocol drops exponentially. Apache Tribes is very similar in architecture and functionality to JGroups, hence we suspect that the same limitations are applicable. However, for most practical system, this limit is acceptable. However, the performance would be unacceptable for massively scalable systems.

### 2.3 Amazon EC2

#### 2.3.1 Using EC2

To use Amazon EC2;

1. Create an Amazon Machine Image (AMI) containing the applications, libraries, data and associated configuration settings. Or use preconfigured, templated images to get up and running immediately.
2. Upload the AMI into Amazon S3. Amazon EC2 provides tools that make storing the AMI simple. Amazon S3 provides a safe, reliable and fast repository to store the AMIs.
3. Use Amazon EC2 web service to configure security and network access.
4. Start, terminate, and monitor as many instances of the AMI as needed
5. Pay only for the resources actually consumed (instance-hours or data transfer)

A running instance could have 3 types of IP addresses;

1. Private IP Address - Can be used only within an EC2 region
2. Public IP Address: - Internet routable IP address. Public IP Addresses are no longer usable upon instance termination.
3. Elastic IP Address - Elastic IP Addresses are allocated to accounts and can be remapped to other instances when desired.

2.3.2 Instance Types

There are 3 types of EC2 instance types [6]

Table 1: EC2 Instance Types

<table>
<thead>
<tr>
<th>Instance</th>
<th>Memory (GB)</th>
<th>Computing Capacity (ECU)</th>
<th>Storage (GB)</th>
<th>Platform</th>
<th>Price ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>1.7</td>
<td>1 (1 virtual core with 1 ECU)</td>
<td>160</td>
<td>32-bit</td>
<td>0.10</td>
</tr>
<tr>
<td>Large</td>
<td>7.5</td>
<td>4 (2 virtual cores with 2 ECU each)</td>
<td>850</td>
<td>64-bit</td>
<td>0.40</td>
</tr>
<tr>
<td>Extra Large</td>
<td>15</td>
<td>8 (4 virtual cores with 2 ECU each)</td>
<td>1690</td>
<td>64-bit</td>
<td>0.8</td>
</tr>
</tbody>
</table>

One *EC2 Compute Unit (ECU)* provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. Pricing is per instance-hour consumed for each instance type. A customer will select an instance type based on the computing power & storage required.

2.3.3 Features for Building Failure Resilient Applications

Recently Amazon announced two new features; Availability Zones & Elastic IP Addresses, which make it much more possible to and easier to build fault resilient applications.

1. Availability Zones

One of the key requirements of highly available & highly assured systems is that the application should be available despite the failure of a few components. Clustered deployments is one of the common solutions to address these requirements. It is also common for a cluster of nodes to be deployed at a single location such as a data center. What if this data center is destroyed by fire or floods? What if the entire data center looses power? How can we satisfy the availability guarantees in case of such a scenario? To ensure availability despite these catastrophic failures is that replicas of the application should be located on separate infrastructure and preferably on separate physical locations.

Amazon has already thought about this and recently announced *Availability Zones*. The point of availability zones is the following: if we launch a server in zone A and a second server in zone B, then the probability that both go down at the same time due
to an external event is extremely small. This simple property allows us to construct highly reliable web services by placing servers into multiple zones such that the failure of one zone doesn’t disrupt the service or at the very least, allows us to rapidly reconstruct the service in the second zone. Availability Zones allow the customer to specify in which location to launch a new EC2 instance. The world is divided up into Regions and a Region can hold multiple Availability Zones. These Zones are distinct locations within a region that are engineered to be insulated from failures in other Availability Zones and provide inexpensive, low latency network connectivity to other Availability Zones in the same region. By launching instances in separate Availability Zones, applications can be protected from failure of a single location. Each availability zone runs on its own physically distinct, independent infrastructure, and is engineered to be highly reliable. Common points of failures like generators and cooling equipment are not shared across availability zones, and availability zones are designed to be independent with failure modes like fires and flooding.

For intra availability zone data transfers, there is no charge, whereas for intra-availability zone data transfers, there is a $0.01 per GB in/out charge [1].

2. Elastic IP Addresses

Elastic IP addresses are static IP addresses designed for dynamic cloud computing. Elastic IP addresses are associated with a customer account and allow the customer to do its own dynamic mapping of IP address to instance. Using this dynamic mapping, applications can remain reachable even in the presence of failures. An Elastic IP address is associated with your account, not a particular instance, and you control that address until you choose to explicitly release it. Unlike traditional static IP addresses, however, Elastic IP addresses allow you to mask instance or availability zone failures by programmatically remapping your public IP addresses to any instance associated with your account. However, if the Elastic IP address is used $0.01 per GB in/out will be charged [1]. Therefore, when communicating within an availability zone, it is always best to use the private IP.

2.3.4 Data Transfer Charges

[1] provides the data transfer charges. When nodes in the cluster are deployed on different availability zones within the same region, there is a charge of $0.01 per GB
in/out. If the availability zones are in different regions, then the normal Internet data transfer rates will be applicable. Hence when the cluster is deployed across multiple regions for higher fault tolerance there will be a higher charge.

2.4 Giga Spaces

GigaSpaces [23] provides a general purpose solution for running stateful applications on Amazon EC2 & tries to address several challenges that hinder performance and scalability. This application is also trying to address a similar problem; stateful applications that face particular challenges in horizontally scaling across a pool of machines. GigaSpaces can also scale up and down depending on load, but it is not clear from the literature how the scaling up or down is going to affect response times and throughput. It also provides a management interface of hosted applications called the GigaSpaces Management Center, which is similar to the cluster management functionality available in Axis2. However, this is not a replicated application like the Axis2 management application, hence can become unavailable if the node running the Management Center fails.

GigaSpaces is a proprietary application, where as Axis2 is open source. One of the drawbacks of writing an application which is to be deployed on GigaSpaces is that the developer has to be aware of GigaSpaces and use the APIs provided by them to develop the applications. Hence, there need to be some code changes if an application is to be migrated to GigaSpaces. In the case of Axis2, the Web service developer is unaware of the clustering implementation and can simply deploy a service which was developed in a clustering unaware manner on a cluster. If the developer has properly stored all the state information that needs to be replicated across the cluster, by placing it in the proper Axis2 context [5], the clustering framework will take care of replicating this state transparent to the users.

[16] discusses about using GigaSpaces for Web Services. The manner in which state is maintained and replicated in quite different from the Axis2 programming and development model and also as mentioned earlier, the code will be tied with the GigaSpaces APIs to a certain extent. One of the major differences when comparing Axis2 with GigaSpaces is that, Axis2 Web service authors do not have to be aware of the fact that the service is going to be deployed in a clustered environment.
Chapter 3
Methodology
In this section, we will introduce the concepts of multicastless membership discovery, dynamic load balancing, interacting with the Amazon EC2 Web service, a load analysis algorithm with autoscaling & autohealing cluster. Later, we will look at how the setup has to be deployed on Amazon EC2, the relevant configurations and other related aspects.

3.1 Introduction

The current Axis2 clustering implementation uses multicast based membership discovery. However, Amazon has restricted multicasting on the EC2 cloud, and also there is no guarantee that the nodes will be added to the same multicast domain. Preliminary investigations revealed that multicast-based discovery schemes do not work on EC2. Hence, we first had to implement a Well-Known Address (WKA) based discovery scheme. In such a scheme, a Membership Manager will be available at a known IP address. This IP address is an Amazon Elastic IP address. The implementation is generic, but in the context of an EC2 setup, the well-known member also happens to be the primary load balancer. When a new application or load balancer node joins the cluster, it will first try to connect to a well-known address, and register itself. Then it will obtain the list of current members from the Membership Manager.

Each new member will register with this Membership Manager running on the well-known member and will receive membership change notifications. The Membership Manager will track the group membership view changes. The well-known primary load balancer member has at least one backup. If this primary load balancer fails, a backup will take over from the primary load balancer, and will also assume the IP address of the previously well-known load balancer, thus becoming the well-known member. Hence the system continues to be available & operate correctly even in the case of well-known member failure.

A Load Analyzer will also run on the primary load balancer. This analyzer will periodically observe the load on an application cluster & make scale-up or scale-down decisions. If the perceived load seems to be high over a period of time, this task will
decide to scale-up the cluster. Similarly, if the perceived load is low over a period of time, this task will decide to scale-down the cluster.

Another task will periodically run and check whether the cluster conforms to the specified minimum configuration. If this is not met, it will start up new EC2 instances so that this minimum configuration is guaranteed at all times.

3.2 Well-known Address (WKA) based membership

Membership discovery & management are essential features of any clustered deployment. There are several widely used membership schemes such as static membership & dynamic membership. In a static scheme, the members in the cluster are preconfigured; new or unknown members cannot join the group. In dynamic schemes, new members can join or leave the group at any time. For an Amazon EC2 cluster which autoscales, we require a dynamic membership scheme. In such a scheme, the traditional manner in which membership is discovered and managed is based on IP multicast. In such an approach, when a new member joins the group, it will send a MEMBER_JOINED multicast to a designated multicast socket. All members in a given group should use the same multicast IP address and port. Subsequently, this member will multicast heartbeat signals. All members will periodically multicast heartbeat signals. The failure detectors running on all members will keep track of the heartbeats and if a heartbeat is not received from a member within a specified interval, will categorize this member as suspected. Once again, when the heartbeat is received, this member will be categorized as available.

On a cloud computing environment such as Amazon EC2, multicasting is generally not possible. This is because the IP addresses for each image instance is automatically assigned by Amazon, and generally these will not belong to the same multicast domain. Also, multicasting will not be possible across Amazon's availability zones. However, Amazon has explicitly disabled multicasting on its networks for obvious reasons.
We introduce a well-known address (WKA) based membership discovery & management scheme which can be used on environments where multicasting is not possible. In this setup, there are one or more members which are assigned with well-known IP addresses. All other members are aware about these well-known members. At least one well known member should be started up before any other member. It is also possible to assign a well-known address to the member which started up first. An elastic IP address can be assigned to this first member. When other members boot-up, they will try to contact one or more well-known members by sending a JOIN message. The well-known member will add this new member to its membership list, notify all other members about this new member joining by sending a MEMBER_JOINED message to the group, and will send the MEMBER_LIST message to the newly joined member. Now, all group members will be aware about this new member joining, and the new member will learn about its group membership. This is illustrated in Figure 6.

The possibility of one or more well-known members failing is always there. In such
an event, when the well-known member rejoins, it would have lost all knowledge about the membership. However, when a well-known member rejoins, all other members can detect this event. The other members will notify about the group membership by sending MEMBER_LIST messages to the newly rejoined well-known member. This is illustrated in Figure 7.

One drawback in this setup is that if all well-known members fail, new members will not be able to join the group. However, the existing members will continue to function correctly. Members failing subsequently can also be detected since each member watches out for a TCP ping heartbeat from all other known members. On Amazon EC2, the elastic IP address can be remapped to another member within seconds, hence the cluster recovers very quickly. Therefore, this deployment will continue to function correctly at all times.

3.3 Fault Resilient Dynamic Load Balancing

Static load balancing involves a static application group. The details of such a group are provided using a static configuration file. The shortcoming of this traditional approach is that members outside this group cannot join, and the IP addresses of such members should be known beforehand. Clearly, this approach is not suitable for deployment on EC2. Using group communication mechanisms, dynamic load balancing can be achieved. For fault resilience and avoiding single points of failure, the load balancers also need to be replicated. The load balancer can handle multiple application groups. Application members can join or leave their respective groups. The load balancers should be able to detect these membership changes. Hence, the load balancers need to be part of the application groups as well as shown in Figure 8. In this setup, the load balancers will be aware of membership changes in the load balancer group as well as the application groups. But the application group members (i.e. members in Group A & Group B) will not be generally aware of membership changes in the load balancer group.
Figure 8: Load balancer & application groups

The exception is when a load balancer member also happens to be a well-known member. In such a scenario, all members in all groups will be able to detect the well-known load balancer member leaving the group or rejoining the group.

Figure 9: Active-passive load balancers with Elastic IP

The Web service clients will only be aware of the elastic IP address of the load balancer, in this case 75.101.131.185 (shown in Figure 9), and will send their requests to this IP address. This load balancer node, LB1, also acts as a well-known member, hence membership management will revolve around LB1. Elastic IPs provided by Amazon can be remapped to a different Amazon Machine Instance (AMI) at any time. In the case of LB1 failing, this even will be automatically detected by the passive LB2 load balancer member which will remap the elastic IP to itself. So this setup is self-
healing. Client requests that were sent at the exact moment of LB1 failing, or before the responses were sent by LB1, will fail. However, subsequently, clients can retry and continue. Application membership will also continue to work once the elastic IP has been remapped.

![Membership Channel Architecture](image)

**Figure 10: Membership channel architecture**

All load balancer members are connected to all membership channels of all groups as illustrated in Figure 10. Hence all load balancers can detect membership changes in all groups, including the load balancer group. The application members are only connected to the membership channels of corresponding to the respective groups.
State replication messages & member initialization messages will only be sent within the respective groups, hence there will be no interference. This is because there are separate channels corresponding to the different groups for this purpose, as illustrated in Figure 11.

As shown in Figure 12, LB1 is a well-known member and another application member, AppMember(A) which belongs to group A joins. In this scenario, LB1 which is a well-known member, also happens to be a load balancer. A new application member, AppMember(A) which belongs to Group A, initiates a join by sending a JOIN message to the well-known LB1 member via Group A's membership channel.
Next, LB1 will notify all members in Group A as well as all the members in the load balancer group, Group LB, about this new member joining by sending MEMBER_JOINED messages to these groups via the respective membership channels of these groups. Finally, LB1 will send the MEMBER_LIST(A) to AppMember(A) notifying AppMember(A) about its group membership via Group A's membership channel. When AppMember(A) receives this message, it will notice that the well-known member does not actually belong to its group, hence will remove LB1 from its Group membership list. Therefore, state replication messages in Group A will not be sent to LB1. However, AppMember(A) will be able to detect the failure of LB1, but will not send any state replication or group management messages to LB1.
As shown in Figure 13, when a load balancer (LB2) which does not have a well-known address joins, it will send the JOIN message to a well-known peer load balancer (LB1) via the membership channel of Group LB. Now LB1 will notify the load balancer group (Group LB) about the new load balancer member LB2 joining by sending a MEMBER_JOINED(LB2) message to the load balancer group, Group LB.

The newly joined load balancer member, LB2, does not have knowledge about the membership of the load balancer group (Group LB) or the application domain groups (Group A, Group B up to Group n), in this case. Hence, the well-known load balancer, LB1, will inform about the memberships of the different groups by sending a series of messages; MEMBER_LIST(LB) followed by MEMBER_LIST(A), MEMBER_LIST(B) up to MEMBER_LIST(n) via the respective membership channels. Once this process has been completed successfully, the new load balancer member LB2 would be deemed to have successfully joined.
In the event of a well-known member failing, the setup can continue to function correctly. However, new members cannot join the different groups. This is true for members in the load balancer group, Group LB, as well as all application groups. We need to note that members leaving the respective groups will be detected by all members in the respective groups. This will be further discussed in the Failure detection section. However, when the well-known member becomes available, the groups can continue to grow, since new members can once again join. A recovery procedure takes place when a well-known member rejoins after failing. When a well-known member (LB1 in this case) rejoins, all members in all groups can detect this. Once they detect LB1 rejoining, they will inform LB1 about the group membership of their respective groups by sending MEMBER_LIST messages via the respective membership channels.

The message exchange sequence for this scenario has been shown in Figure 14.

3.4 Dynamic Load Balancing with Load Analysis

When the load increases, the primary load balancer will spawn new Axis2 application instances, and these instances will join the group. In order to detect changes in the load, a load analyzer will execute on the primary load balancer. When the load increases beyond specified thresholds, new Axis2 application instances will be started, and when the load drops below specified thresholds, unnecessary application instances will be terminated. The load balancers itself can be deployed in their own cluster in order to increase availability. Typically, such a cluster will contain 2 load balancer
instances; an active and passive instance each. We will later introduce a load analysis algorithm which will be executed on the active load balancer.

3.5 Failure Detection

Failure detection is an important requirement for any clustering implementation. Detecting failures in an asynchronous set up is a theoretical impossibility. This is mainly because we cannot set an upper bound on time in such an environment. In [19] Vogels argues that we do not assume that some person is deceased just because he does not answer the phone, we will do some background investigation like asking his landlord or neighbors, checking whether the electricity & water is being consumed and so on. The point is, timeouts at the transport layer are not always sufficient for failure detection. We can consult other data sources such as the Operating System where the process runs, use information about status of network routing nodes or can augment with application-specific solutions. In other words, some sort of external failure detection mechanism would generally provide better results. Anyway, this will not detect processes that seem to be UNSUSPECTED but are not working correctly. In other words, Byzantine failures cannot be detected. A robust Group Communication Framework can utilize these techniques to detect process failure with a higher degree of certainty which will suffice in all practical situations. However, such a setup will take longer to detect a failure due to the overhead involved. Therefore, we may have to live with a trade off; Unreliable Failure Detectors. Such a failure detector will set the status of a process to SUSPECTED, if a heartbeat is not received from the process within a certain time interval. Also, if the process fails soon after sending the heartbeat, the failure detector will still continue to classify the process as UNSUSPECTED.
3.6 Deployment Architecture

A load balancer group, running identical AMIs of the Apache Synapse load balancer fronts an application group, which runs identical AMIs of Apache Axis2. As illustrated in Figure 15, both these types of groups can run in the cloud. Two Amazon EC2 security group, one for the load balancers & the other for the Axis2 application instances, need to be created. Permissions must be granted for members in both groups to communicate with each other. Also, in the load balancer security group, the relevant HTTP & HTTPS ports should be open.

Figure 15: Deployment on EC2
Typically, the load balancer group will have one active and one passive Synapse load balancer. The active load balancer instance is assigned with an elastic IP address. This IP address also serves as the well-known address for the entire deployment. In the case of failure of the active load balancer, the passive load balancer will notice this failure, reassign the well-known elastic IP address to itself, enable its HTTP transports, and mark itself as the active load balancer. It will also fire up another load balancer AMI instance so that the system does not completely fail, if the second load balancer also fails. So generally, there will always be two load balancer AMIs executing. State replication specific to load balancing will take place within the load balancer group. Although it is possible to completely turn off state replication, it is unlikely that it will be done in the case of the load balancer group since some of the critical operations in Synapse such as access throttling & caching rely on this state, and if these states are not replicated and a failure occurs, the SLAs cannot be met.

The Apache Axis2 application nodes are unaware about the load balancers and the load balancer group. However these nodes are aware about the members in their own application group. Therefore, state replication can take place between the nodes in this group, if it has been enabled in the configuration provided using the axis2.xml file. When new application nodes join the group or existing application nodes leave the group, all other correct application nodes will be able to detect these view changes. The correct application nodes will not try to forward any group communication messages to the nodes deemed as failed.

The load balancer nodes are special nodes which belong to multiple groups. They belong to a load balancer group as well as the application groups they are load balancing.

```xml
<loadBalancer enable="true">
  <applicationDomain
    name="apache.axis2.app.domain"
    handler="org.apache.axis2.clustering.DefaultLoadBalanceEventHandler"/>
</loadBalancer>
```

The clustering configuration section in the axis2.xml of the load balancer defines one or more `applicationDomain` entries. These entries specify the name of the application groups being load balanced. The handler attribute specifies the implementation class through which Synapse can be notified of changes in the application domain membership. The above XML segment specifies that the load balancer is responsible
for load balancing across the `apache.axis2.app.domain` application group. The deployment should be configured properly so that the axis2.xml files of the nodes in the `apache.axis2.app.domain` group specify this value for the value of the domain parameter. It is this information that enables the load balancer nodes to become part of the application groups. Any membership view changes, in the load balancer group or application group, by way of nodes crashing, network link failures or new nodes joining can therefore be detected by the load balancers. When an application node fails, the load balancer will no longer try to forward requests to that failed node. When a new node joins, the load balancer will start forwarding requests to that newly joined node. It is also possible to configure an application group purely for scalability, in which case state replication is not necessary. Such a setup will have better performance than a system configured for state replication due to the overheads involved with state replication.

The external Web service clients only see a particular IP, in this case the elastic IP, and will connect to the node having that IP. In this setup, this will be the active load balancer, which has the elastic IP address assigned to it. The requests received will be load balanced across the application group, according to a specified load balancing algorithm. The load balancing algorithm can be specified in the Synapse configuration. There is no guarantee that all the requests from a particular client will be forwarded to the same application node in the general case. However, if sticky sessions are used, all the requests from a particular client will always be directed to the same application node throughout the session. However, such HTTP sessions are not migrated, hence in the event of a failure of the application node, all the work a client was doing will be lost, and the client may have to start all over again. It is possible that a request from a particular client is forwarded by the load balancer to an Axis2 application node and that application node fails while processing that request. In such a scenario, if failover has been enabled in the load balancer, the request will be forwarded to another node, if sticky sessions are not used. There can also be the case of the active load balancer failing while a client is invoking or trying to invoke it. In such a case, the passive load balancer will reassign the elastic IP address to itself. This will take at most a few seconds. The client can retry its request, and start over. Therefore, the failure of the primary (active) load balancer does not result in complete system failure. So the system can be viewed as an auto-healing system.
The autoscaling logic also resides on the load balancer. It is the most convenient & appropriate place to put the autoscaling functionality since the load balancer is the most suitable component in the system to gather data & statistics on the load on the cluster. While the system is in operation, the active load balancer keeps track of the number of requests in flight, and based on that value, makes decisions on autoscaling. If the load balancer perceives that the load is above a certain configured threshold for a certain period, it may decide that the load on the application group is high and spawn a new application node instance. Similarly, if the load balancer perceives that the load on a certain group of application nodes is lower that some specified threshold, it may terminate a few application node instances. However, if high availability is required, the application group should have at least two nodes running. If only scalability is required, having only a single application node will suffice. However, the load balancer node should periodically check and make sure that at least one instance of the application node AMI is running, even when only scalability is required.
3.7 Apache Synapse Autoscaling Load Balancer

3.7.1 Normal Message Flow

In the Synapse load balancer, the request from a client is intercepted by the *AutoscaleInMediator*. This mediator generates a universally unique ID (UUID) for this message, and records that UUID, along with the timestamp in the *MessageID List*. Next, the message reaches the *SendMediator*, which calls the *DynamicLoadBalanceEndpoint*. The dynamic load balancing logic will dispatch the request to an appropriate application node endpoint, depending on the load balancing algorithm. This is known as the IN mediation in the main sequence.

If the Axis2 node successfully processed the request, the OUT sequence will start to be executed. The first mediator in this sequence, *AutoscaleOutMediator*, will remove the entry in the *MessageID* List corresponding to the message with the UUID of the message in the IN sequence. Next the *SendMediator* within the OUT mediator in the main sequence will be called, at which point the response will be sent to the client which originally sent the request.

![Figure 16: Normal message flow](image-url)
At any point in time, the length of the MessageID List represents the number of requests currently being served, or the number of requests in flight. In general, if the load on the cluster is high, there will be more requests in flight and hence the length of the MessageID List will be relatively large, over a observed time period.

A LoadAnalyzerTask will periodically check the length of the MessageID List, and infer whether the load on the cluster is high, low or normal. Synapse supports registration of tasks which will be run at specified intervals. The LoadAnalyzerTask is such a Synapse task, which is configured using the synapse.xml file. Over several observation cycles, if the MessageID List length is beyond a certain configured threshold, this task will infer that the load on the cluster is high, and will spawn a new AMI instance. It will still continue to observe the list's length over several iterations, and looking at the list length will decide whether the load has reduced. If the load has still not reduced, it will spawn one more instance and continue. This may continue until a specified upper limit on the number of AMIs is reached. Similarly, if over several observation cycles, if the list's length is zero or almost zero, it will check the number of currently running instances & if that number is greater than 2, it will kill an instance. An instance which is currently processing a client request should not be killed. This can be easily done since the MessageID List holds the Synapse MessageContext which contains the IP address of the node to which the load balanced request was dispatched to. That IP address value can be compared against the IP address of an AMI which is designated to be killed.

Communicating with the Amazon EC2 is via secured Web service calls. All requests have to be signed using the private key belonging to the Amazon Web Service account holder. The generated WS-Security BinarySecurityToken has to be sent along with each request. An EC2 Client library was developed using Apache Axis2 client libraries for Web service invocation and Apache Rampart for WS-Security related functions. It is using this client library that the LoadAnalyzerTask communicates with the Amazon EC2 Web service, to launch new instances, terminate instances & get information about instances. The normal message flow has been depicted in Figure 16.
3.7.2 Error Message Flow

While dispatching a request to an application node or after a request is dispatched to an application node, an error can occur. If an application node crashes after a request is dispatched to it or while the request is being dispatched to it, and failover has been enabled on the DynamicLoadbalanceEndpoint the load balancer will try to dispatch the request to a different application node. However, if all such nodes crash, the FaultSequence will take effect. In this sequence, the AutoscaleOutMediator will remove the message placed in the MessageID List and forward the request to the MakeFaultMediator, which will create a fault message and hand it over to the SendMediator, which will send the fault message to the client. Similarly, if the client crashes after sending the request but before receiving the response, the FaultSequence will take effect. This ensures that stale messages do not remain in the MessageID List. The error message flow has been depicted in Figure 17.

Figure 17: Error flow
Figure 18: synapse.xml

Figure 18 shows a typical synapse.xml configuration file for the Synapse load balancer. The main sequence section defines the In & Out list mediators. These are simply mediators which aggregate other mediators. The **InMediator** defines the autoscaleIn mediator, send mediator & drop mediator in that order. This means, when a message is received by the load balancer, it will first hit the AutoscaleInMediator & then the SendMediator. Within the **SendMediator**, the **DynamicLoadBalanceEndpoint** will be called. The load balancing algorithm can be specified in the DynamicLoadBalanceEndpoint configuration section. Even though it defaults to RoundRobin load balancing, other implementations can easily be provided. For a practical scalability cluster, generally it is better to set failover to “true” so that the operation can be retried while the failure is transparent to the client. However, this may not be possible if sticky sessions are used. In such a configuration, the client has to be notified about the failure right away.
Note that the members in the application group have not been defined in this configuration. The membership of the application group is inferred through group communication semantics.

Finally, a *DropMediator* has been added to the end of the sequence to indicate that it is the end of the message sequence. The *OutMediator* defines the *AutoscaleOutMediator & SendMediator* in that order. Towards the end, the errorHandler sequence has been defined to deal with client failures or failure of an entire cluster of application nodes.

Separating out the autoscale mediators also has another advantage; if autoscaling is not desired but only dynamic load balancing is required, simply removing the autoscale mediator entries in the synapse configuration file will be sufficient.

### 3.8 Load Analyzer Task

This Synapse task periodically runs and monitors the load on the system, and makes scale up or scale down decisions.

#### 3.8.1 Load Analysis Task Configuration

This is the task configuration section in the synapse.xml file which configures the LoadAnalyzerTask. The task configuration XML fragment is shown in Figure 19.
Figure 19: LoadAnalyzerTask configuration in synapse.xml

Table 2 describes the different parameters that are used for configuring the LoadAnalyzerTask, as well as the purpose of each of those parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ec2PrivateKey</td>
<td>File path to the EC2 private key file</td>
</tr>
<tr>
<td>ec2Cert</td>
<td>File path to the EC2 certificate key file</td>
</tr>
<tr>
<td>instanceType</td>
<td>EC2 instance type of the instances to be launched</td>
</tr>
<tr>
<td>applicationGroup</td>
<td>EC2 group in which the Axis2 application instances should be launched</td>
</tr>
<tr>
<td>loadBalancerGroup</td>
<td>EC2 group in which the load balancer instances should be launched</td>
</tr>
<tr>
<td>instanceAdditionalInfo</td>
<td>Additional information about the instances</td>
</tr>
<tr>
<td>key</td>
<td>The AWS key to be used when launching the instances. This would be needed if</td>
</tr>
<tr>
<td></td>
<td>we have to connect to the running instances using SSH</td>
</tr>
<tr>
<td>applicationPayload</td>
<td>The file path of the application payload file. This payload has instance</td>
</tr>
<tr>
<td></td>
<td>specific configuration &amp; will be sent when Axis2 instances are launched.</td>
</tr>
<tr>
<td>loadBalancerPayload</td>
<td>The file path of the load balancer payload file. This</td>
</tr>
</tbody>
</table>


payload has instance specific configuration & will be sent when load balancer instances are launched.

<table>
<thead>
<tr>
<th>elasticIP</th>
<th>Elastic IP address which will be assigned to the primary load balancer</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageExpiryTime</td>
<td>The maximum time the message tokens need to be kept in the MessageID List</td>
</tr>
<tr>
<td>maxAppInstances</td>
<td>Maximum number of application instances the system should scale up to</td>
</tr>
<tr>
<td>minAppInstances</td>
<td>Minimum number of application instances that should be kept running, even when the load is low</td>
</tr>
<tr>
<td>minLoadBalancerInstances</td>
<td>Minimum number of load balancer instances that need to be maintained</td>
</tr>
<tr>
<td>queueLengthPerNode</td>
<td>Average Length of the MessageID List per Axis2 node, which indicates that the system may need to be scaled up or scaled down. This gives an indication to about the maximum number of concurrent requests that can be processed by a single Axis2 node within a reasonable time period. If the average queue length is greater than the (queueLengthPerNode x number of Axis2 nodes), the system is scaled up.</td>
</tr>
<tr>
<td>roundsToAverage</td>
<td>This is the minimum number of observations that are needed to make a scale up or scale down decision. A circular array which contains number of messages in flight are stored in this array, and periodically averaged, if the size of this array is greater than 'roundsToAverage'</td>
</tr>
<tr>
<td>instancesPerScaleUp</td>
<td>Number of EC2 instances that need to be started up, when the decision is made to scale up the system</td>
</tr>
<tr>
<td>Trigger interval</td>
<td>The time interval at which the LoadAnalyzerTask runs</td>
</tr>
</tbody>
</table>

### 3.9 Load Analysis Algorithm

The AutoscaleInMediator creates a UUID for each incoming message and places it in the MessageID List. Before the corresponding response is sent to the caller, this message is removed from that list by the AutoscaleOutMediator. The LoadAnalyzerTask periodically runs and observes the length of this list (listLength), and places that length into another array which will be later used to compare the moving average over a period of time. This average value gives an indication of the load on the system. The pseudo code of the load analysis algorithm is given below.
BEGIN
if (instancesBeingStartedUp > 0)
  return

messageIdListLengths.add(sizeof(msgIdList.size));

if (sizeof(messageIdListLengths) > roundsToAverage)
  messageIdListLengths.remove(0); // Ensure circular queue behavior
else
  return;

for each item in messageIdListLengths
  total = total + item

long avgMsgIdListLength = total / roundsToAverage;

if ( avgMsgIdListLength > (current Axis2 nodes * queueLengthPerNode) ) // Scale up situation
  if ( runningEC2Instances < maxInstances)
    start new EC2 instances
  else if ( avgMsgIdListLength < ( ( current Axis2 nodes - 1 ) * queueLengthPerNode)) // Scale down situation
    if ( runningEC2Instances > minInstances)
      for(RunningEC2Instances)
        if (running time in the hour >=58 minutes )
          terminate an EC2 instance

END

3.9.1 Making the Scale-Up Decision

If the average length of the MessageID List (avgMsgIdListLength) is greater than the queueLengthPerNode multiplied by the number of currently running nodes, it indicates that the system needs to be scaled up. However, if currently there are instances being started up, we do not try to scale up the system, but wait till those instances have started up. This is because it takes a few minutes for instances to start up & join the application group. The system is scaled up by starting a specified number (instancesPerScaleUp) of EC2 instances, if the maximum number of allowed instances (maxInstances) in the system has not been exceeded.

3.9.2 Making the Scale-Down Decision

If the average length of the MessageID List (avgMsgIdListLength) is less than (number of Axis2 nodes -1 ) multiplied by queueLengthPerNode the system may need to be scaled-down. If the currently running number of instances is greater than the minimum number of instances (minInstances) that should be running, the system may be scaled down. However, the system will check the time the instance was started at, and if its running time has run into another hour, it will not be terminated until it gets
close to the end of that hour. This is because, the instance hour has been already charged for by Amazon.

### 3.10 EC2 Client Library

Controlling the Amazon EC2 setup is handled through Web service invocations to the Amazon EC2 Web service. All the communication with the Amazon EC2 services is handled using the EC2 Client library. This library completely abstracts out the complexities associated with securely communicating with the EC2 Web service.

### 3.11 Handling Primary Load Balancer Failure

The load balancer group should contain at least 2 node balancer nodes in order to avoid a single point of failure. Typically an active & passive node balancer will be deployed. The active load balancer is also called the primary load balancer. All requests will be received by the active load balancer. The elastic IP address will be assigned to the active load balancer. If the active load balancer fails, the passive load balancer will try to contact the active load balancer & verify that it has actually failed. If it is positive that the active load balancer has failed, the passive load balancer will reassign the elastic IP address to itself, terminate the active load balancer EC2 instance if it is still running, enable its transport listeners, start another load balancer instance which will take on the role of being the passive load balancer, and finally mark itself as the active load balancer. Clients that were connected to the originally active load balancer will experience a failure. However, within a few seconds, those clients can resend their requests and these requests will be successfully processed. Hence this system is auto-healing as well, since it can automatically recover from such failures. We will later show how a Synapse task which does sanity check can periodically observe the cluster & maintain the minimum number of required instances.

### 3.12 Axis2 Application Cluster

The nodes in the Axis2 application cluster are the real worker nodes which host the Web service applications. Such a cluster may be configured for scalability only, or both scalability & high availability. In the latter case, state replication is a
requirement. Axis2 best practices specifies that the Web services implementation should be stateless. However, state can be maintained in the Axis2 context hierarchy. More specifically, java.io.Serializable objects stored in the respective ServiceContexts, ServiceGroupContexts & ConfigurationContext, can be automatically replicated across the cluster. The service authors need not write a single line of cluster-aware code; the Axis2 clustering implementation will completely take care of the replication, provided that Context management has been enabled in the axis2.xml file. This is done by enabling the contextManager entry in that file. Axis2 clustering also lets the author have more control over the state replication if required. The author could choose the specific objects in specific contexts that need to be replicated and notify Axis2 that those objects need to be replicated, as and when he chooses.

```xml
<cluster class="org.apache.axis2.clustering.tribes.TribesClusterManager" enable="true">
  <parameter name="AvoidInitiation">true</parameter>
  <parameter name="membershipScheme">wka</parameter>
  <parameter name="domain">apache.axis2.app.domain</parameter>
  <parameter name="synchronizeAll">true</parameter>
  <parameter name="maxRetries">10</parameter>
  <members>
    <member>
      <hostName>$ElasticIP</hostName>
      <port>4000</port>
    </member>
  </members>
  <contextManager class="org.apache.axis2.clustering.context.DefaultContextManager" enable="true">
    <replication>
      <defaults>
        <exclude name="local_*"/>
        <exclude name="LOCAL_*"/>
      </defaults>
    </replication>
  </contextManager>
</cluster>
```

**Figure 20: axis2.xml in application node**

Figure 20 shows the clustering configuration section of an Axis2 application node. We will look at the important parameters in this configuration.

The **membershipScheme** parameter should be set to **wka** in order to make the cluster work on EC2. This indicates that well-known address based membership discovery should be used. The **domain** parameter specifies the application group to which the Axis2 nodes belong to. This should be same as the **applicationDomain**
parameter specified in the synapse.xml file of the load balancer. In the member configuration section it would suffice to provide the elastic IP address of the well-known primary load balancer. Membership management in this setup revolves around the well-known active load balancer, which has the elastic IP address assigned to it.

### 3.13 Deployment on EC2

In this section, we will look at how the autoscaling system can be setup on EC2 & how the different configurations are maintained on S3. We have created a single AMI which contains the load balancer as well as Axis2. When an instance of an AMI is launched, it is provided with a payload which contains pointers to the deployment configuration. Specifically, there is a launch-params file in the payload which specifies the INSTANCE_TYPE of the AMI. If the INSTANCE_TYPE was 'synapse', it will behave as a load balancer instance & if the INSTANCE_TYPE was 'axis2' it will behave as an Axis2 instance. It is mandatory to provide this payload when launching an instance of this AMI. If the payload is unavailable, the instance will behave as an ordinary instance, and Synapse or Axis2 will not be automatically launched. An EC2 instance can retrieve its launch payload by sending an HTTP GET request to [http://169.254.169.254/1.0/user-data](http://169.254.169.254/1.0/user-data). It can also get information about itself by sending an HTTP GET request to [http://169.254.169.254/1.0/meta-data/](http://169.254.169.254/1.0/meta-data/).

A cron job which runs every minute is registered in the AMI. The job is specified in a shell script, autoscale-init.sh. This script is shown in Figure 21. The autoscale-init.sh script also ensures that either the synapse process or Axis2 process is always running. If not, it will automatically start the relevant process. This is done by calling 'start' on the synapse daemon or axis2 daemon.
#!/usr/bin/env bash

export LOG=/var/log/autoscale.log

if [ "first" -eq 1 ]; then
    echo Axes2 or Synapse Java process already running. Exiting. >> $LOG
    exit 0
else
    rm /root/mpi2/*
    rm /root/mpi2/*-char
    rm /root/mpi2/*-pid
fi

wget http://169.254.169.254/1.0/user-data -o /tmp/payload.zip

if [ ! -s /tmp/payload ]; then
    echo Payload already extracted >> $LOG
else
    unarchive /tmp/payload.zip -d /tmp/
fi

if [ ! -s /tmp/autoscale-init.sh ]; then
    echo Extracted payload >> $LOG
else
    echo rc.local: payload.zip is corrupted >> $LOG
fi

for i in /usr/bin/ruby /root/get-launch-params.rb
    do
        export $i
        done
echo Exported launch parameters >> $LOG

for i in /usr/bin/curl http://169.254.169.254/1.0/meta-data/
    do
        if [ $i == /v1/instance/attributes/instance-type ]; then
            echo Launching Synapse with repo $SYNAPSE_REPO >> $LOG
            mkdir -p /tmp/mpi2/payload
            touch /tmp/mpi2/payload/launch-params
            echo INSTANCE_TYPE=mpi2/AIX152_REPO=$SYNAPSE_REPO,AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID,AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY >> /tmp/mpi2/payload/launch-params
            cd /tmp/mpi2
            zip-r mpi2-payload.zip payload >> $LOG
            rm -rf /tmp/mpi2/payload >> $LOG
            echo Going to sync repo >> $LOG
            /usr/bin/ruby /root/mpi2/sync_repo.rb --make-dir -r $SYNAPSE_REPO /root/mpi2/sync_repo >> $LOG
            echo Synced the repo successfully >> $LOG
            cp /tmp/mpi2/payload/* /root/mpi2
            cd /root/mpi2
            echo Synchronizing mpi2/synapse-repository.sh start >> $LOG
        elif [ "$INSTANCE_TYPE" = "mpi2" ]; then
            echo Launching Axis2 with repo $AXIS2_REPO >> $LOG
            rm -rf /root/mpi2/repo
            echo Going to sync the repo >> $LOG
            /usr/bin/ruby /root/mpi2/sync_repo.rb --make-dir -r $AXIS2_REPO /root/mpi2/sync_repo >> $LOG
            echo Synced the repo successfully >> $LOG
            cd /root/mpi2
            echo Auto-scaling mpi2/axis2/synapse-demon.sh start >> $LOG
fi

Figure 21: autoscale-init.sh script
Axis2 & Synapse require their own configuration files during startup. Such configuration files cannot be placed in an AMI since then the AMI will need to be rebundled if the configuration needs to be changed. Hence such configuration files will be maintained in an S3 bucket. Axis2 & Synapse instances will have different S3 bucket folders from which they will load their respective configurations. When an AMI starts up, it will load the configuration files from a specified bucket using the `s3sync` tool. A Unix init script will call `s3sync` to load the relevant configuration files during the booting process of the AMI. How `s3sync` is called is shown in Figure 21.

### 3.13.1 Starting up an Axis2 Instance

An Axis2 instance requires an `axis2.xml` file in order to obtain its configuration. There may be other specific configuration files that may be needed. Such files are obtained from a specified S3 bucket during bootup. This is shown by the 'Load configuration' arrow in Figure 22. In addition, the Axis2 services that need to be deployed should also be loaded. The services & modules repository are maintained in the same S3 bucket. Loading of the services & modules during bootup using `s3cmd` is shown by the 'Load repo' arrow in Figure 22.

![Figure 22: Axis2 instance bootup](image)

The following script segment shows how the Axis2 S3 bucket & the local Axis2 instance's configuration & repository directory, are synchronized.
3.13.2 Starting up a Synapse Load Balancer Instance

In the case of the Synapse load balancer, the we need to load the synapse.xml & axis2.xml files from the S3 bucket. The cluster configuration section in the axis2.xml file is needed to properly setup the cluster. Synapse makes use of the underlying Axis2 clustering implementation. This is used for state replication within the load balancer group as well as discovering & managing Axis2 application instances. Loading the configuration needed during Synapse startup is shown by the 'Load configuration' arrow in Figure 23.

![Figure 23: Synapse load balancer instance startup](image)

The following script segment shows how the Synapse load balancer's S3 bucket & the local Synapse instance's configuration directory, are synchronized.
3.13.3 Auto-starting & Auto-healing a Cluster

Starting up each & every load balancer node & the initial set of Axis2 application nodes can be quite cumbersome. Hence, this task has to be automated. The system has been designed in such a way that only the primary Synapse instance needs to be manually started. This instance is marked as 'Initial Instance' in Figure 24. When this initial instance starts up, it will load its configuration file from a specified S3 bucket. This configuration file specifies information about the number of load balancer EC2 instances that should be available in the system, the Elastic IP address of the primary load balancer instance, as well as the minimum number of Axis2 application EC2 instances that need to be available in the system. The autoscale-init.sh script will launch a Synapse instance. When Synapse starts up, a task will look at the minimum configuration of the system & will start up the necessary number of nodes. The initial instance started up should be a load balancer instance & this instance will take on the responsibility of assigning the Elastic IP to its EC2 instance & booting up the other load balancer instances & Axis2 application node instances. This process is known as bootstrapping a cluster (Figure 24). This is achieved with a Synapse Task.

![Figure 24: Bootstrapping the system](image)

This task periodically runs & checks the health of the cluster. It checks whether the minimum number of load balancer instances & Axis2 application nodes are running in
the system. If not, it will startup up load balancer & Axis2 application node instances so that the minimum requirement is met at all times.

3.14 Failure Scenarios

In this section, we will take a look at some of the different types of failures that can occur in this system, and possible ways of recovering from these failures.

3.14.1 Load Balancer Failure

If the primary load balancer EC2 instance crashes, the secondary load balancer will detect the failure & remap the elastic IP address to itself, and spawn another load balancer instance which will then become the secondary load balancer. Clients which were connected to the old primary load balancer will notice this failure. However, when they retry after a few seconds, they will be able to connect to the instance which remapped itself to the secondary load balancer. A split brain scenario will not occur since the instance which has the elastic IP addressed mapped to it, will automatically be considered as the primary. In the case of the secondary load balancer EC2 instance crashing, the primary load balancer will start another new instance of a load balancer.

The EC2 instance may be running but the load balancer process might have crashed, frozen, or gone into a failed state. There are several ways in which we can recover from such a failure. There is a cron job running within the EC2 instance which periodically tries to connect to the HTTP & HTTPS ports of the load balancer. If these ports are inaccessible, the cron job will locate the failed process, kill it & start a new process. The JVM can become frozen for some reason. When that happens, the Java Service Wrapper process which started & monitors the load balancer process, will restart the load balancer process. We can also use a process monitoring tool such as Monit to monitor the log files, memory, disk usage & so on, and take remedial actions when anomalies are detected.
3.14.2 Axis2 Application Process Failures

Similar to the load balancer EC2 instance failure case, EC2 instances running Axis2 worker nodes can also crash. In such a case, during the sanity check phase of the load balancer Synapse task, the required number of instances will be started, and also additional Axis2 instances will be started depending on the load. If state replication has been enabled, the state would also have been replicated to other nodes on the cluster, so the system can recover. In this case too, a cron job monitors the Axis2 process and restarts failed Axis2 processes. The Java Service Wrapper process also restarts the Axis2 process if it has frozen or failed.
Chapter 4
Observations, Results & Analysis
In this chapter, we will show how to run load tests for 2 scenarios, derive the results from these test runs, and show how autoscaling contributes towards increase in throughput when the load increases, whereas the response times are more or less constant. Moreover, we will show how the performance degrades when the load increases & the system does not scale out. Performance degradation can be seen in terms of a drastic drop in throughput & a drastic increase in the average response time. The increase in the number of failed or timed out requests is another indicator of performance degradation.

### 4.1 Performance Testing Methodology

We will be using the popular Apache Bench (ab) tool, which is widely used for performance testing Web based systems. In other words, Apache Bench will act as the client which loads the system. Although initially developed for benchmarking the Apache HTTP server, it is now used for benchmarking other HTTP servers as well. Our autoscaling system is based on the SOAP/HTTP protocol, hence Apache Bench can be effectively used for load testing our system. The payload, total number of requests to be sent, as well as the level of concurrency can be provided as input parameters to Apache Bench. We hosted a test Web Service, PerfService, on the Axis2 worker nodes. The echo operation in this test service is invoked during the test runs. The echo operation does some work that puts a considerable load on the CPU. Hence, when an Axis2 node hosting this service is hit with a large number of concurrent requests, it will place significant strain on the machine.

Apache Bench produces a test run report similar to the following at the end of a run.

```
Server Software: Synapse-HttpComponents-NIO
Server Hostname: 75.101.131.185
Server Port: 8280

Document Path: /services/PerfService
Document Length: 780 bytes

Concurrency Level: 10
Time taken for tests: 3.666 seconds
Complete requests: 10
Failed requests: 0
Write errors: 0
Keep-Alive requests: 0
Total transferred: 9420 bytes
Total POSTed: 9890
HTML transferred: 7800 bytes
```
Requests per second: 2.73 [#/sec] (mean)
Time per request: 3666.006 [ms] (mean)
Time per request: 366.601 [ms] (mean, across all concurrent requests)
Transfer rate: 2.51 [Kbytes/sec] received
2.63 kb/s sent
5.14 kb/s total

Connection Times (ms)
min mean[+/-sd] median max
Connect: 1 1 0.1 1 1
Processing: 530 1477 1239.2 1441 3665
Waiting: 439 1453 1259.7 1441 3664
Total: 531 1478 1239.2 1443 3666

Percentage of the requests served within a certain time (ms)
50% 1443
66% 1551
75% 1654
80% 3666
90% 3666
95% 3666
98% 3666
99% 3666
100% 3666 (longest request)

We can clearly get an idea about the response times & throughput by analyzing this report.

The two test scenarios are as follows:

4.1.1 Scenario 1 – No autoscaling

A single Axis2 worker node sits behind a non-autoscaling Synapse load balancer. A single Apache Bench client loads the system & gathers statistics at the end of each run. This scenario is depicted in Figure 25.
4.1.2 Scenario 2 – Autoscaling Enabled

This is the autoscaling scenario where one or more Axis2 nodes will sit behind a Synapse load balancer which is configured for autoscaling. A single Apache Bench client loads the system & gathers statistics at the end of each run. This scenario is depicted in Figure 26.
4.2 Test Results

![Graph showing response time variation](image)

Figure 27: Response Time Variation
4.3 Analysis of Test Results

When there is no autoscaling, the throughput of the system is constant even though the number of concurrent request is increasing. Also, the response time drastically increases. When autoscaling is enabled, the throughput increases with concurrency, and the response time is almost constant. However, we cannot achieve ideal throughput values & response times because an AMI take a few minutes to start up, hence during those few minutes, there is a performance degradation. This is because during these few minutes it takes an AMI to bootup, the extra requests have to be served by the current group of Axis2 nodes. The gap between the ideal throughput & the actual throughput is due to this reason. The respective graphs are shown in Figure 27 & Figure 28.

This shows that the system nicely scales when the concurrency increases. The clients do not see a significant drop in the quality of service & response times.
Chapter 5

Conclusion
Autoscaling Web services on cloud computing environments such as Amazon EC2 has a technical appeal as well as an economic appeal. We try to address the issue of allocating unnecessary safety-net capacity and also make the system increase or decrease its cumulative computing power depending on the actual load, instead of having a static configuration which has been allocated based on the anticipated peak load. Such a traditional setup on cloud computing environments is a wastage of resources, and we can provide a better solution to the peak load problem using some of the inherent features of the cloud. The available computing power should vary proportionate to the actual load experienced. This can be achieved by starting up new EC2 instances when the load increases & terminating running EC2 instances when the load decreases.

Multicast-based membership discovery does not work in such an infrastructure. Traditional multicast based member discovery mechanisms have to be replaced with a Well-Known Address (WKA) based scheme on cloud computing infrastructures such as EC2 since multicast has been blocked on such networks and since there is no guarantee that all members will belong to the same multicast domain. This is because network and IP addresses are randomly assigned from an available set of addresses. This thesis introduced a WKA based membership discovery protocol & showed how the protocol can robustly handle different failure scenarios.

This thesis also introduced the concept of membership-aware dynamic load balancing, where the load balancer can itself made part of the group it is distributing the load across, so that membership changes in these application nodes can be detected by the load balancer. Traditionally, the list of nodes across which the load is distributed had to be provided to the load balancer. Such a load balancer is known as a static load balancer. Hence, new members could not join the application group at runtime. However, such a setup will not work economically on EC2 since the only way of achieving static load balancing is to reserve a set of elastic IP addresses, assign them to the application instances & and then provide those IP addresses to the load balancer. The maximum number of application members will also be bound by the number of elastic IP addresses reserved, if such an approach is followed. This defeats the idea of autoscaling since new members cannot join the cluster & IP addresses of members cannot change to anything other than those assigned from the set of elastic
IP addresses. Our approach enables membership discovery by the load balancer, hence making it a membership-aware dynamic load balancer.

The fault resilience features provided by Amazon EC2 such as elastic IP addresses & availability zones, can be utilized so that the system is truly fault tolerant, irrespective of failure causes, even in some catastrophic situations such as an entire availability zone failing. Even failures of well-known members who have elastic IP addresses assigned to them can be recovered from by dynamically reassigning the elastic IP address. The members can be distributed across availability zones so that complete failure of a region will not result in failure of the system.

We also introduced the concept of an autohealing cluster is a mechanism where the system continues to maintain its minimum configuration, even when some nodes fail. A task which periodically executes on a node checks the current configuration of the cluster against a specified minimum requirement, and ensures that this minimum requirement is always satisfied.

Amazon recently introduced its own autoscaling load balancer implementation called Elastic Load Balancer (ELB) [25]. Some of the concepts between the Amazon implementation & this implementation are quite similar. The Amazon autoscaling takes into consideration additional parameters such as the CPU usage when deciding to scale-up or down. The main advantage in adopting this approach is that it is not bound to the Amazon APIs or implementation, hence we can easily adopt this system to run on any other cloud as well. Different autoscaling algorithms can also be plugged-in this implementation. The main disadvantage in this approach as opposed to using the Amazon load balancer is that, in this implementation, additional redundant load balancer instances need to be run, whereas in Amazon’s approach, the Amazon infrastructure itself takes care of this.

Even though this thesis concentrates on autoscaling Web services, the major concepts introduced here such as Well-known address based membership discovery in environments which do not allow IP multicasting, Membership-aware dynamic load balancing & Autohealing cluster can be easily adopted for other areas.
We have also shown how the system's response time drastically increases with load in the non-autoscaling scenario, and how this can be translated to constant response time with autoscaling enabled. This also means that the overall system throughput increases with load in the autoscaled scenario, whereas it is constant in the non-autoscaled case.

5.1 Future Work

As future work, we could implement the same autoscaling setup with different group communication frameworks & carry out a performance comparison study of these different systems in an autoscaling environment. We could also provide tooling that can be used for managing an autoscaling Web services application server. Such tooling will need to handle aspects such as atomically changing the policies of running services and atomically deploying & undeploying services.

Another enhancement to this implementation will be to develop an Amazon EC2 aware load balance algorithm which will be able to take into consideration the actual load on each EC2 instance when deciding on load balancing across the members. Parameters such as the current processor usage & memory usage can be taken into account. This approach will be much more effective than simply taking into consideration the number of messages in flight.
REFERENCES


