Auto-scaling Web Services on Amazon EC2

Afkham Azeez
Department of Computer Science & Engineering
University of Moratuwa, Sri Lanka
WSO2 Inc.
azeez@{apache.org, wso2.com}

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. In this paper, we will look at an approach towards automatically scaling Web service applications while maintaining the availability & scalability guarantees at an optimum cost. The Web service application developer should only need to write the application once, and simply deploy it on the cloud. The scalability & availability guarantees will be provided automatically by the underlying infrastructure. We will describe in detail an approach towards building auto-scaling Web services on Amazon EC2.

Keywords

Auto-scaling, Web services, Apache Axis2, Amazon EC2, Fault tolerance, High availability, Scalability, Well-known Addressing, Dynamic load balancing

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] rather 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 [1] (Amazon EC2) 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 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] is a middleware platform which enables hosting of Web service applications and supports some of the major Web services standards. Axis2 can be deployed in a clustered configuration. In fact, clustering is a major value addition provided by Axis2 for an enterprise deployment. Apache Axis2 supports clustering for high availability & scalability. 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 charge by Amazon is mainly 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. Hence we can achieve an optimal balance between the cost and performance using this auto-scaling approach. This will be a very appealing idea from a business perspective. We will be looking in detail how Apache Axis2 Web services can be auto-scaled on Amazon EC2.

2. 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 [5] which can cater to varying computing & memory capacity requirements.

2.1 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 [6] - 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 [7]- 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.

3. Apache Axis2

Axis2 [4] is a middleware platform which enables hosting of Web service applications and supports some of the major Web services standards and can be deployed in a clustered configuration. 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 [8] 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). 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) 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"*.

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 high scalability guarantees, whereas in the case of stateful services, multiple replicas are necessary to provide high availability guarantees.

4. Prior Work

GigaSpaces [10] 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. However, this is a general purpose solution and the focus of this paper is deploying highly scalable Web service applications on EC2.

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 [11]. 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 [11]. 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 [12] take into account several aspects such retrying, in-order , at-most-once delivery, and persistence. This work tries to address some of these shortcomings in this area. The Apache Tribes group communication framework (GCF) based clustering implementation for Axis2 also is a very recent development.

5. 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 auto-scales, 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.

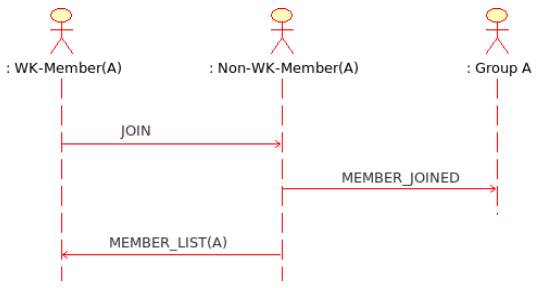

Fig 1: Member joins group

This paper introduces 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-know 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 1.

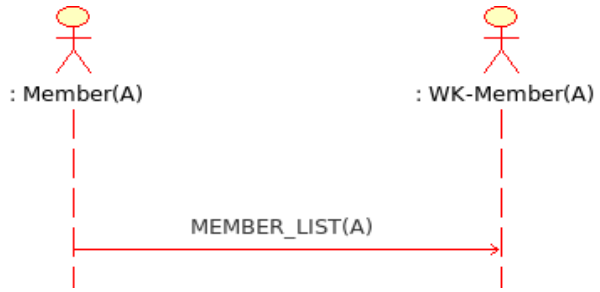

Fig 2: Well-known member rejoins after crashing

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 2.

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.

6. Fault resilient dynamic load balancing

For dynamic load balancing, group membership is used. 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 3. 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.

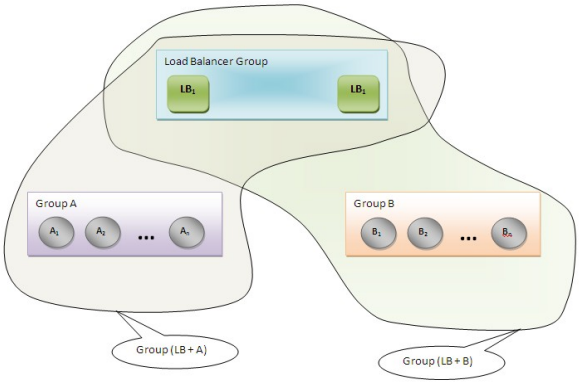

Fig 3: Load balancer & application groups

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.

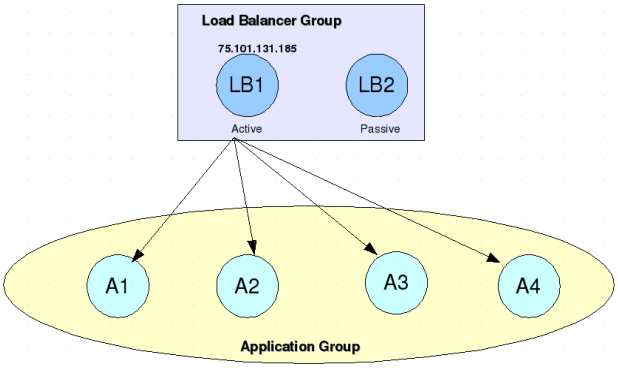

Fig 4: Active-passive load balancers with Elastic IP

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 4), 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.

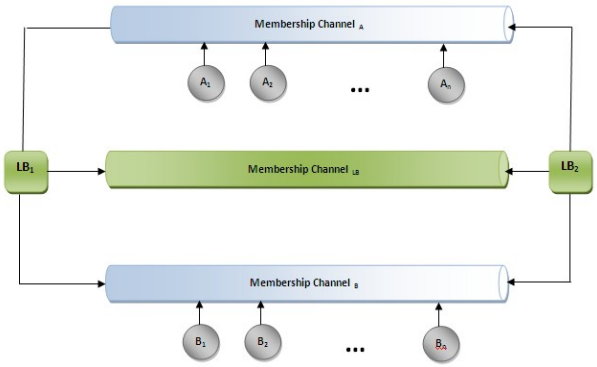

Fig 5: Membership channel architecture

All load balancer members are connected to all membership channels of all groups as illustrated in Figure 5. 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.

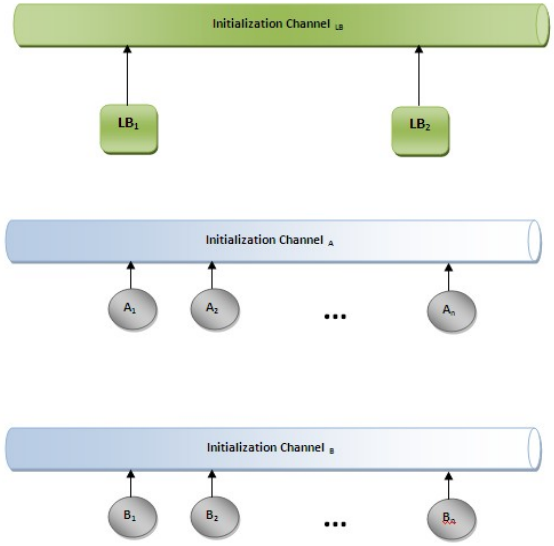

Fig 6: Initialization channel architecture

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 6.

As shown in Figure 7, 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.

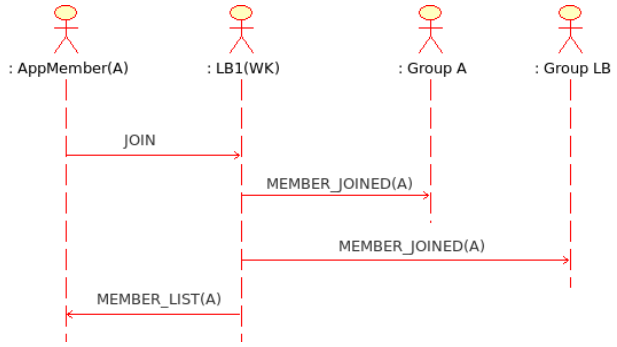

Fig 7: Application member joins. A load balancer is also a well-known member

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 8, when a load balancer (LB2) which is 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.

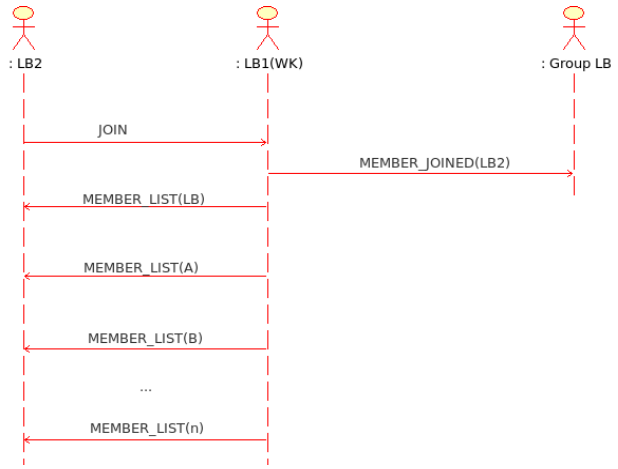

Fig 8: A non-WK load balancer joins

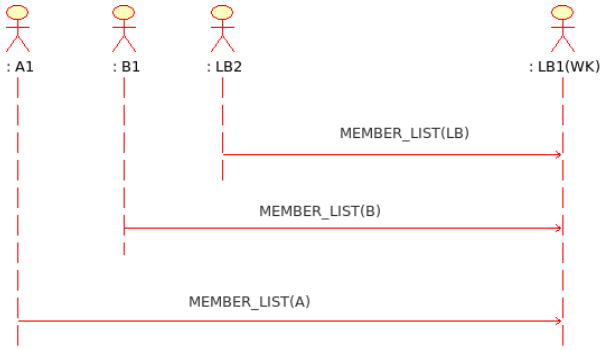

Fig 9: A well-known load balancer rejoins after crashing

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 9.

7. Dynamic load balancing with traffic 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 traffic 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 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 the *ec2-auto-scaling* load balancing algorithm which will be executed on the active load balancer.

8. 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 [13] 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.

8.1 Failure Detection Models

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

Synchronous failure detection models [14] 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.

The value of the asynchronous model [14] is that it makes no assumptions on time, and 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. So we can figure out 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.

In the Apache Axis2 clustering implementation, it is possible to incorporate different failure detection implementations. Hence depending on the accuracy required, better failure detection implementations can be used. However, we will need to consider the cost-benefit trade-off for these different implementations.

9. Overcoming Distributed System Failures

The general approach for overcoming failures in a distributed system is to replace each critical component with a group of components that can each act on behalf of the original one . This is the approach we are following in the Amazon 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.

Redundancy is the widely adopted technique when it comes to fault tolerance. 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 default 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.

10. Implementation

The load balancer used is Apache Synapse. The dynamic load balancing capabilities of Apache Synapse are utilized. This load balancer will be deployed in its own cluster with one active instance & one passive instance. The active instance will also have a well-known elastic IP address. The application domain names across which the load has to be balanced can be provided to the Apache Synapse dynamic load balancing configuration. The load balancing algorithm can also be provided. When deployed on Amazon EC2, we will use the *ec2-auto-scaling* algorithm. This algorithm will make use of a traffic analyzer which will keep track of the average time taken to process each client request and the average requests per second in a given time period. If the average processing time or average requests per second is increasing, this indicates an increase in load. When the load hits certain thresholds, new Axis2 application instances will be started up. These new Axis2 application instances will join the group using the membership information from the well-known Apache Synapse load balancer instance. Again, these configurations can be provided using the Synapse load balancer configuration. Similarly, when the load reduces beyond these thresholds, unnecessary instances may be terminated after a certain grace period. This grace period is needed so that the system does not go into a cycle of starting new instances & terminating them after a short while. If the active Apache Synapse load balancer instance fails, the passive load balancer will detect this failure, reassign the elastic IP to itself, and then take over as the active load balancer. The passive instance will double check to see if the previous active instance has actually failed, and if so, will start a new passive load balancer instance.

The application members will be Axis2 instances which will host the Web service applications. We can configure this application cluster to enable state replication, in which case, the system will be highly available. We could also configure this cluster only for scalability without state replication, which will give better performance.

An administrator with cloud configuration privileges will define the environment. The load balancer cluster configuration, traffic analysis parameters, defining and assigning application domain to these load balancer will

be done by this administrator. The person who deploys the services will not see this complexity & will be provided with a Web-based front end and scripts for deploying services on the cloud. The complexities involved with auto-scaling will be transparent to the service authors & service deployers.

11. Future Work

There are several popular group communication frameworks (GCF). We need to analyze whether Apache Tribes is the best GCF to be used on Amazon EC2, in terms of performance & cost. JGroups [15] is such a GCF, which is widely used. Spread [16] is another GCF which is used for group communication across Wide Area Networks. Ensemble [17] & Ricochet [18] are GCF pioneered by Birman. We could carry out a comparison study of Axis2 when these different GCF based clustering implementations are used, to see which gives better performance and better economy on Amazon EC2.

12. Conclusions

Auto-scaling Axis2 [19] Web service applications on Amazon EC2 is a very appealing idea from a business point of view. Such an approach makes efficient usage of resources on a cloud computing environment, and achieves an optimal balance between performance, cost and availability & scalability guarantees.

References

1. Amazon EC2, http://www.amazon.com/gp/browse.html?node=201590011, 21st April 2008
2. Google App Engine, http://code.google.com/appengine/, 21st April 2008
3. Who Are The Biggest Users of Amazon Web Services? It's Not Startups. http://www.techcrunch.com/2008/04/21/who-are-the-biggest-users-of-amazon-web-services-its-not-startups/, 21st April 2008
4. Apache Axis2, http://ws.apache.org/axis2, 21st April 2008
5. Amazon EC2 Instance Types, http://www.amazon.com/Instances-EC2-AWS/b/ref=sc_fe_c_0_201590011_2?ie=UTF8&node=370375011&no=201590011&me=A36L942TSJ2AJA, 21st April 2008
6. Amazon EC2 Availability Zones, http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1347, 21st April 2008
7. Amazon EC2 Elastic Ips http://www.amazon.com/gp/browse.html?node=201590011, 21st April 2008
8. Web Services and Axis2 Architecture, http://www.ibm.com/developerworks/webservices/library/ws-apacheaxis2, 21st April 2008
9. Apache Tribes, http://tomcat.apache.org/tomcat-6.0-doc/tribes/introduction.html, 21st April 2008
10. GigaSpaces - High-Performance, Scalable Applications on the Amazon Elastic Compute Cloud, http://www.gigaspaces.com/download/GigaSpaces_and_Amazon_EC2.pdf, 21st April 2008
11. Reliable Distributed Systems, Technologies, Web Services & Applications, by Kenneth P. Birman, pp. 242, 352-354, 369, 572
12. WS-Reliable Messaging, http://www.ibm.com/developerworks/library/specification/ws-rm/, 21st April 2008
13. World Wide Failures, Werner Vogels, http://www.cs.cornell.edu/projects/spinglass/public_pdfs/World%20Wide.pdf, 21st April 2008
14. Distributed Systems, Concepts and Design, 4th Edition, by George Coulouris, Jean Dollimore & Tim Kindberg
15. JGroups, http://www.javagroups.com/javagroupsnew/docs/, 21st April 2008
16. Spread Toolkit, http://spread.org/, 21st April 2008
17. Ensemble Groupware System, http://www.cs.technion.ac.il/dsl/projects/Ensemble/overview.html, 21st April 2008
18. Ricochet: Lateral Error Correction for Time-Critical Multicast, Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, Stefan Pleisch ,http://www.cs.cornell.edu/~mahesh/publications/docs/ricochet-camera.pdf, 21st April 2008
19. S. Perera et al. Axis2, Middleware for Next Generation Web Services, In Proc. ICWS 2006, pp. 833-840, Sept. 2006.