

A FRAMEWORK FOR DEVELOPING HIGHLY AVAILABLE AND SCALABLE WEB SERVICES

Mark Alan MacKinnon Gardiner

Department of Electrical and Computer Engineering
University of Auckland, Auckland, New Zealand

Abstract

Web services are subject to wild and unpredictable loads. A Java framework, named Hydra, was created for use with a cloud computing service to autonomously scale a service, so that it provides a consistent quality of service. The framework is built on the autonomic principles of self-configuring, self-healing, self-optimising and self-protecting. It works independently of the application it is scaling and it is configurable so that it can adapt to the needs of the specific web service it is scaling. A modular design means that the behaviour of the system can change at run time. The framework was evaluated using an implementation designed for PeerWise, a practice peer-assessment website for students, and using Amazon's Elastic Compute Cloud (EC2) as a cloud computing service.

1. Introduction

A web service is a software system designed to support interoperable machine-to-machine interaction over a network [1]. In the context of the Hydra framework, the web services that the framework intends to scale include all of the kinds of services that are available over the Internet. This includes services such as websites, download servers, Internet relay chat networks and data warehousing services. All of these web services are often subject to rapidly growing and unpredictable loads[2, 3], which creates a problem when the load experienced by a web service grows beyond what the software was designed to cope with, or what the available physical machinery can handle. Each web service may be prone to a different type of load, for example, a web service that requires a significant amount of computation for each request, may require additional CPUs as the number of requests increases. Similarly, a service that acts as a cache may not have an effective amount of memory for the number of requesting machines. These factors of load can contribute to a degraded quality of service (QoS) as a system becomes overloaded. Adding servers is one method of reducing this degraded quality of service[2, 14]. This report details the implementation and design decisions that came about when creating the Hydra framework, so that it

increases the quality of service and availability of computational units.

1.1. Objectives

The main objective of this project was to create a framework that allows web services to scale, so that it provides a consistent quality of service, while minimising the cost. Once this framework has been created, an example implementation was to be created, to demonstrate the scaling of a web service. In particular the system is hoped to run autonomously, maintaining and managing itself. This can be seen in four functional areas of self-configuring, self-healing, self-optimising and self-protecting. Self-configuring should be represented by the ability to adapt to different conditions such as using different metrics for scaling as the web service changes behaviour. Self-healing should recover the system as parts of the service crash or become unavailable. Self-optimising should be demonstrated by an optimal number of servers being activated for any given load on a system and the system should be self-protecting by authenticating communication amongst servers and excluding servers that fail to do so. The framework should also be simple to use and well documented, so that developers can quickly and easily configure the framework for their web service. Finally, the framework should work independently of the service it is scaling, allowing the service to be developed without any knowledge of how the framework handles scaling.

1.2. Approach

The approach to creating the framework came from the principles defined in IBM's Autonomic Computing Manifesto [4]. This approach looks at the system like a self-regulating and biological system, where all parts of the system are self regulated without conscious intervention. Hydra uses a modular approach, where the modules are decoupled from each other. Each functional module runs as its own thread and by itself does not rely on the function of any other functional modules. This was simplified by using Java and object orientated programming principles.

2. Motivation

Scaling web services has been a challenge since the early stages of the Internet. The current approach to scaling many small scale web services is to manually add a new machine to a local cluster. This is certainly a reasonable approach for a web service used within a small organisation that knows how the service will be used and to what extent it will have to scale. However, to provide a consistent quality of service to a growing number of customers, this may not be sufficient.

Taking this approach of manually adding servers to scale a web service with an unpredictable load, the service would need to be constantly monitored to ensure it is providing a consistent quality of service and additional machinery would have to be bought and configured as soon as the physical capacity appeared not to be great enough. The problems with monitoring and adding more machinery is the cost and delay involved. If a system becomes overloaded too quickly, there may not be the time to get in and configure the additional machinery required to prevent a degradation of service. For many services this can mean a loss of customers. If the increase in load is an infrequent occurrence, then it may not be cost effective to buy the required additional machines.

One example of a web service facing these problems is a website for the Australian Open[5]. The Australian Open is a tennis tournament which is held in January each year, and the website receives a significant increase of traffic over the tournament period (Figure 1).

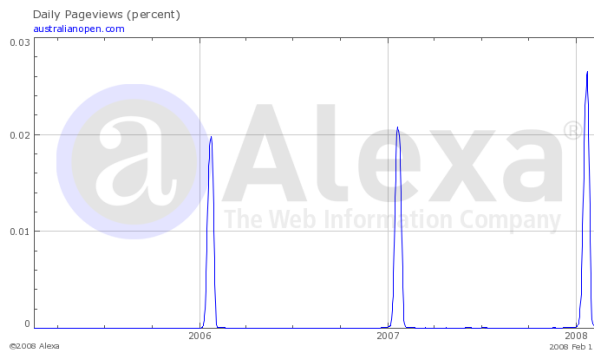


Figure 1 Predicted percentage of internet page views for australianoopen.com by Alexia.

The increase in traffic can amount to over 100 times its typical volume (22 million visits in a couple of weeks)[5]. For a change in capacity of this scale, it would not be financially practical to have 100 servers dormant throughout the rest of the year for every server in use.

Another example is Animoto. Animoto is an application for creating custom music videos using pictures taken from the popular social networking website Facebook. Animoto experienced a steady linear rate of growth with around 25,000 users signing up over the course of a month. When Animoto found that many users were

signing up, but never creating a video, they decided to automatically generate a video using each users picture gallery. The result was a staggering increase in load on the web service (Figure 2).

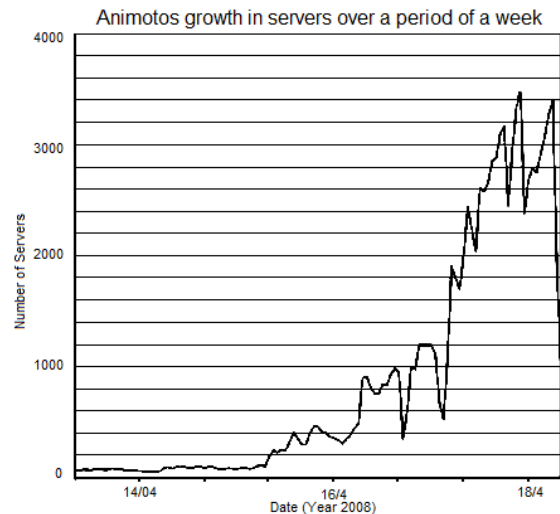


Figure 2 Number of servers used by Animoto's music video generating service.

Over a period of four days, the number of people signed up to Animoto's service grew from 25,000 people to 250,000 people. Consequently, the number of servers Animoto was using grew from less than a hundred servers to a peak of over 3,400 in the space of four days.

3. Background

A simple method to cope with more load is to add more servers providing the service. Likewise if a server crashes, the service can be recovered by replacing the failed server. The framework that was developed was named Hydra, after the Lernaean Hydra, a mythical beast that responded to attacks in much the same way as our framework. According to Greek mythology, whenever one of the Hydras many heads were cut off, new heads would grow back in its place to maintain its defence. Responding to load in the same way, the Hydra framework starts additional servers as current servers are overcome with load. In addition to this, to minimise cost, Hydra stops any additional servers as the load is reduced.

The main problems with the approach of manually scaling a web service to keep up with load, is the cost and time involved. When expanding a web service not only do more machines have to be bought, but additional floor space, maintenance staff, and bandwidth has to be bought. Ideally a business should only have to run as many servers as is needed at any one time. If too many servers are bought, then the excess servers would spend most of their time providing no financial return or simply not in use and depreciating in value. Conversely, if

too few servers are bought, the service as a whole could suffer, which can translate into loss of customers, loss of information and ultimately in a business context, loss of revenue. This resourcing problem has been partially solved thanks to recent work in cloud computing services. It may also take too long to get in new servers, find space for them, and configure them before service is lost.

Cloud computing is a relatively new in the area of Internet hosting. The idea with a compute cloud is to provide a service on vast numbers of relatively cheap low performance machines, as opposed to a service on a few high powered machines. It is estimated that Google for example has around half a million servers in a dozen locations to provide their services[6]. By having this service spread across physical locations as well as machines, the service is more robust and can be described as “Self-healing” as it absorbs network or machine failures [6]. Some large companies such as Amazon, Google and IBM who have their own large-scale data centres have started sell excess computing ability in the form of Amazon Web Services, Google App Engine and IBMs Blue Cloud respectively. For the exemplary implementation of the Hydra framework, Amazons infrastructure was used.

3.1. Amazon Web Services

Amazon.com provides a set of web services, Amazon Web Services (AWS), for use by developers and their companies [7, 8]. We used three of these services, namely the Elastic Compute Cloud (EC2), the Simple Storage Service (S3), and the Amazon Elastic IPs. The Amazon Elastic Compute Cloud provides hosting for web services on a scalable infrastructure. Using EC2, it is fairly easy to start additional virtual machines, which can hold a replica of the service and can be used to help balance the load upon that service. For the implementation of Hydra using PeerWise, Amazon EC2 was used as a basis for hosting and scaling the web service.

With EC2, each instance is a complete virtual machine and is billed at an hourly rate. By using AWS it was possible to overcome the problem that it is not practical to have a large number of physical machines on hand for sporadic increases in service traffic. New machine instances can be started with just a few minutes delay, and a web service could effectively span an uninhibited number of machines. Amazon does not provide a framework for automatically scaling a web service based upon the load it is experiencing, and this is what was addressed by the Hydra framework.

The simple storage service provides reliable and secure storage of data. This service is charged by the quantity of storage and requests, and provides a platform for Hydra to share state information or even new modules.

Lastly, Amazons elastic IPs allows quick reassignment of a static IP to any running machine instance through an internal Amazon mapping. This means that

an external domain name can change which machine it is ultimately pointing to without having to make changes at the DNS level which could take days.

3.2. PeerWise

PeerWise is an online collaborative multiple-choice question repository[9] (Figure 3).

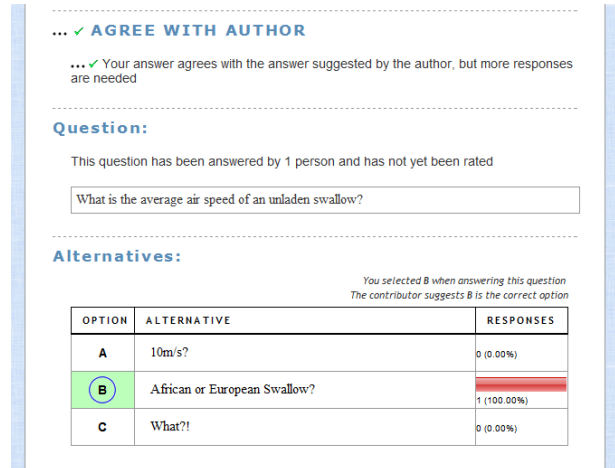


Figure 3 Screenshot of PeerWise showing a form for viewing collective responses to a question.

The PeerWise system is aimed at students who can write multiple choice questions for other students to answer, or practice by answering questions proposed by other students. The system allows the students to see what the proposed answer for a question was by its creator and see the distribution of answers for a particular question. The system also supports commenting on questions and keeping track of the answers a particular student has made.

3.3. Related Work

Throughout the development of the Hydra framework several similar frameworks have started to emerge. There were however a few existing frameworks such as RightScale and 3Tera at Hydras inception. These systems differ from the Hydra framework however in that they tie you into a particular architecture.

Another framework we investigated was RightScale. RightScale is a commercial portal system for using Amazon Web Services, and provides limited functionality for scaling web services to ensure a quality of service. One of the biggest drawbacks of this system is that there is that the developer is tied into a particular web framework and as an example there is no method for implementing proactive scaling. RightScale is built specifically for use with Amazon Web Services and is targeted at website services that use an HAProxy load balancer and three tier architecture. This differs from the Hydra system, in

that Hydra should work with any kind of load balancer or application setup. In addition, unlike RightScale, it should have the ability to scale by adaptively predicting when there will be an increase in traffic. Further to this, RightScale’s scaling is controlled by the RightScale instance, and if this goes down, the web service will no longer scale.

3tera is a cloud computing service that offers a visual application to design and automate the system. This is controlled by a 3tera server which represents a single point of failure[10]. While there is quite a bit of support for designing a website deployment, this system is not so straight forward for other kinds of web services. Furthermore the system requires a lot of configuration.

There has also been work around starting and stopping servers to conserve power[14]. This showed that excess servers can be successfully stopped to help save power and money, however this doesn’t address the problem of having to upscale a service as the load becomes too much for the available resources to handle.

4. Key Design Issues

When designing the system, there were a few key issues surrounding when to start additional machines, how this will be done and what the needs of different kinds of web service will be like. This includes the issue of whether it is better to have a centralised control which monitors the system or to distribute the control across machines. The next issues included how to measure the quality of service(QoS) being provided by the system, if the system should cater for proactive as well as reactive scaling, and lastly how to balance the load across the system.

4.1. Distributed vs. Centralised Control

The first framework that was envisioned was one with a centralised control. Centralising the control would have many benefits and would be relatively simple to create. One machine instance would act as the authority and monitor all instances providing the service.

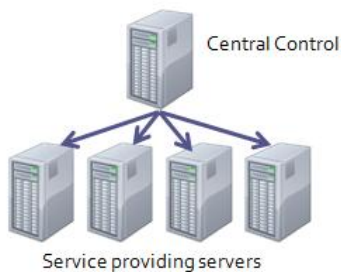


Figure 4 A single server with centralised control

The controlling instance would then start and stop instances as it determined was necessary, similar to RightScales approach. The arrows in Figure 4 represent

the path of communication where each service providing server listens to a single server for commands to start or stop. This machine could also control load balancing as it could use the information it knows about which instances are under the greatest load, and defer requests to instances with the least amount of load.

The main problem with this approach is that it creates a single point of failure. It could potentially limit the degree to which the service could scale if all requests have to go through it. This introduces a bottleneck in the system which should be avoided. Should the controlling instance become unavailable for any other reason, then the system will not scale or, in the case where it is in charge of distributing load, the service could become unavailable all together.

Distributing the control was the second option that was considered. A distributed system would work by attempting to replicate the controlling service across multiple instances so that if a subsection of the controlling instances were to fail, any remaining instances could recover. In Figure 5, the arrows between the servers show that each of the servers would have to communicate amongst all the remaining servers.

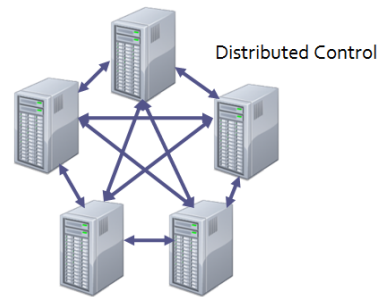


Figure 5 Servers controlled by a distributed control

To control the system, these instances would have to collaborate by sharing load information, and collectively deciding to stop or start a new instance when one is required. The added reliability of distributed control creates many situations which require coordination. For instance, it would be dangerous if each controlling instance independently recognised a decreasing quality of service and simultaneously decide to start an additional instance. Instead of just one additional instance starting, each of the controlling instances would start one independently. Likewise for a lull in the traffic, each control might decide it is no longer needed and the system could shut down completely. A further problem could arise when trying to shut down the service. Every time one of the services is shut down, the remaining instances may assume that it crashed and start up a new instance in its place. Communication amongst instances in this situation can be important to ensure that all servers agree upon decisions being made.

4.2. Measuring the Quality of Service

Measuring the quality of service being provided was identified as one of the key factors that will determine how accurately the system responds to load. Measuring the CPU usage, or the number of processes waiting in a queue to access the processor, is a common metric for assessing load on a server, which in turn affects the quality of service provided. This may not be the case for many web services which are heavily reliant on other system resources. As such it was determined that for a system to effectively measure the quality of service for different web services, the system should have the ability to use different sets of heuristics which could be tailored to the web service Hydra is scaling.

4.3. Proactive vs. Reactive Scaling

Proactive scaling and reactive scaling both play a part in the traditional methods of scaling web services. Proactive scaling comes about where designers of a system attempt to predict the load a service will experience and allocate enough servers for the predicted usage. An example of reactive scaling would be when a web service experiences a sudden influx of traffic and more servers are added to the current web service to take some of the load. It is important that our system provides a consistent quality of service, which could naively be accomplished by starting many more servers than is predictably needed. This would unnecessarily increase the cost. It was decided that any amount of proactive scaling would just increase cost unless the Hydra system couldn't reactively scale fast enough. A major strength of automating the reactive scaling is that the Hydra framework can maintain constant vigilance on the web service and perform the reactive scaling in a much more timely fashion than a system requiring intervention would. While it was seen that in most cases only reactive scaling should be necessary, for cases where it is assumed Hydra could not scale quickly enough the framework should support proactive scaling.

4.4. Load Balancing

Load balancing proves to be a problem with distributed web services because without it, some instances may become inundated with traffic giving a poor quality of service to some users, while other instances sit idle. It is possible that some services, such as a performance testing cluster, may not have to balance requests and would scale to meet a certain overall performance or quota, but these are not as common.

To balance the load across a web service, a couple of techniques such as using a proxy and domain name system (DNS) load balancing techniques were considered. In a system where certain instances are recognised as load balancers, these could distribute the re-

quests evenly amongst the instances they know about. This could be done using an existing software package such as HAProxy, but this introduces a point of failure in the system as the whole service relies upon a functioning load balancer. A second method of load balancing is through the DNS. DNS servers can perform load balancing by alternating the order of IP addresses are returned in response to a DNS request. This would allow the DNS to suggest different instances to each subsequent request through a "Round-Robin" technique. This solution does not take into account the load on each service and an overloaded service will still likely receive a $(1/\text{Number of listed instances})$, proportion of the requests. If one of the mapped servers were to fail, the DNS can take up to three days to update.

The Hydra framework is different from many load balancing systems because Hydra can have tailored load information about the servers to balance requests across. Instead of distributing requests evenly amongst the available services, requests can be more intelligently distributed by the load they are performing. This gives a good incentive for using a customised load balancer.

Many services require session state where interaction between a particular server and a client is tracked. In a stateless system, each request can be handled by a different server, allowing the load balancer to constantly map new requests to the least loaded server.

When this is not possible, the load balancer may have to somehow keep track of which servers clients should be mapped to. A smart client would be able to allow the service to redirect it to a server to use and if that server were to fail, the client could re-ask the load balancer which server it should communicate with.

Another problem with balancing requests is that client behaviour can change. Many clients could be mapped to a server without creating too much load. If the nature of the requests were to change the server may suddenly become overloaded. An example of this could be an online auctioning service. These auctions generally run over a predefined period of time. As an auction progresses, requests may be orientated towards getting the current value of a bid on an item. Most of the time, the returned result could be a cached value that does not need much computation. As the deadline for the auction approaches, requests could change to more computationally heavy bid requests and it becomes less likely that requests for the current value can be cached. The change in behaviour could overload the server. A smarter client could again be redirected to the load balancer by the server to find non-overloaded server.

A simple implementation of a load balancer could proxy all incoming requests to the least loaded server.

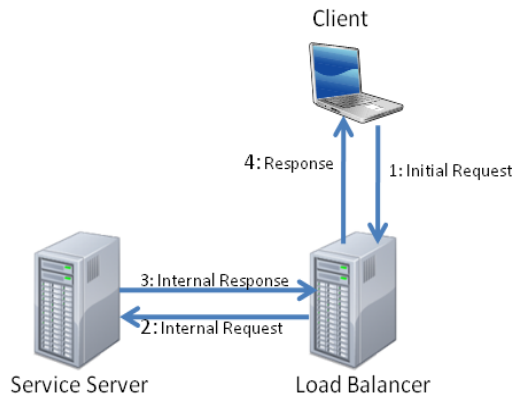


Figure 6 A load balancer redirecting traffic to the optimal server.

In Figure 6, the initial request is sent to the load balancer. The load balancer will have information about the load on each server and can pass on the request to the least loaded. Once that server produces a response, the response can be redelivered back to the client. A load balancer implemented in this fashion has the potential to overload as all requests must go through it. It could however keep track of session information and allow for redistributing load more easily.

The second approach is to redirect clients once to a suitable server(Figure 7).

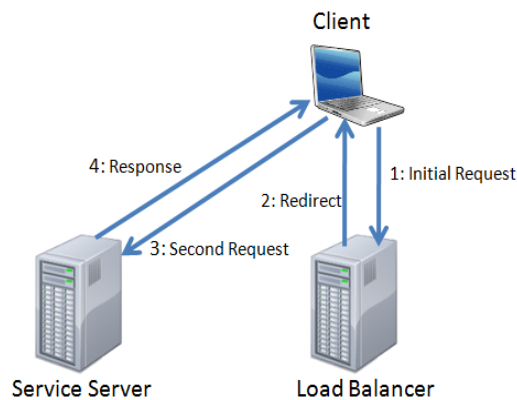


Figure 7 A load balancer redirecting traffic to the optimal server.

In this example, the client makes a request of the load balancer and the load balancer simply redirects the client to an optimal server. With this approach, there are also a few issues. Once a client has been redirected to a server, if the server was to fail, the client would have to have code telling it to re-query the load balancer. Furthermore, if the server the client was accessing started to become overloaded, the client would be stuck with that server. Again

client code could allow the server to request the client to query the load balancer for a new server to access.

5. Framework

Java was used to implement the Hydra framework, as Java's five primary goals fit in very well with the use and development of the framework. The framework can be extended upon for use on multiple operating systems, Java has built in support for using computer networks and Java is designed to execute code from remote sources securely[11].

The framework is designed so that virtually any method of controlling the network or choosing when to scale is possible.

5.1. Architecture

The program which controls these instances will be built in a modular fashion. There are five standard modules. These can be seen in Figure 8.

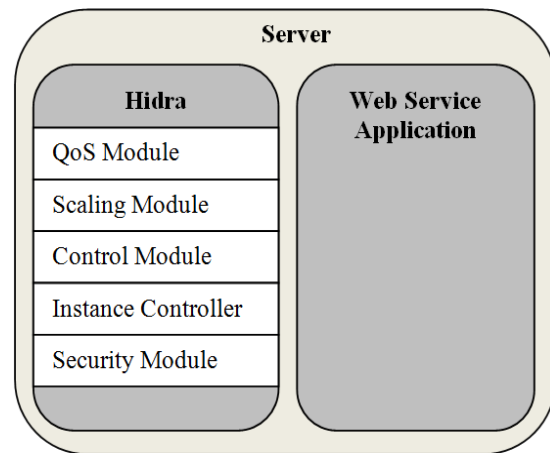


Figure 8 Deployment arrangement of a machine instance, running a web service and the scalable application with its constituent modules.

In this figure, the server is running two programs. One program (on the right hand side) is the web service application, the second is the Hydra program which is made up of five modules. Each module controls a part of how the overall service will behave. The instance controller communicates with the cloud computing service to start and stop instances. The control module determines how the servers will organise themselves. The scaling module determines what conditions are necessary to start or stop servers. The quality of service module determines how overloaded the system is and the security module authenticates inter-server communication.

5.1.1. Quality of Service module

The module for calculating the quality of service will return a status value, which is a simple integer value out of

100 that roughly indicates the systems usage as a percentage. This will then be used to determine if that system is overloaded. In a simple case this value may be calculated as a direct mapping of the total percentage of CPU usage. For a different module the value it returns may be the maximum of CPU usage, memory usage, and number of connected sessions out of 50.

5.1.2. *Scaling Module*

The second module will control how the system is scaled. It will also be responsible both proactive and reactive scaling. Proactive scaling could be controlled by using a schedule to determine the minimum number of running instances at any time. If it was known that between 4pm and 11pm each day was a busy period, it would be possible set a minimum number of three instances during this time, and a minimum of one instance otherwise. It will also be possible to schedule for days. The second proactive feature that could be implemented in this module is an algorithm for monitoring the values returned by the QoS module. Using these values, this module can look for patterns which may forecast an increase in future activity, allowing it to pre-emptively scale the system, or switch QoS monitoring modules to a more sensitive module.

This module can support reactive scaling by analysing the load values returned by the QoS module, starting or stopping virtual machines as needed. One implementation of this model could be to start a new instance if 80% of the current instances are overloaded. Another implementation could make use of two threshold values, one that starts a new server if the 70% of the system is overloaded and one that starts three servers if the system is 85% overloaded.

5.1.3. *Control Module*

The control module is in charge of which instance is in charge of starting and stopping instances. Although this system is designed to have a distributed approach, for many applications at any point in time there will be one instance in control of starting and stopping additional machine instances. This is to stop multiple instances sending a request to start a new server when the service is overloaded, resulting in too many starting. This instance will be elected by the control modules and will rule the system until it becomes unavailable, at which point a new control will be elected.

The control module is in charge of starting or stopping any load balancers and can start or stop external applications such as the web service application it is scaling.

5.1.4. *Instance Controller*

The instance controller performs two functions. Firstly it handles all communication with the cloud computing

vendor. The other modules use the instance controller to find the list of running instances and any modifications to this list go through the instance controller which synchronises the changes with other Hydra instances in the group.

5.1.5. *Security Module*

The security module can authenticate communication between the instances. One example of how this module could be used is to encrypt load values and digitally sign them so that messages from malicious servers do not manipulate the scaling of the system.

5.2. **Rationale**

The reason for building this program in a modular fashion is that the required functionality of each module may differ depending on the implementation of the web service. It is possible that modules may be required to change at run time.

An example where this is of use is in the quality of service module. In addition to the scale of requests changing over time, the types of request may change as well. A banking service for example may require a quality of service module that measures the number of requests to an instance per second throughout the day. At midnight this service may start to processes all the requests at once. When this occurs, the utilisation of resources changes from a bandwidth intensive service, to a computationally expensive service. The scaling module could change the active quality of service module to one which is sensitive to CPU usage, allowing the number of active services to be utilised towards CPU usage.

The scaling module could also be swapped, to allow for different safety factors that provide a more or less consistent quality of service. For instance, a system aiming to minimise cost may only start additional instances when all available instances are overloaded. Alternatively a critical system may add instances as soon as there is an average load across the system of 50%.

Having a modular control module allows different organisations of control. For a small system, it may be sufficient to have each of the instances publishing their status directly to the controlling instance. On a much larger system, it may make sense to have a deeper hierarchy where the controlling server polls a few deputy servers who in turn monitor a small group of instances. It is also possible to use a module to specify a single instance which will always be in control of the system.

5.3. **Risks**

The system is designed with a distributed form of control in mind. There is no single point of failure with this approach because while the control server or any of the additional instances may become unavailable, the system

will recover. It is still possible to create a control module which makes the system behave in a centralised manner, and where the controlling instance would be a central point of failure. For a distributed control structure, there is still the possibility that if the service becomes divided, each part of the system will rule itself. Since the system scales on a load basis, each division of the system will work to manage load and the overall effect should not be much different than a single overall system. This could also introduce problems with two subsystems modifying the DNS.

It is also possible to design a control module which only samples the load on one server to scale the service. This would be a reasonable assumption if the load was balanced uniformly across all service instances. If this is not the case, the particular web service being monitored may be overloaded while a significant proportion of others available services are not. In this case the controlling application would continue to start unnecessary new instances.

Finally the web service may be subjected to malicious distributed denial of service (DDOS) attacks. These work by flooding the web service with requests so that there are not enough resources available to service legitimate requests. It may be that a firewall or proxy may be able to filter out many of these false requests. The Hydra system could respond by scaling up to a point where it can handle all the illegitimate as well as the legitimate requests. The problem with this approach is that it could become financially costly. If this is an issue, then the Hydra system will have in place the ability to allow a human operator to be notified of a rapid scale of the web service, or even approve the scaling before a large number of extra instances are initiated.

6. Example Implementation

PeerWise was used as a web service for testing the Hydra framework and Amazons Elastic Compute Cloud was used as the cloud computing vendor. Architecturally PeerWise is a three tier system with an application tier written in PHP and a MySQL database tier for the data persistence. To scale this system, ideally both the PHP application tier and the MySQL database tier would be scaled by their own deployment of Hydra servers. By separating the two tiers a tailored set of modules could be used to scale each set and these could focus on the needs of the particular tier. With any multi-tiered system it is likely that the best deployment would be a group of Hydra controlled instances for each of the tiers. This is because it overcomes the problem of finding the right number of servers for each tier, by scaling each tier to fit that tiers demand. Scaling databases however can be difficult and for the purpose of the PeerWise implementation, only one tier of servers was used with each server containing the PHP application and a corresponding MySQL database.

This would mean that students using the system would only be able to share and answer questions with other students on the same server.

It was important to keep a low response time for PeerWise because it is often used as a studying tool for students. Studies have shown that users tolerate around eight seconds of delay before retrying a request or leaving a website[3]. Usage patterns can be expected to increase before tests or exams. As a result availability during these periods is considered important. Using information regarding test schedules and studying patterns of students it was hoped that a proactive scaling scheme could be evaluated.

A customised set of modules were created for use with PeerWise and EC2. These modules were focused around the needs for a website service and could be reused, as-is, for any similar website application. These modules also worked independently from the PeerWise system meaning that no PeerWise code had to be altered, to use our system.

6.1. Instance Controller

The instance controller was the module specific to Amazons EC2 service. The instance controller manipulated an underlying object model of the active server instances modelling these as "AWSMachineInstances" and primarily took care of communication with Amazon.

In addition to starting servers on EC2, two other services were utilised for the PeerWise implementation. Whenever an instance became a leader, the Instance controller would bind the current instance to one of Amazons Elastic IPs. This allowed the domain name to be mapped to the elastic IP rather than the leading instance. Whenever the leader changed, the elastic IP can effectively change the machine a domain name resolves to very quickly. The second additional service was Amazons simple storage. The Simple Storage Service can be used by modules to reliably store information such as the record of load values. For the PeerWise implementation it was only used to store the Hydra modules but it was hoped that in future, information such as historical load values and load balancing mappings could be stored reliably in a simple storage bucket.

Amazon provides a library of Java tools to perform actions with the EC2 service. These tools were integrated into the instance controller so that instances could be started or stopped with a simple method call.

6.2. Control Module

The method we used to control the PeerWise Hydra instances was to use a two level hierarchy with one leading instance with the remaining instances reporting their load to the single instance. This leading instance can handle all the requests for starting and stopping instances, pre-

venting the problem where multiple servers make the decision to start instances simultaneously.

To make sure that there was always a single leader, a leader election algorithm was used. The algorithm selected was for timed asynchronous distributed systems [12]. One of the requirements for this algorithm was a reliable multicast communication. This is currently unsupported between Amazon machine instances and as such a Java framework called JGroups was used that implemented this functionality.

6.3. Load Balancing Module

A load balancer was implemented for balancing HTTP requests to PeerWise. Each time a request comes into the service, it goes through the load balancer. The first time a request comes from a particular IP, the IP is mapped to the current least loaded server. This works well if the usage pattern of users is expected to be consistent, which is more or less the case with PeerWise. One alternative approach was to balance every request, but this introduces the problem of migrating sessions amongst servers and could be too slow as every incoming request would need up to date load information on the servers to be balanced. The second alternative approach was to balance the load based upon a Round-Robin algorithm where requests are given to each server in turn, or in the case of a two-tier Round-Robin, to a subset of servers that are not overloaded[13].

6.4. Quality of Service Module

The metrics used for the quality of service is closely related to the performance needs of PeerWise and the hardware on each machine instance. Each Amazon instance has 1.7 GB of memory, 160 GB of storage and a 32-bit virtual CPU core with performance equivalent to a 1 GHz processor. Each PeerWise instance requires significant CPU processing in the PHP application and the database, so this was chosen as the main metric for determining an overloaded server.

6.5. Scaling Module

The scaling module for PeerWise made scaling decisions based upon the number of overloaded servers providing a service. In order to account for a certain amount of unbalanced load by the load balancer, the service would only scale once a certain proportion of servers were overloaded. For this implementation 50% was used. To decide if a particular server was overloaded, the average load for the past ten load values had to be greater than the threshold of a 60. Likewise the least loaded machine would be stopped if 50% of the servers dropped below a threshold of 40% usage. New values come in approximately every 10 seconds, so new servers would start if half the running servers had an average CPU load of

over 60% for the past 100 seconds. The scaling module would also detect crashed or faulty instances by finding instances which haven't supplied a load value in the past 100 seconds.

6.6. Security Module

The security module for the PeerWise system simply accepted all communication. Since the deployment of the system was not publicly known and the working of the Hydra system is not publicly available it was safe to assume that security wouldn't be a problem for the evaluation.

7. Evaluation

The Hydra framework was evaluated over a LAN network in the University of Auckland computer laboratories and briefly on Amazons Elastic Compute Cloud. In the LAN environment JMeter, a Java performance testing framework, was used to test a cluster of three machines running the PeerWise system and an Instance Controller implemented for Lab machines. When the load balanced across the machines reached the threshold values, the machines successfully requested a new machine to start. Likewise when the load dropped below the minimum threshold, a request was sent to the least loaded machine to shut down. We were also able to simulate a crash on the leader of the deployment to initiate an election that resulted in a new leader being elected.

When the deployment was tested on the Amazon Elastic Compute Cloud, synchronisation code that was written for lab computers failed to work across the Amazon network. This has since been amended and results of performance tests should be generated in the near future.

8. Future Work

In the future it is hoped that modules for different kinds of web services will be developed. For the current set of modules, finding optimal threshold and combinations of scaling metrics would help improve the systems performance.

For scaling, evaluating the usefulness of a proactive scaling scheme could be evaluated. It is hypothesised that this is unnecessary as it is believed the system can reactively scale fast enough to keep up with the load.

Another area which could be evaluated is the different methods of load balancing. Load balancing is well researched area and comparing a load balancing scheme with an existing load balancing application such as HAProxy could highlight areas that could be improved.

9. Conclusion

Web services, especially stateless services and ones with smart clients, can easily be scaled with an autonomic framework. The challenge of resourcing physical servers to expand a web service can be overcome with the use of a cloud computing service. The Hydra framework displays self-configuring behaviour by allowing modules, which partition behaviour of parts of the system, to be changed at run time, as well as an elected control system where the system elects a leader when one crashes or becomes unavailable. Hydra is self-healing as it masks the failure of any servers. The number of servers running is kept at a minimum to provide a consistent quality of service. This self-optimisation also minimises the cost of running the overall system. Lastly all communication between the framework and the cloud computing service, or amongst the running services can be encrypted to allow self protection. This has been displayed by an implementation of the Hydra framework for the PeerWise system in both a Laboratory LAN environment and on an remote Amazon Elastic Compute Cloud.

10. Acknowledgements

The student would like to thank his project supervisor Dr Ian Warren, second examiner Dr Gerald Weber and project partner Nick Irvine for their support and guidance. Acknowledgment should also be given to Mike Culver for his introduction to Amazon Web Services, and Paul Denny for the use of the PeerWise in the evaluation of the Hydra framework.

11. References

- [1] W3C. 04/05/08, 2008; <http://www.w3.org/TR/ws-arch/#whatis>.
- [2] A. Bosworth, "Developing Web services. Data Engineering, 2001. Proceedings. 17th International Conference on." pp. 477-481.
- [3] P. George, and H. K. Randy, "Effective web service load balancing through statistical monitoring," *Commun. ACM*, vol. 49, no. 3, pp. 48-54, 2006.
- [4] J. O. Kephart, and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41-50, 2003.
- [5] IBM. "Take the tennis to 1.9 billion viewers worldwide? Done.," 04/05/08, 2008; http://www-07.ibm.com/innovation/au/ausopen/pdf/CaseStudy_01.pdf.
- [6] W. Aaron, "Computing in the clouds," *netWorker*, vol. 11, no. 4, pp. 16-25, 2007.
- [7] Amazon.com. "Why Use Amazon Web Services?," 01/05/08, 2008; <http://www.amazon.com/Why-Use-AWS/>.
- [8] Mike Culver, "Webscale Computing.," Seminar at University of Auckland, 2008
- [9] Paul Denny. "PeerWise," 04/05/08, 2008; <http://peerwise.cs.auckland.ac.nz/>.
- [10] 3tera. 13/09/08; <http://www.3tera.com/AppLogic/What-it-can-do.php#scale>.
- [11] Sun. "Design Goals of the Java Programming Language," 26/08/08; <http://java.sun.com/docs/white/langenv/Intro.doc2.html>.
- [12] A. Amintabar, A. Kostin, L. Ilushechkina, "A Leader Election Protocol for Timed Asynchronous Distributed Systems," 2006.
- [13] C. Michele, and S. Y. Philip, "Adaptive TTL schemes for load balancing of distributed Web servers," *SIGMETRICS Perform. Eval. Rev.*, vol. 25, no. 2, pp. 36-42, 1997.
- [14] L. Chia-Hung, B. Ying-Wen, L. Ming-Bo et al., "The saving of energy in Web server clusters by utilizing dynamic sever management." pp. 253-257 vol.1.