Restoring TCP sessions with a DHT

Peter Boers
peter.boers@os3.nl

Wednesday 3rd August, 2016

Abstract

Traditionally web scale applications use various load balancing techniques to maintain TCP sessions. Usually this means sharing all TCP session information over a number of load balancing appliances. This model is unsustainable in very large infrastructures. Our research proposes a novel method of maintaining TCP session integrity. It sets out to investigate if storing TCP session information can be done in a Peer-to-Peer, Distributed Hash Table, overlay network. By evaluating a proof of concept, we conclude that storing TCP session information in a DHT, might provide a reliable and scalable solution to maintain TCP session integrity in the event of network failures.

1 Introduction

With the ever increasing amount of nodes in networks there comes an ever increasing demand on the management of the network. How does one make it possible to keep a dynamic environment with thousands of Virtual Machines (VM) and various services online, clustered and balanced. In their recent draft RFC about their network architecture, Facebook has indicated a trend in network design and implementation [1].

The trend these days is to move towards IP based networks. Large parties such as Facebook are starting to rely on protocols such as the well established Border Gateway Protocol v 4 (BGP) as an Interior Gateway protocol instead of OSPF or IS-IS [1, 2]. In their networks each host is takes part in the BGP protocol to announce routes into the network. With this new network model there might be an opportunity to use the protocol properties of BGP and its protocol extensions, to create inherent redundancy in the Network layer.

With Web-Scale infrastructure, load balancing is becoming an issue as they need more and more bandwidth to be able to handle the throughput of these large infrastructures.

Typical load balancing devices operate in Active/Passive mode and use technologies such as Virtual Router Redundancy Protocal (VRRP) to maintain high availability. Furthermore load balancers share information about sessions to make sure TCP session are consistent. As they are mostly located at the edge of a network, they can create bottlenecks and single points of failure.

The technical report by Facebook indicates that the wish is to remove the load balancer at the edge of the network and do more traffic engineering and balancing on the network itself [1]. Equal Cost Multi Path routing (ECMP) is a protocol that together with Anycast will be able to do this. All hosts that are part of the service will announce their membership of the Anycast group and ECMP will be used to be able to do a more fine grained balancing over the network[3].

Anycast has been adopted by the global Root Domain Name System (DNS) zones to create a highly available infrastructure which is transparent to the users of the DNS service [4]. It is well known that it works well with stateless communication such as DNS, but that it is more difficult to use with stateful connections such as the Transmission Control Protocol(TCP).

In moving towards this Anycast and ECMP architecture, there is still a problem that needs to be solved. In the case that ECMP gets recalculated through an event on the network due to a failure or a reconfiguring of paths, a TCP session still needs to reach the right Anycast host to maintain consistency. This research will target this problem and discuss a possible solution for this problem.

Distributed Hash Tables (DHT) have long since been used to store information in a distributed way. As web scale architecture handles vast numbers of TCP sessions, if load balancing is removed from the edge of the Networks to the hosts, there must be a way in which hosts can look up TCP session information in the case that a host receives an invalid TCP session due to a topology change. We propose to use a DHT to store TCP session information and design a proof of concept, which we use to compare to more traditional load balancing methods.

2 Research Questions

Section 1 has defined the problem and hinted towards a solution here we state the direction of the research and help scope the project:

How can a DHT be leveraged to maintain TCP session state in the case of a failure in a Large BGP networks with thousands of hosts [1]?

- What technical requirements are needed to maintain the TCP session in the case of a failure?
- Does using a DHT to look up invalid sessions provide enough performance so that the session can continue?

3 Related Work

In the past there have been a number of efforts into researching the feasibility of using Anycast as a method to establish stateful resiliency in the network layer. What follows are a number of contributions about Anycast, load balancing in general and scaling in large systems. The following related work establishes a context and gives information about a number of solutions that already exist to establish scalable networks and load balancing.

3.1 Network Protocols

The TCP protocol is the workhorse of Internet traffic. It is a stateful protocol that ensures a coherent end-to-end bytestream, from one application to another[5]. Research has been done to combine Anycast with TCP for load balancing solutions. The prime method that a number of these researches have proposed is an extension to the TCP protocol [6]. Miura et al propose a modification of the TCP protocol so that it can handle the changing of the TCP socket tuple. This would make it possible that Anycast can be used for the initial set up of the connection and then the connection would be established on a different unique Internet Protocol(IP) address of the server. Even though it is well thought out in theory, in practice this has not been implemented. We believe that in searching a solution to the research questions, we need to find a solution that does not necessitate a protocol change as this greatly lowers the chance of it ever being implemented.

Weber et al evaluated potential methods of setting up Anycast in IP version six (IPv6) networks as Anycast was designed into the protocol [7]. In their survey they also touch upon the problems that exist when using a stateful protocol. In previous papers it was stated that using the source routing option in the IPv4 header could be an option. However as this is no longer recommended by the Internet Engineering Task Force (IETF). Due to security reasons it has been stripped from the protocol [8].

3.2 Load balancing

Load balancing is typically the area of distributed systems research. There are a number of papers that describe how this can be used in combination with Anycast. Castro et al, experimented with Anycast and Chord to create a group in which resources were shared [9]. They used it to be able to schedule jobs, manages storage and manage bandwidth.

A comparative study from 2010 also establishes the fact that the load balancing model of today where a small number of instances are responsible for the distribution of load is not going to be feasible in the near future [10]. In their paper they mention a number of different approaches to load balancing in cloud computing where the cloud is self aware and can balance the load, within the application, between instances.

Research into combining load balancing and DHT has been mainly done to improve the load distribution in a Peer to Peer (P2P) network. [11] proposes a method for nodes in a p2p ring to become proximity aware and balance the load between close neighbors.

3.3 Distributed Systems

This research looks towards the field of distributed systems research to help find solutions to the problem of scaling. The area of distributed system computing may provide a number of solutions and strategies which could be employed. Examples are DHT implementations and scaling strategies.

3.3.1 Distributed Hash Tables

Performance of DHT networks has also been widely scrutinised. As the scale of a network increases the look ups will inherently take longer however they always take $O = log(n)$ time where n is the amount of nodes in a network [12]. This achieves predictable scaling, however the question is if it is feasible to incorporate this with handling TCP sessions.

Chord is one of the most well know examples of a DHT, however the globally used Bittorrent network has adopted the Kademlia protocol as DHT overlay [13]. This Peer-to-Peer overlay network is easier to implement in software compared to Chord as it makes use of the XOR operation whilst calculating routing tables. Both systems achieve the same search efficiency and are both a viable choice to implement in our design.

3.3.2 Scaling

The report by Facebook and Arista networks[1], the writers indicate that a key feature in their design is that the network need to be horizontally scalable. Our solution to the load balancing challenge must meet the same requirements to be a viable solution otherwise it will not be usable.

To help define how to scale in large systems we fall back on Distributed Systems scaling theory by Tannenbaum and van Steen [14]. In their book they describe that scaling can be done in various ways, but that you can identify a number of areas that can be used to categorise types of scaling. Prime examples of this are; Administrative scaling, Geographical scaling, Functional scaling, etc.

In conjunction with categories of scaling they also state that two scaling strategies exist; scaling up and scaling out[14]. Scaling up means buying bigger and better hardware, a strategy advised and employed by vendors of traditional load balancers. Scaling out means buying more identical devices and connecting them together to solve a capacity problem. Our solution needs to be compatible with the latter scenario as this is the growth strategy that Facebook and Arista propose in their report [1].

4 Traditional load balancing solutions

The design of load balancing solutions is very much a problem of handling scale. What is the right design for the amount of hosts and traffic that you need to handle? Depending on the scale Distributed systems theories will come into play regarding the maintenance of a consistent view of the network and application. Load balancing architectures can be done on many layers in the OSI stack. They can be done on network layer, transport layer and even on the application layer. When looking at solutions on the network and transport layer we see the following options that sysadmins have for their infrastructure.

4.1 Hardware load balancing

Consider the traditional approach to load balancing as shown in Figure 1 on layer 3/4. Here we see the Internet represented by the cloud, an edge router and then two hardware load balancers in the box with the dotted line who are in sync with each other. They provide a balancing solution for a number of back end servers that are able to reach both balancers through the network. For the outside world and the inside world the two balancing nodes seem like they are one single unit.

These boxes share a session table and have some sort of keep alive mechanism which will trigger a fail over from the active unit if it fails. If the appliances work correctly during a fail over, the users and the servers will not know that anything happened. This system is limited by the amount of bandwidth that the appliances are able to support and the amount of sessions that you can track in the session table. The obvious solution to scale this, is to buy bigger load balancers and increase their capacity. So called scaling up[14].

These appliances can implement a number of different algorithms that will attempt to share the load of incoming requests over the available hosts in its balancing pool. Examples of these algoritms are: Round Robin and Weighted Least Connections [15].

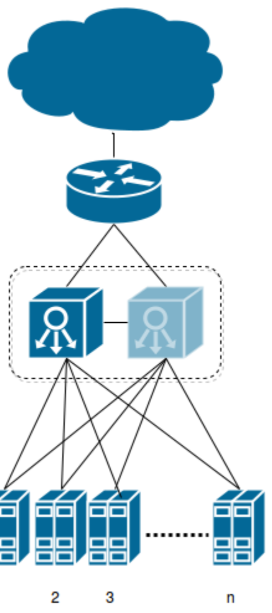

Figure 1: Load balancing with an active (dark blue)/passive(light blue) setup. This setup scales up to n hosts depending on the capacity of the Load Balancers

Next to various load balancing algorithms the balancers are also capable of using techniques such as TLS offloading, compression, proxying, caching and TCP connection marshaling. The latter meaning that a balancers will marshal multiple external connections through one single connection to a backend device.

4.2 Software load balancing

The previously section describes solutions that are available in hardware and describes appliances that can be bought and used. However there are also a great deal of possibilities of installing software load balancers. Prime examples are Linux Virtual Server(LVS)[1] and HAProxy[2]. LVS can be used similar to hardware solutions. This package of Linux can be installed in a virtual machine or on a bare-metal server and can function as a load balancer for a cluster.

LVS can be configured in an active/passive mode where the high available pair shares a virtual IP address and is able to fail over if a keep alive timer has timed-out. As you can see from the description LVS functions as a traditional load balancer, and only works on layers 3 and 4 in the OSI model.

HA proxy is another well known software load balancer in Linux which can work on layer 3 and 4, but is really designed to proxy services who use the HTTP protocol. It is optimized to provide good performance towards high traffic sites and can be tuned towards a specific application. It is often used together with LVS. HAProxy is able to do the application level balancing and LVS takes care of the Layer 3 balancing.

Another example is a package called "conntrackd", which synchronises the connection tracking table between high available hosts. Often this is uses together with "keepalived" as configured together they can do similar things as LVS [16].

4.3 Scaling traditional architectures

The above examples operate on the premise that there are either one or more appliances that are active at one point in time. Secondly, that all appliances in the load balancing cluster share the same view of the network and that they are transparent towards the application and user.

By adding various technologies such as compression and offloading they attempt to make the application more resilient and reduce the amount of traffic on the network so that the appliances can scale better.

In doing so, these appliances and software packages are becoming more and more complex, as they need to be able to handle a wide variety of situations. Further more by letting the load balancers do such a wide variety of tasks, debugging problems on the network become more difficult and if a failure occurs on the balancer, this may have serious ramifications.

4.3.1 How to handle more connections?

Even though appliances that do balancing are getting more and more efficient, at one point one needs to scale even more. This can be done by adding more appliances and creating an active/active setup with tens of nodes. These nodes will all be aware of each others states and back end servers and provide a consistent view of the application they are trying to balance to the outside world.

By operating on the philosophy that all nodes in the network need to know the complete state and have the complete state in their own memory, it is easy to see that in Large Scale Networks with thousands of hosts, this is not possible anymore. It simply will cost too many resources to keep track of all this information and to make sure that all nodes have the same view.

Furthermore a second reason this does not scale is the fact that the network topology as shown in Figure 1 requires a lot of investment to upgrade. Lets assume that as your applications grows your balancer is no longer able to handle the traffic and incoming connections. If you are running an active/passive setup as shown in the figure, you will need to buy two new appliances instead of one. In the case of a failure of one of the nodes the other must be able to take over all of the traffic seamlessly, meaning that simply buying one new device will not be good enough as the fail over device will not be able to handle all of the normal traffic.

Alternatively if we assume that the setup is active/active, this means that we would need to buy one or more appliances to come back to a state which is acceptable for the network. Depending on the architecture and failure domain, a 2 node active/active setup may never operate above 50% capacity as if one fails the other must be able to handle all traffic if the network is to survive. Managing scaling in this scenario requires carefull planning and fore thought.

4.4 Handling Failures

Traditional load balancers handle failures differently depending on which layer they operate. In general, when a load balancer detects a failure on one of the real servers, it will make the failed node inactive and no longer forward any traffic to it. Depending on the type of load balancer and how it is configured, TCP sessions will be seamlessly handed to a new server or it may need to be reestablished.

5 New Network Design

As stated in the previous Section 4 the more traditional solutions to scaling a large network assume that all load balancing nodes are equal and that all states need to be synchronised between the members of the load balancing group. Due to the aforementioned problems considering scalability and complexity, the draft RFC by Facebook and Arista tries to tackle the problem in a new way where their mantra in creating a network design is defined as follows:

[1] http://www.linuxvirtualserver.org
[2] http://www.haproxy.org

'Environments of this scale have a unique set of network requirements with an emphasis on operational simplicity and network stability [1].'

5.1 Using proven tech - Routing

The solutions that Facebook and Arista propose for the network design form the basis of the scenario and tests of this research. In their design they assume that the network topology is based upon layer 3. By doing this they alleviate many problems that Layer 2 has, when it is necessary to scale up and out. At one point traditional layer 2 without enhancements, will no longer handle the traffic within one single broadcast domain, especially with thousands of hosts. This means that in the new design, all nodes on the network take part in the routing protocol of choice, which in this case is EBGP.

They have chosen this design and protocol to enable the network to scale faster horizontally and because BGP is relatively simple to setup and use. OSPF and IS-IS are widely used as Interior Gateway routing protocols but have much more protocol overhead, and are more difficult to configure and debug. Figure 2 shows an example topology as proposed in the RFC design.

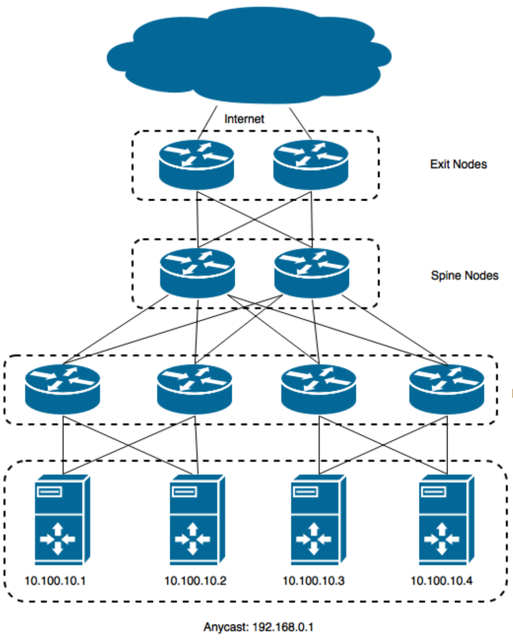

Figure 2: This is the topology on which the experiments were conducted. The interconnects are shown and the Anycast group with their unique identifiers.

By using the design in Figure 2 it is easy to see how this would scale horizontally. If you need more ports or bandwidth, the only thing you need to do is add more spines and leaves. This allows the network to scale with exactly the amount of ports and traffic that the device you just added to the network allows, meaning it is very predictable and cost effective.

5.1.1 All nodes are Routers

In the past traditional designs of networks have chosen to have a minimum amount of routers due to the fact that it is very costly to buy appliances that support these routing protocols. However, recently vendors such as Cumulus Networks are able to deliver white label boxes to customers. These devices are able to run an operating system like Linux and have specific optimized network drivers for performance [17].

This enables users to install an open source routing daemons like Quagga. The Quagga daemon is a lightweight program that has implemented various network protocol standards[18]. Overnight it has become possible to run a routing protocol cheaply on all sorts hardware without a user having to pay for license fees.

The architecture of Figure 2 has now become possible.

5.1.2 BGP unnumbered

To alleviate management problems in a network such as this, it is necessary to make make sure that routers are added to the BGP topology easily. The BGP protocol depends on the setup of a session with a neighbor[19]. When this is done correctly it will be able to share routing information through this session. Normally, this requires manual input and management. An extension to the BGP protocol by Cisco, allows administrators to setup BGP sessions automatically by making use of certain intricacies of IPv6; Neighbor Discovery(ND) and Router Advertisements(RA)[20].

Simply put, this extension relies on the fact that each interface of a router will automatically configure a Link-Local IPv6 address with StateLess Address AutoConfiguration (SLAAC). To make sure that duplicate addresses are not chosen, the ND protocol is used and by then making use of RAs the routers are able to setup a BGP session and share routes on the IPv6 local link. Figure 3 gives a good overview of how it works.

5.1.3 Link state detection

It is well known that IGP protocols such as IS-IS and OSPF are capable of detecting Link state changes and rerouting packets very efficiently, BGP however is not as good at doing this in a standard configuration. Normally it sends keep alive messages every 30 seconds or so to its neighbor and changes its routes according to the information this generates.

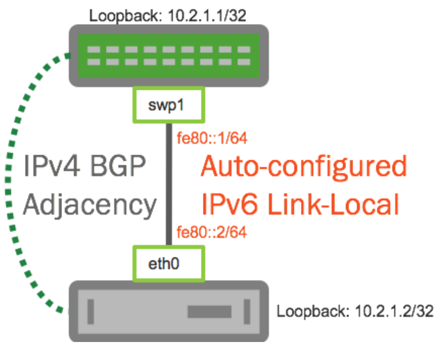

Figure 3: The implementation of BGP unnumbered according to RFC-5549[20]

In this topology this is not fast enough. However together with the BGP unnumbered configuration it is possible to make BGP detect link failures by binding it to a specific interface. Therefore if the Kernel detects a Link State change on the interface, BGP will update its routes accordingly and thereby trigger a topology change on the network[21]. This is called setting up Interface Scoped BGP sessions.

5.2 Load Balancing

Now that the topology has changed, the new design no longer allows for the traditional load balancing solution on Layers 3 and 4. If an administrator would choose to do this he/she would greatly compromise future ability to scale out horizontally. The question of sharing load across the servers however remains. The RFC by Facebook and Arista states that load balancing will be done on network level using the principles of Anycast and Equal Cost MultiPath (ECMP) routing.

5.2.1 Anycast

The Anycast principle has been used to great effect in the global DNS root server infrastructure, enabling DNS requests to be routed towards the closest root DNS server according to the length of the path to the IP address. The theory being, that if the *same* IP address is announced to the network from *multiple* locations, routers will automatically choose the route to said IP address over the least amount of hops[19]. In the case of DNS this makes sure that requests are serviced from the nodes closest to the requester and it makes sure that the load can be evenly spread throughout the infrastructure.

The new network design will make use of these characteristics. All servers in the network shown in Figure 2 (the bottom layer) take part in an Anycast group. Each server announces a specific loopback address to the world on which it hosts a service. In this case it is the address “192.168.0.1.” However when one looks closely at the topology, you can see that the amount of hops from the Internet represented by the cloud at the top of Figure 2, to a Server, is always three hops. This is regardless of the route a packet takes in the network and assuming that packets always take the shortest route from the Internet to a Server.

Facebooks and Aristas use of Anycast represents the membership of the server in a specific group. It enables a server to become available in a group of servers that are hosting a specific application, by announcing an Anycast IP address to the network.

5.2.2 ECMP

The previous paragraph showed that the locality of a node no longer influences the route a packet takes on the network to a server, as all paths are of equal length. Instead, the path of a packet is manipulated by making use of ECMP[22]. The name of this protocol extension already suggests that it can be used in a case such as this, as the routes from Internet to Server flow along multiple paths of equal cost.

In principle ECMP calculates routes by hashing the source of a packet and then choosing the path that it needs to take. All results of the hash are mapped to certain buckets and if the result of the hash falls into a certain bucket, the router knows what the next hop of the packet will be. Figure 4 shows a schematic representation of how ECMP works. Incoming packets are mapped to a certain bucket and then forwarded to the corresponding next-hop.

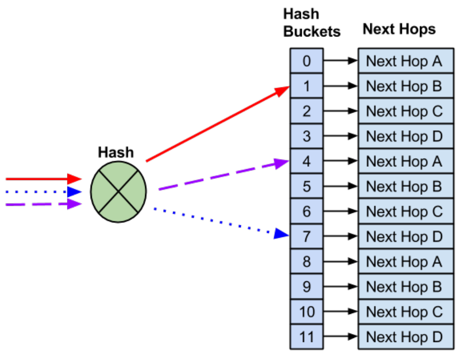

Figure 4: A reprensentation of the hashbuckets in the ECMP protocol

ECMP can be manipulated on routers by adding a weight to an outgoing interface. This means that administrators can easily control and balance the load of traffic on the network over all hops and servers on the network. It allows a very fine grain control of the flows and is just as effective as the means of distributing load that traditional solutions use.

5.3 Handling Failures

In this design, when a failure occurs on the network, or if an administrator cycles a router on the network, ECMP will recalculate the routes according to the the available interfaces towards the destination. However, traditional solutions allow for TCP session state synchronisation across the load balancing appliances. It makes sure that the integrity of the TCP session remains intact when the connection is passed on to the next node. The client only notices a minimal hiccup, but is still able to access the service.

The new design does not cater for this, because when ECMP recalculates, the TCP session will be forwarded to a different server. It will not recognize the session and will send a TCP Reset (RST) message to the client as per the protocol standard. In doing so the clients TCP session is terminated, which depending on the use case is something that is undesirable.

5.3.1 Maintaining the TCP session integrity

As the new network solution no longer has a limited amount of nodes over which session states need to be synchronised, it is not realistic to expect that all TCP sessions are synchronised across the platform. If this would be done it would cause an quadratical growth in network management traffic every time a node is added. Such a network can be described as a fully connected network (complete graph) [23]. This does not scale, for obvious reasons.

As stated in the Research Questions in Section 2 we propose to solve this scaling problem by implementing a Distributed Hash Table solution to store the TCP session identifiers to help maintain the TCP session state.

6 New load balancer design

The new network design requires a new way of maintaining TCP session state across a very large network. Section 5 makes it clear that the old solution of synchronising state between all nodes in the Anycast group is infeasible due to the problem that traffic overhead increases quadratically, every time a new node is added to the network.

This Section defines what a scalable solution is for this problem, it states what protocols are involved and finally shows the design of a system that can solve the problem.

6.1 Distributed systems

Scaling has been a much discussed subject in the research community. In the past M.Hill asserted that saying that something is scalable does not even mean anything [24], however in recent years the consensus is that scalability is a key requirement in the area of Distributed systems and can be split up into a number of different elements[14].

In their RFC Facebook and Arista talk about wanting to achieve horizontal scalability [1]. This can be seen as a form of functional and load scalability. Of course there are many other aspects of scalability that come into the network design as proposed. However, if one looks purely at achieving the most efficient design then functional and load scalability definitions match the closest.

The theory of distributed systems resides around creating applications that through a middelware layer, can communicate with each other to allow more nodes and systems to work together for coordination and communication purposes. In this way the middelware helps the infrastructure to achieve the goal of the application, by combining the efforts of all nodes without one having to know or do everything [14].

6.1.1 Maintaining state

As we have seen, the challenge that this research tries to solve is the question of how to maintain TCP session state across thousands of servers. We believe that the most efficient way of doing this is by using the principles defined by distributed systems. If one looks at the topology you see that all nodes in the Anycast group are equal participants without a centralised coordinator. This makes the Anycast group an ideal candidate to use the Peer-To-Peer (P2P) overlay model. P2P overlays operate on the premise that all nodes in the network are equal participants and have equal resources [25].

6.2 Distributed Hash Table

A discovery that lead to a great change in how files and information is shared on the Internet is the introduction of the Distributed Hash Table, Chord [12]. First introduced in 2001 it sets out to create a way in which P2P overlays are able to store and share information on the network. In their model, they propose an efficient way of searching a piece of data on a P2P overlay which is made out of a ring of nodes. Simply put, by mapping pieces of information to a key, and making all nodes on the overlay responsible for a part of the key space, they can combine all the resources of the P2P overlay and not have to store duplicate information.

DHT implementations come in various shapes and sizes and this research uses the Kademlia protocol to implements its DHT [13]. The reason for this is due to the fact that it has been widely adopted in P2P networks. A notable example being the Bitorrent network [26]. The Kademlia search algorithm enjoys wider adoption as it makes use of the XOR operation to calculate where the information is located on the network. The maintenance of routing tables and calculation of routes is therefore less costly compared to the Chord implementation [27]. The search algorithm implemented by the Kademlia protocol achieves a predictable search in the following order:

$$O(n) = log(n)$$

If the node on the network does not have the information located at its own location, it can find the information in $log(n)$ hops, where n is the amount of nodes in the DHT network.

6.2.1 Maintaining state

To facilitate the storage of information in the P2P overlay we propose to use an implementation of a DHT. In this way we do not need to replicate information to all nodes on the DHT, but we do make sure that the information can be looked up in an efficient and predictable way. Ensuring this will mean we can make assertions over how scalable the new load balancing design is.

6.3 Transmission Control Protocol

The Transmission Control Protocol (TCP) is the defacto standard for providing reliable, end to end data transport over the Internet. TCP is implemented directly above layer 3 on the OSI stack and provides an interface for the application to transport a byte stream over a network. The TCP protocol can be best described as a state machine with a role for the client and a role for the server. The states of the machine describe the life cycle of TCP session setup, data transfer and session tear down. By sending sequence numbers and acknowledgements the client and the server ensure a reliable transfer of data from one point to the next [5]. Figure 5 shows the states that a TCP session can have at have in a certain instant.

6.3.1 Tracking TCP connections

A TCP connection is defined as the relationship between a socket on the client side of the connection and a socket on the server side of the connection. A socket is a tuple which combines an IP address and a port number. The IP address identifies the interface on which a packet arrives and the port number identifies the process to whom the information is addressed. Together the client and server sockets are the TCP 4-tuple [29].

TCP connections can be tracked by the Linux Kernel and this provides information about the status of a TCP session between a client and a server [30]. On the server side this can be used to see if the TCP session is valid. In the case of an

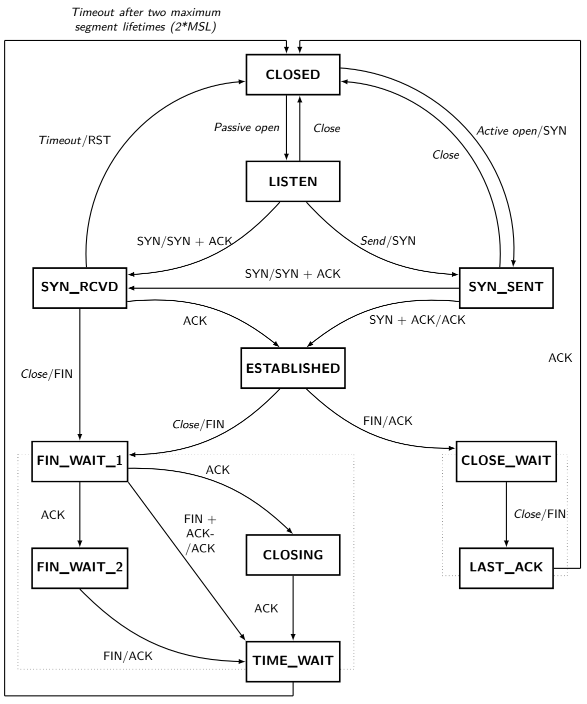

Figure 5: The TCP state machine for the server and client. The server is the left side of the picture and the client is represented by the right side of the picture [28].

erroneous packet, the server will send the client a RST message. This means the socket becomes free on the server side and the client will need to re-initiate the connection. The connection tracking system detects validity of a session by looking at the acknowledged bytes in the TCP header. If the acknowledged bytes is greater than zero, the protocol knows that the packet is not the first packet of a session and therefore invalid.

In our proof of concept we need to act upon such a misplaced session. Instead of directly sending a RST message to the client we need to lookup the session on the DHT to see if it can be associated with a different node in the Anycast group. If so, we can forward the packet to the correct host. Otherwise, we act as the protocol dictates and send the RST message anyway.

6.3.2 Maintaining State

To maintain session state we propose that every new session to the Anycast group is stored by the server on the DHT. As you can see in Figure 2, servers announce two loopback addresses; One is the Anycast address and the other is their unique IPv4 address. The information stored on the DHT represents a TCP 4-tuple. The `client socket` will act as a key and the `unique loopback address` and port serves as the value.

Figure 6 clearly shows how a search on the DHT

```
{ "20.1.1.1:1234" : "10.0.0.2:80" }
```

Figure 6: This is an example in JSON of the information that will be stored in the DHT.

would work, in the case of an erroneous session. When an unknown connection is received, the TCP client socket information is used as a key to look up the identifier of the correct node.

6.4 Combining all elements

When combining the previous information we propose the following design to solve the problem of maintaining TCP session states across thousands of nodes in an Anycast group. The proof of concept will be made up of the following elements:

- A network Topology implemented according to the new design.
- A group of servers taking part in a Kademlia, DHT overlay network.
- A group of web servers who listen on an Anycast address and look up and store TCP session information in the provided DHT.

Figure 7, Figure 8 and Figure 9 show the different interactions that happen between client, server and Kademlia network.

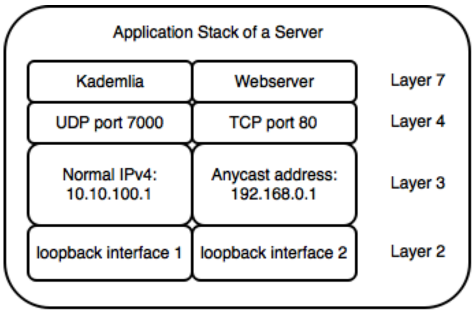

Figure 7: This shows the applications running on the server. The DHT is listening on UDP port 7000 and uses this to listen to messages on the overlay. At the same time it is hosting a web application on port 80 on the Anycast address.

6.4.1 Summary

The proposed design can in theory achieve horizontal scalability. When implemented well it is possible to add and delete nodes from the network without having to explicitly inform every other node on the network. Nodes join the web server group by announcing the Anycast address and can join the DHT by knowing one single other node in the DHT.

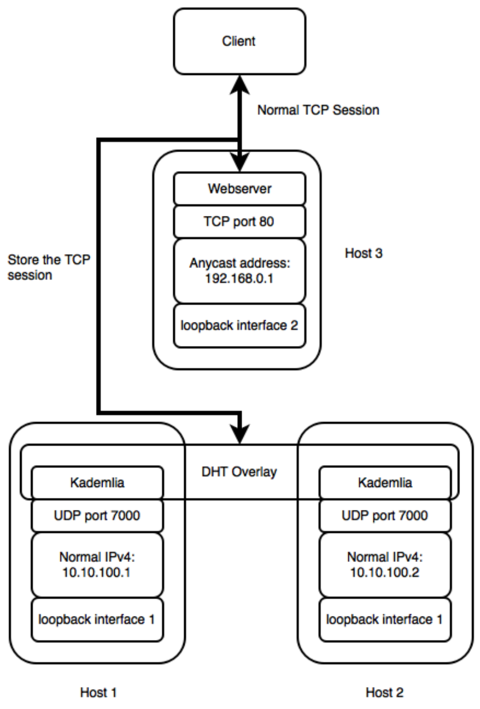

Figure 8: This shows the interactions between client, web server and Kademlia network when following a normal flow. We see that upon session establishment asynchronously the session is stored on the DHT network.

As the DHT search is very predictable this means administrators can calculate maximum search times that are needed to look up invalidated sessions.

7 Method

The new network design as laid out in Section 5 shows how load balancing will be done on a Large BGP network. However a question remains about how to make sure you can preserve hundreds of thousands of TCP session states across such a large network. This Section will lay out a scenario which we will test and evaluate to see if the Research questions can be answered. The test scenario and results will be gathered by building a proof of concept.

7.1 Proof of concept

The design of the load balancer as elaborated upon in Section 6 was implemented with the scripting language `python 2.7`. It was implemented in such a way as to make sure that TCP requests were not blocked by look ups in the DHT, to approach a real world scenario as close as possible. This means that the web server was running in one thread and the DHT in another thread.

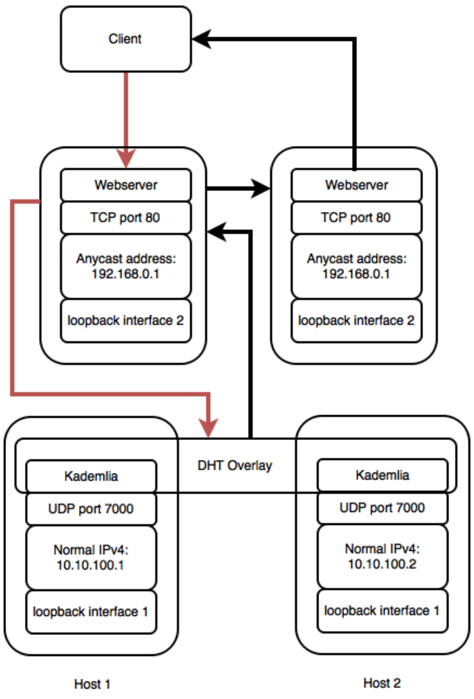

Figure 9: This figure shows what happens in the event that an erroneous packet is sent to the web server. When you follow the arrows in anti-clockwise direction you see that the server will look up the client socket, forward the packet to the original server and the original server will service the session. This keeps happening until TCP session termination.

7.1.1 Web server

The web server was implemented by using the socket library provided by python. It served a simple binary file towards any connection that connected to it. All new connections to the web server are asynchronously entered into the DHT.

7.1.2 DHT

The Kademlia protocol is implemented by using a library available on Github. The overlay is setup by starting the network with a single root node and from that point adding nodes to it by bootstrapping from one of the existing nodes on the Kademlia network. The bootstrapping process makes sure the node can find its correct place in the overlay. All communication between nodes on the overlay is done using UDP.

7.1.3 Sniffer and Forwarder

If the server detects a wrong packet it looks up the correct destination in the DHT overlay, unpacks the IP header, removes the Anycast address, inserts the correct destination IP, recalculates all checksums and sends the packet towards the correct node. The look up on the DHT overlay is only done once as subsequent look ups can be done by using a local cache. Figure 10 shows the changes to the ip header to reroute the traffic to the right node. The value in red is the value that is stored with the client TCP session key on the DHT.

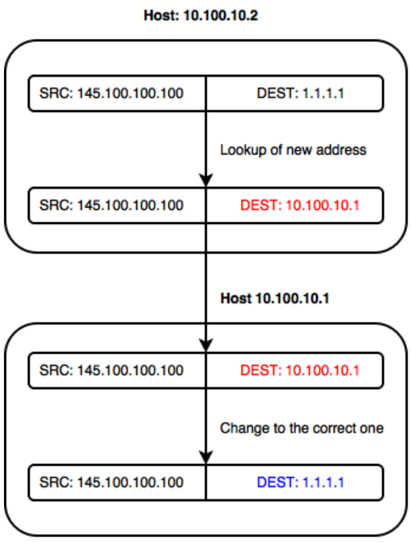

Figure 10: This shows the changes that are made in the IP header on each host when an incorrect packet arrives.

7.2 The Scenario

The scenario of this research assumes there is a network with four servers connected to a number of routers that allow access towards the Internet. The servers in this topology represent machines that host a number of web services on TCP port 80. Together these servers take part in a Distributed Hash Table overlay network and announce the shared Anycast address that hosts the web service with BGP. A visualisation of the topology is seen in Figure 2. Furthermore we assume that it is configured and set up as described in Section 5, the new network design.

7.2.1 Testing the proof of concept

During testing we assume the role of a visitor of the Anycast site, who is downloading a large file. The goal is to ensure that we have an active and valid TCP session during the time of the testing and that we are able to ensure TCP session continuity if the test is successful. Prerequisites for this test are that time is synchronised in the topology and that the DHT overlay and web server are started successfully.

We define the following metrics as being relevant. Combining them will tell us what the performance is of the DHT:

- **Setting time:** This is defined as the time it takes to set a key-value pair on the DHT overlay. It is measured from the moment when a packet is sent on the network to the moment it receive an acknowledgement packet.
- **Detection time:** This is defined as the time it takes for the network to detect a link failure on the BGP topology, up to the moment the overlay reacts to an ECMP rerouted packet. According to the design this means the overlay will react to the first ECMP rerouted TCP packet.
- **Look up time:** This is defined as the time it takes for a node to look up a key value pair on the DHT overlay network. It is the time it takes to send a question and receive a result from the network.

To measure these metrics we established the following procedure:

1. From one of the exit nodes of the topology establish a TCP session on the Anycast address to download the file.
2. Collect the time stamps that are printed before and after storing the TCP session in the DHT. This is the DHT "setting time."
3. Simulate a link failure on the network by manually shutting down an interface over which the traffic is flowing.
4. Let ECMP recalculate the path of the traffic so it will be forwarded to the next Anycast node.
5. Collect the time stamp in the system log for when the Kernel detects interface shutdown and compare it with the time that is printed just before the script looks up an entry in the DHT. This is the total "link state detection time."
6. Collect the time stamp before and after the session look up on the DHT, this is the "look up time" of the network.
7. Verify the TCP packet is forwarded correctly and the TCP session is still active.

These steps were repeated a number of times to calculate an average time for the three metrics so we can analyse if the results are predictable and consistent.

7.2.2 Comparing to traditional solutions

With the results gathered in the testing phase, we can compare the achieved performance to three traditional load balancers available on the market today. Traditional balancers use health checks and timeout timers to measure availability of nodes in their group. Depending on the timer thresholds they are configured to act in a certain way.

What follows are a number of definitions that will help understand how traditional load balancers work.

- **Health Check:** A traditional load balancer can be configured to check the functionality of an application or simply the response to an ICMP request to judge if a node is responsive.
- **Check interval:** A load balancer will perform the health check at regular intervals. This can be configured with the "Check interval."
- **Attempts:** Upon failure of a check, a balancer can be configured to make a number of further attempts before it will take action.
- **Timeout timer:** When a check fails, the "timeout timer" defines the maximum amount of time a balancer may wait before it needs to take action.

A combination of the aforementioned definitions are used to configure load balancers to detect a failure. This is fundamentally different compared to the new design as it depends on different types of inputs and not on a stray TCP session

Table 1 shows how these checks can be configured in the case of three vendors of traditional load balancers; Amazon Webservices software balancer [31], Kemp Technologies hardware appliance[32], f5 networks hardware appliance [33].

7.3 Hardware and software used

The topology was built using certain hardware and software. This section briefly explains the hardware and software that has been used to conduct the experiments.

7.3.1 Hardware and Operating Systems

The topology was created by making use of Virtual Machines (VMs) with the Virtual Box provider. The topology needed to be capable to run 16 VMs. To be able to run this with some speed we used a hardware node with the following specifications; two, `Six-Core AMD Opteron(tm) Processor 2435`, with `64GB` of RAM and a RAID 1 setup of two `60GB` SAS hard drives. This node was running a fully patched `Ubuntu 14.04.4 LTS` operating system with `Virtual Box version 4.3.36`

The Virtual Machines could be split up into two types: Servers and Routers. The server machines had the following specifications; One Core of the `Six-Core AMD Opteron(tm)`

Product	Health check	Check Interval	Attempts	Timeout timer
AWS	ICMP/TCP/HTTP	30 (5)	2 (1)	5
Kemp	ICMP/TCP/HTTP	9 (3)	2 (1)	4
f5*	ICMP/TCP/HTTP	5	-	16

Table 1: The health check configuration offered by three major vendors. All values represented are default values. Known minimum values are shown in between "( )".
f5*: The timeout setting should be three times the Interval setting, plus 1 second [33]. f5 do not configure attempts.

`Processor 2435`, with `400MB` of RAM and a logical volume of around `4 GB`. The server VMs ran a fully patched version of `Ubuntu 14.04.4 LTS`.

The router VMs have the same hardware specifications, but run `Cumulus Linux 3.0.0`.

7.3.2 Software Installed

The hardware node uses Vagrant[3] and Ansible[4] to spin up the VMs that run the topology. The source of VagrantFile and packages were provided by Cumulus Linux and are available in https://github.com/packetninja. The server VMs required the following python packages to run the proof of concept which was responsible for creating the prototype: `pydht, pynetfilter_conntrack, ipy`. The source of the proof of concept is available here: https://github.com/pboers1988/dhtsession. The following Linux package was also installed from the repositories to run the code: `libnetfilter-conntrack3`

7.3.3 Settings to run the topology

The Linux Kernel does not track incoming connections with a vanilla install. To enable connection tracking the easiest solution was to create an `iptables` rule which instructed Linux to track all incoming connections in the connection tracking table:

```
iptables -I INPUT -m state --state
↪  NEW,RELATED,ESTABLISHED,INVALID,
↪  UNTRACKED -j ACCEPT
```

As some of the logic of the python script depended on certain states of the TCP session it was necessary to make it fast enough to be able to intercept the states of the session in the connection tracking table. As the TCP stack is very efficient in Linux we needed to introduce a delay on the network card to help the script cope with the speed and make it able to handle the logic. The delay was introduced on both links towards the leaf nodes in the topology with the `tc` tool:

```
tc qdisc add dev eth1 root netem delay
↪  10ms
```

Lastly, default TCP behaviour dictates that all connections that come in an interface, who are not recognized, will receive a TCP reset (RST) packet back. For our experiment we disabled the TCP RST packets as this would mean the client would close their socket while we were trying to reroute the traffic to the correct destination. The following `iptables` rule was used to achieve this goal:

```
iptables -A OUTPUT -p tcp --tcp-flags RST
↪  RST -j DROP
```

8 Results

In this section we provide the results of testing the proof of concept according to the definitions we stated in Section 7, "setting time," "detection time" and "look up time." We gathered data over a number of tests. Last of all we show if the Proof of concept worked as expected.

8.1 Setting time

The setting time is defined as the time it takes to set a key-value pair on the DHT overlay of 4 nodes. It is measured by checking the time difference between the moment setting begins and an okay is returned. Figure 11 shows the results of 15 measurements.

8.2 Detection time

The detection time is defined as the time between the moment a router detects a link failure and the moment a rerouted packet is searched on the DHT. This time encompasses the recalculation of ECMP and may also include a number of retransmissions by the client, as the some packets may have been dropped during the link failure. Figure 12 shows the results of 10 measurements.

[3]An interpreter that is able to read definition files and interface with a number of VM managers to automate the provisioning of VMs
[4]A configuration management tool based on python

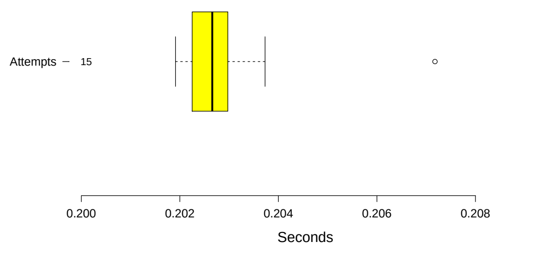

Figure 11: This plot shows the time in seconds that it takes to set the Key - Value pair on the DHT.

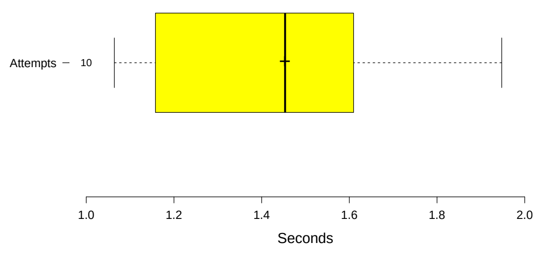

Figure 12: This plot shows the time in seconds that it takes between the failure of a link and detection.

8.3 Look up time

This is the time that is needed to look up a key on the network. It is measured from the moment the search starts until the result is received and can be processed. Figure 13 shows the results of 11 measurements.

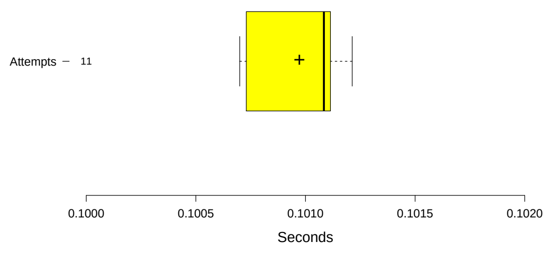

Figure 13: This plot shows the time in seconds that it takes for a node to look up a key on the DHT.

8.4 Proof of concept

The last step of the proof of concept has not yet been implemented. Our python script is able to look up sessions and forward them to the correct node, however the correct node is not able to recognise the packet for reasons we discuss later in the report (Section 9). Unfortunately this means the TCP session does not continue when the packet gets forwarded. Even though the proof of concept is not complete, this did not have impact on the gathering of the measurements, as those steps are implemented correctly.

9 Discussion

In this section we discuss the results of the research and point out some areas that need further investigation and work. In particular some areas of the proof of concept still need work to really see if this solution is viable.

9.1 Proof of concept

The proof of concept has been designed and built in a short space of time and at the point of testing was under active development. The design of the concept indicates that the last step of the chain is to reestablish the TCP session at the original node. This should happen when a packet is forwarded from the temporary node, who looked up a stray TCP session in the DHT. As the destination IP is changed during the forwarding phase from the Anycast address to the unique IPv4 adress of the original node, the original node does not recognise the packet(Figure 10). In its connection tracking table the server sees a relationship between the Anycast address and the client socket and not the unique Ipv4 address and the client. It therefore treats the packet just as it would treat a regular misplaced TCP session and sends a TCP RST towards the client, completely defeating the purpose of the new design.

It should be relatively trivial to capture these packets and forward them to the local Anycast address, or perhaps to encapsulate the original Anycast packet through TCP or UDP and transport it in that way from the temporary node to the original node, however this has not been implemented and is as yet untested. This is unfortunate as it makes any judgement about the success of this proof of concept up for debate.

9.2 TCP Protocol

Even though the complete proof of concept is not finished at the time of writing, most of the elements of the design have been successfully implemented. Furthermore the results show that it is not necessary to make an intrinsic change in the TCP stack, unlike Miura et al [6]. This design does not step out of the bounds that the TCP protocol requires and this research focuses more on how to implement it. Our solution states that before a TCP RST is sent, there needs to be a check to see if the session exists on a different node by looking that up on the DHT.

We believe that this model is very achievable, because the only change necessary needs to be done on the server side and is completely transparent to the client. Only changing implementation on the server side is much easier then changing things on the client **and** server side. With this design it should be possible to make the solution transparent for the TCP client.

9.3 Timing Results

In comparing the various timing results to more traditional solutions, we see that on the very small scale experiment that our solution is much faster. Our solution can achieve the rerouting of TCP packets within two seconds, this is much faster than the theoretical minimum of traditional solution, which is between 5 and 10 seconds. The results also show that setting, detection and look up with this system is very predictable and consistent. Where traditional solutions rely on methods such as ICMP to check if a node is still alive, our method reacts to a misplaced packet and forwards this accordingly. This is inherently much faster than waiting until certain arbitrary timers in health checks have failed. Our model assumes that when a packet is rerouted this means that a failure has occurred and a problem needs to be solved.

9.3.1 Scaling

Previously nodes could look up results in memory which is obviously very fast, however in our case network delay needs to be taken into account as we often need to query another node in the DHT overlay. Even though this may sound like a performance cost, our solution is very scalable. The network latency together with the fixed search amount of $log(n)$ that the DHT provides, means that look up time can be calculated exactly. These attributes are predictable which means admins can make choices about how far they would like to scale.

9.3.2 Calculating look up times

From the measurements gathered we can define the successful look up time in relation to the amount of nodes in the network to make it a predictable model. Let n be the amount of nodes in the network, then look up time T is dependant on the average look up time on every node L and the amount of nodes that need to be visited during a search on the DHT $log(n)$.

$$T(n) = L \cdot log(n)$$

L in this case contains the average network latency and search time of one hop. T provides a means to estimate the look up time of a key on the DHT overlay when a network increases in size.

9.3.3 Implement as a native application

As mentioned in Section 7 to make the application work we needed to induce artificial delay on the link. Furthermore, the `pydht` library was only evaluated in a rudimentary way. We believe that an implementation in a native binary of the proof of concept needs to be made to ensure the accuracy of the timing results. The measurements in this research are more of an indication to see if the network performs in a consistent manner.

9.4 Robustness

The robustness of this solution is still very much debatable. Our test case was to see if **one** TCP stream could be rerouted. In the case of a network outage, or in the case that multiple links fail simultaneously, there will be many more TCP sessions that need to be looked up within a very small time frame. It remains to be seen if a solution such as this is suitable for HTTP traffic. In previous research it has been shown that the longevity of a HTTP TCP session decreases logarithmically depending on the bandwidth between client and server [34]. A HTTP session could already be complete by the time a session is stored on the DHT.

It is well known and confirmed by research that with packet loss TCP performance is reduced significantly [35]. It could be easier to just setup a new session instead of trying to save the existing session.

9.5 Open issues

Quite obviously this solution still has to be tested on a larger scale with many TCP sessions and it remains to be seen if this solution will achieve what is necessary to recover those TCP sessions. The conclusions of this research are based on observations in a very small scale in a virtual environment whilst trying to recover one TCP session.

Secondly the last step in the chain, the reestablishing of the TCP session between the original node and the client has not been tested. This is of course the ultimate goal of the design and still needs to be implemented.

Lastly the integrity of the DHT needs to be tested: this research has not looked at how DHTs react to constant setting and removal of data.

10 Conclusion

This research has designed a new solution to TCP session integrity management in Large Scale BGP networks. In building the proof of concept, we have shown that a DHT could be used to store TCP session information and provide a means for a large cluster to spread out all the session information across a large network. Secondly on a small scale, the presented solution outperforms traditional solutions. It does not wait for timeout timers, like traditional solutions, but reacts on TCP packets who are rerouted by ECMP that are not recognized by new host on which they arrive.

Our design does not require any change in the TCP protocol, but could be made to work by changing some implementation logic on the server side. The client side of the TCP session can connect and interact with the server in a normal way. This makes eventual implementation much more achievable as it is transparent to the client.

Finally, the measured results indicate that in the case of one TCP session our solution performs in a very predictable manner. Even though we believe that the proof of concept is a relatively naive implementation which could benefit from a number of improvements, the look up results show that it is predictable and consistent. Along with the fact that DHT search performance scales efficiently, we think this model could be a viable solution for maintaining TCP session integrity on Large BGP networks.

11 Future Work

To continue effort in this area of research we identify the following opportunities:

- Finishing of the proof of concept by making sure the TCP session is restored.
- The testing of the proof of concept on a larger scale with more TCP sessions and more servers in the Anycast group.
- Measure realistic web traffic and see if the DHT is robust enough to handle typical bursty HTTP traffic.
- Convert the python script to a native binary or driver and measure the performance.

Acknowledgements

I would like to thank Attila de Groot for providing the initial idea and infrastructure to conduct this research. Secondly, Rama Darbha and Pete Lumbis of Cumulus networks for their feedback and patience during the course of the project.

References

[1] P. Lapukhov, A. Premji and J. Mitchell. *Use of BGP for routing in large-scale data centers*. Tech. rep. Technical report, IETF, 2016. URL: https://datatracker.ietf.org/doc/draft-ietf-rtgwg-bgp-routing-large-dc/.

[2] Office of the Secretary-General. *Chapter Two: Understanding Telecommunication Network Trends*. 2011. URL: https://www.itu.int/osg/spu/ip/chapter_two.html (visited on 30/05/2016).

[3] C. Hopps. *Analysis of an Equal-Cost Multi-Path Algorithm*. Tech. rep. Technical report, IETF, 2000. URL: https://tools.ietf.org/html/rfc2992.

[4] Wikipedia. *Anycast — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-June-2016]. 2016. URL: https://en.wikipedia.org/wiki/Anycast.

[5] Wikipedia. *Transmission Control Protocol — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-June-2016]. 2016. URL: https://en.wikipedia.org/wiki/Transmission_Control_Protocol.

[6] Hirokazu Miura et al. 'Server load balancing with network support: Active anycast'. In: *Active Networks*. Springer, 2000, pp. 371–384.

[7] Scott Weber and Liang Cheng. 'A survey of anycast in IPv6 networks'. In: *Communications Magazine, IEEE* 42.1 (2004), pp. 127–132.

[8] Valter Popeskic. *Source-based routing in IPv4 and IPv6 networks*. 2015. URL: http://howdoesinternetwork.com/2014/source-based-routing.

[9] Miguel Castro et al. 'Scalable application-level anycast for highly dynamic groups'. In: *Group Communications and Charges. Technology and Business Models*. Springer, 2003, pp. 47–57.

[10] Martin Randles, David Lamb and A Taleb-Bendiab. 'A comparative study into distributed load balancing algorithms for cloud computing'. In: *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on*. IEEE. 2010, pp. 551–556.

[11] Yingwu Zhu and Yiming Hu. 'Efficient, proximity-aware load balancing for DHT-based P2P systems'. In: *Parallel and distributed systems, ieee transactions on* 16.4 (2005), pp. 349–361.

[12] Ion Stoica et al. 'Chord: A scalable peer-to-peer lookup service for internet applications'. In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160.

[13] Petar Maymounkov and David Mazieres. 'Kademlia: A peer-to-peer information system based on the xor metric'. In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[14] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems - Principles and Paradigms*. Prentice-Hall, 2015.

[15] Wikipedia. *Load balancing (computing) — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-June-2016]. 2016. URL: https://en.wikipedia.org/wiki/Load_balancing_(computing).

[16] Pablo Neira Ayuso. *Conntrackd*. 2016. URL: http://conntrack-tools.netfilter.org/conntrackd.html (visited on 28/06/2016).

[17] Cumulus Networks. *Routing on the Host: An Introduction*. 2016. URL: https://support.cumulusnetworks.com/hc/en-us/articles/216805858-Routing-on-the-Host-An-Introduction (visited on 28/06/2016).

[18] Paul Jakma. *Quagga Routing Suite*. 2016. URL: http://www.nongnu.org/quagga/.

[19] Wikipedia. *Border Gateway Protocol — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-June-2016]. 2016. URL: https://en.wikipedia.org/wiki/Border_Gateway_Protocol.

[20] F. Le Faucheur and E. Rosen. *Advertising IPv4 Network Layer Reachability Information with an IPv6 Next Hop*. Tech. rep. Technical report, IETF, 2009. URL: https://tools.ietf.org/html/rfc5549.

[21] Cumulus Networks. *Managing Unnumbered Interfaces*. 2016. URL: https://docs.cumulusnetworks.com/display/DOCS/Border+Gateway+Protocol+-+BGP#BorderGatewayProtocol-BGP-ConfiguringBGPUnnumberedInterfaces.

[22] Wikipedia. *ECMP — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-June-2016]. 2016. URL: https://en.wikipedia.org/wiki/Equal-cost_multi-path_routing.

[23] Wikipedia. *Network Topology — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-June-2016]. 2016. URL: https://en.wikipedia.org/wiki/Network_topology.

[24] Mark D Hill. 'What is scalability?' In: *ACM SIGARCH Computer Architecture News* 18.4 (1990), pp. 18–21.

[25] Wikipedia. *Peer-To-Peer — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-June-2016]. 2016. URL: https://en.wikipedia.org/wiki/Peer-to-peer.

[26] Wikipedia. *Kademlia — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-June-2016]. 2016. URL: https://en.wikipedia.org/wiki/Kademlia.

[27] Jakob Jenkov. *Peer Routing Table*. 2014. URL: http://tutorials.jenkov.com/p2p/peer-routing-table.html.

[28] Ivan Griffin. *TCP state machine*. 2016. URL: http://www.texample.net/tikz/examples/tcp-state-machine/.

[29] Wikipedia. *Network Socket — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-June-2016]. 2016. URL: https://en.wikipedia.org/wiki/Network_socket.

[30] P Ayuso. 'Netfilter's connection tracking system'. In: *LOGIN: The USENIX magazine* 31.3 (2006).

[31] Amazon. *Elastic Load Balancing - Configure Health Checks*. 2016. URL: http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/elb-healthchecks.html (visited on 28/06/2016).

[32] Kemp Technologies. *Frequently Asked Questions*. 2016. URL: https://support.kemptechnologies.com/hc/en-us/articles/203863135-Web-User-Interface-WUI-#_Toc450210533 (visited on 28/06/2016).

[33] f5 solutions. *Manual Chapter: Configuring Monitors*. 2016. URL: https://support.f5.com/kb/en-us/products/big-ip_ltm/manuals/product/ltm_configuration_guide_10_0_0/ltm_appendixa_monitor_types.html#1172375 (visited on 28/06/2016).

[34] Joachim Charzinski. 'HTTP/TCP connection and flow characteristics'. In: *Performance Evaluation* 42.2 (2000), pp. 149–162.

[35] Anurag Kumar. 'Comparative performance analysis of versions of TCP in a local network with a lossy link'. In: *IEEE/ACM Transactions on Networking (ToN)* 6.4 (1998), pp. 485–498.