Multicast Routing with Load Balancing Using Amazon Web Service

Amit S. Rodge, Chandan Pramanik, Joy Bose, Sandeep Kumar Soni
Web Solutions
Samsung R&D Institute India- Bangalore
Bangalore, India
{amit.rodge, chandan.p, joy.bose, soni.sandeep}@samsung.com

***Abstract*—Web Services such as AWS have become popular in recent times, with many small and large companies wanting to take advantage of cloud computing technology and its possibilities. However, there is an issue with networks set up using such services, in that they do not allow multicast routing and any attempts to set up such a network are hampered because of security related issues that prevent packets moving across subnets in a public cloud. One can bypass such restrictions by using a central server in the AWS cloud as a multicast router to route packets and simulate a multicast network. However, an issue with such an approach is how best to perform load balancing, which includes hot addition and deletion, in the network. In this paper, we present a novel solution to the problem of load balancing in an AWS network where multicast is not supported. We then present the results of some tests on the system to show that the multicaster works on the AWS as well as performs load balancing with different mechanisms including hop and chain. We conclude that our algorithm can perform multicasting in a non-multicast supported network using a load balancer that provides hot addition and deletion in spite of the restrictions.**

Index Terms— cloud computing; Amazon Web Service, multicast routing; HOT addition/deletion; load balancer; HAProxy

I. INTRODUCTION AND PROBLEM STATEMENT

SPDY protocol [8] is becoming very popular over HTTP for providers of web services such as Google Web Service. To form a cluster of nodes or service discovery on a public cloud where the nodes support SPDY, each of the nodes needs to find details of the other nodes. The basic condition of this cluster formation is that every individual host participating in cluster formation should have knowledge of other existing hosts in same subnet constantly. For that we need mechanisms such as multicasting.

To determine the number of nodes in the network cloud and for purposes of scalability, load balancing is desirable. Currently no stable SPDY based load balancers are widely available. Load balancer is a module which balances incoming traffic and distributes its load to backend servers in round robin fashion so load gets equally distributed. To balance the load on SPDY based systems, we need a TCP load balancer such as HAProxy in order to forward SPDY requests. HAProxy takes care of hot addition or removal of any of the hosts from its backend, as well as check the backend host's existence by sending heartbeats continuously.

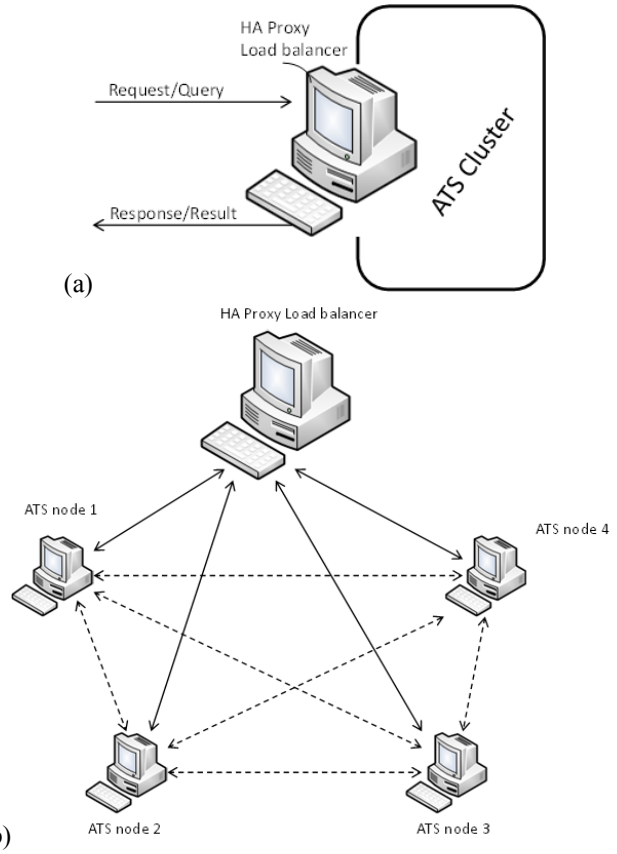

Fig. 1. (a) Illustration of the functioning of a load balancer and (b) Network architecture with 4 nodes and an HAProxy as a load balancer.

AWS [1] is a popular web service used by many providers to host their solutions on the public cloud. However, AWS and similar services do not support multicast [2]. Therefore if we need to support such services on a public cloud, currently it will not be possible.

One available solution that can support multicasting on a public cloud is by using VPN, but that solution is too complex, having issues such as setup, cost and complexity. Moreover, VPNs do not support load balancing either. Without load balancing we cannot scale easily. AWS does support load balancing but their solution is too costly. Besides, AWS only supports only HTTP load balancing and not SPDY, also the

978-1-4799-5364-6/14/$31.00 ©2014 IEEE

developer needs to integrate their multicasting solution with the AWS load balancer.

There is a related problem where the network supports multicasting and the hosts in the network are not willing to register for any multicast group specifically but are still interested to form a group for a defined purpose. This paper talks about how to address all such problems in an unsupported multicasting environment.

Fig. 1 shows the network architecture of our system with HAProxy as the load balancer.

The rest of the paper is organized as follows: in section 2 we look at related work in the area of load balancing in web services which do not support multicast routing. Section 3 introduces the proposed algorithm / network architecture for multicasting and load balancing for different scenarios. In section 4 we do a generalized analysis of bandwidth in each of these scenarios. Section 5 details the experimental setup to evaluate our algorithm, and Section 6 analyzes the results. Section 7 suggests some avenues for future work, and concludes the paper.

II. Related work

There exist a number of related works in the area of multicasting using cloud services.

Bencek et al [3] explained how to enable multicast support on AWS instances in a private cloud. However, this does not deal with the public cloud case, nor does it consider load balancing. The blog article by Barr [5] mentioned the method of elastic load balancing inside of a virtual private cloud, but here too the public cloud case is not handled.

A few people have worked on load balancing algorithms in a cloud environment. Khatua et al [4] spoke of optimizing the utilization of virtual resources in a cloud, including dynamic load balancing with virtualization. Chiba et al [6] proposed optimized load balanced algorithms for data transfer via multicast between an Amazon S3 server to multiple Amazon EC2 instances. Their work focuses more on the optimized distribution of network load and less on the problems with enabling multicasting with load balancing on a public AWS network, as this paper does.

III. Proposed algorithm / network architecture for multicasting and load balancing

In this section, we look at the techniques for multicasting to solve our problem of supporting multicasting in a non-supported multicast network with load balancing, where the hosts support SPDY protocol.

First, we reiterate how a load balancer works. A load balancer is present in the network and handles all incoming client requests which get distributed to the nodes in the network.

All nodes share the same subnet with the Loadbalancer host, whose function is to balance the network loads. The Loadbalancer host runs HAProxy or any other suitable component which does the load balancing.

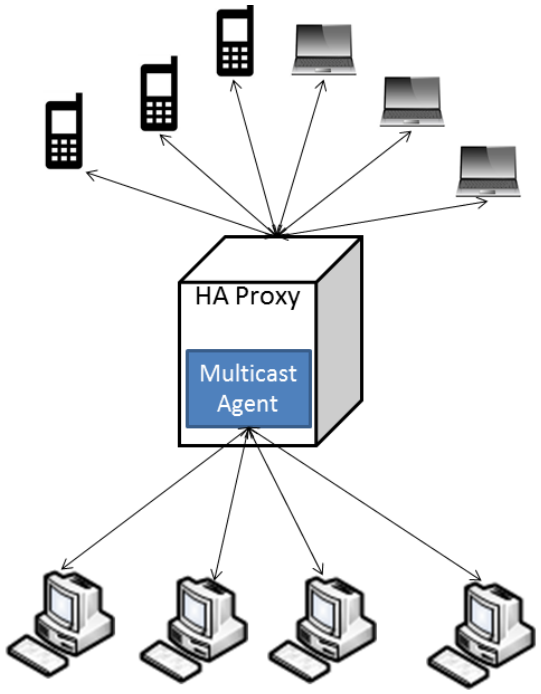

Fig. 2. Setup of the proposed multicast system over an AWS network

The Loadbalancer has a module running Multicaster (which implements the multicast routing of packets to different hosts) as part of the HAProxy, along with it. When all the nodes are turned on in the network, they start listening for incoming client requests from the Loadbalancer and service packets from other hosts. Along with this, the nodes also publish their own respective service packets to the Loadbalancer host.

At the Loadbalancer, the multicaster module listens for service packets from all the hosts. On receipt of a service packet from any one of the hosts, the multicaster just forwards the same packet to other hosts, except for the host from which it originally received the packet. The information about the other hosts can be known from the HAProxy component running on the Loadbalancer host. The HAProxy does health checks of configured hosts with it. Hot Addition or removal of hosts from the HAProxy backend is intimated to the Multicaster. In this way, it takes care of forwarding service packets to unintended hosts. Also, this is beneficial to other hosts who want to participate in a cluster. On hot addition, every other host updates its own table about the new host on the receipt of a service packet from the Multicaster on behalf of new host who is willing to share cluster with it. On hot removal of a host, other hosts wait for an anticipated timeout for service packets from the Multicaster for that particular host. Once it is timed out, it removes its entry from its own table.

Thus, the Multicaster unit assists in simulating multicasting with the help of HAProxy as load balancer for a group of hosts. Also, it translates the group into a cluster of hosts, enabling every host in the cluster to benefit from it in order to extend its application implementation.

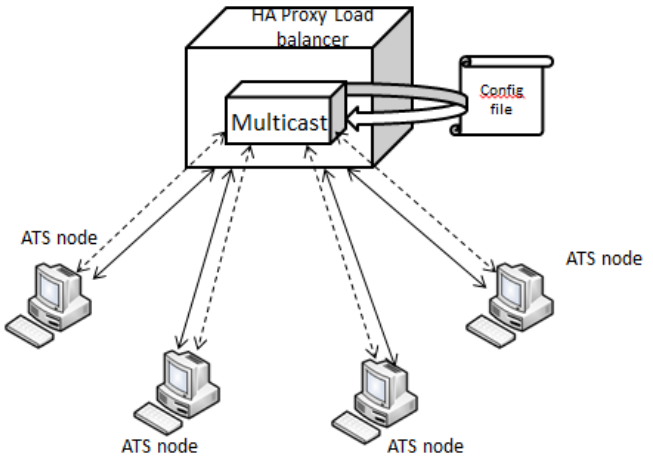

Fig. 3. Illustration of the scenario where multicasting is not supported, single/same subnet, and a single Multicaster along with HAProxy is used as a load balancer. Scalability is achieved by hot addition/deletion. Formula for bandwidth consumption : N(square)*(Packet size) , where N is number of nodes.

Regarding our basic network setup, in order to enable Layer 2 communication over the network, we have used n2n Peer-to-Peer VPN [7] software. This is bound to the internal Ethernet adapter of the AWS Virtual Public Cloud (VPC) instance. As n2n is a secure VPN server, by default it ships with encryption and compression enabled.

We removed encryption and compression as they are not required over an AWS VPC connection. We created a VPC with one subnet, in one availability zone. We used four "m1.xlarge" EM2 instances with public IP addresses. Software we used include Ubuntu OS (Saucy 13.10 amd64 server), n2n Peer-to-Peer VPN and ZeroMQ. Fig. 2 shows the setup of the proposed multicast system over an AWS network.

Now, we consider the working of different scenarios and network configurations. We have three network configurations on which our solution for multicasting with load balancing in an AWS public cloud is applicable: Single subnet, more than 1 region, multiple regions hop and chain.

A. Scenario 1: Single subnet in a single region

Let us consider the simplest scenario where our problem is applicable: there is a single subnet in a single region and there are a number of nodes that need to form a cluster. To form such a cluster, it is necessary that each of the nodes should have existence of the other active nodes. This can be achieved by using a multicaster to forward the packets received by one node to each of the other nodes.

In our solution, each of the nodes sends packets to the dedicated load balancer, on which the multicaster module is running. The multicaster in turn forwards the received packets to each of the other nodes except the node from it which it originated. For a network with 3 nodes named A, B and C, Node A sends a packet to the multicaster, which then forwards it to the other nodes B, C. In this case the number of hops is 3, since the packet goes from A to multicaster, multicaster to B and multicaster to C. On receiving the multicast packet, nodes B and C become aware of the existence of node A.

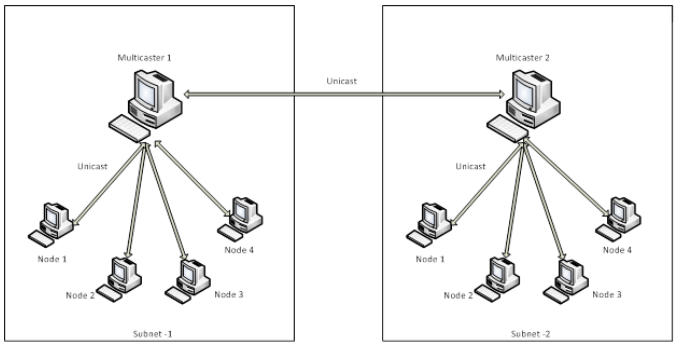

Fig. 4. Illustration of the scenario where multicasting is not supported, there are multiple subnet across regions, and a single multicaster along with HAproxy works as a load balancer for each subnet. Scalability is achieved by hot addition/deletion. Formula for bandwidth consumption = (2*N+1)*(packet size) where N= no of nodes in each region.

Similarly node B sends a packet to the multicaster which forwards to A and C, and C sends the packet which is forwarded to A and B. After these packets are sent, all the nodes will be aware of the other active nodes in the same subnet and hence a cluster is formed. So here, the total number of hops for all the 3 packets is 9. The bandwidth consumption shall be 9 times the packet size.

Generalizing from the above, for n packets travelled in the subnet, bandwidth consumption will be proportional to the square of the number of nodes in the network.

Bandwidth consumption = n^2 * (packet size)

Figure 3 depicts this scenario. Here nodes 1-4 are AWS nodes.

B. Scenario 2: Two subnets in different regions

Now let us consider a slightly more complex scenario: where there are two or more subnets located in different regions which are forming a cluster. This scenario is illustrated in figure 4. Here, there are 2 regions and n nodes per region.

We assume that the delay for one hop between any two machines is constant, and that if a packet travels from one node to another via an intermediate multicaster, we count two hops. In this scenario, a similar logic as the previous scenario would apply for delivery of packets and network bandwidth.

For the current subnet, the number of hops made by a packet sent from one node is n, since there are n-1 other nodes and the one multicaster unit. For each packet sent from a node in region 1 to a node in region 2 (the other subnet), it has to make a total of 1 hop to reach the destination region. Further, there are n destination recipients in the destination region. There is 1 path for 1 region from the current subnet.

Adding the three components above, we get the total bandwidth consumption for one node =

$= n + n + 1 = 2*n + 1$

The actual bandwidth, we multiply this number by the packet size.

C. Scenario 3: Hop and Chaining in different regions

In this scenario, packets are sent from one node to the next. Here too the number of hops will be same as in scenario 2, so the bandwidth consumption will be same.

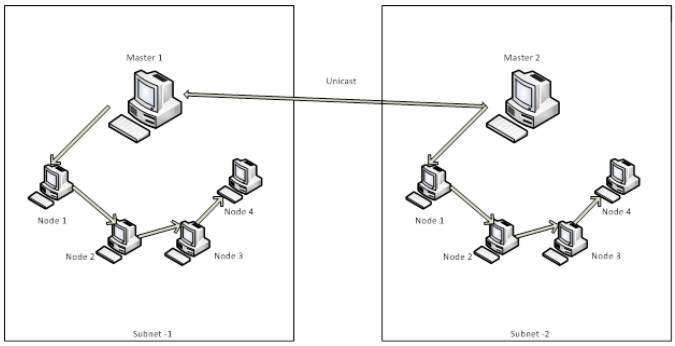

Fig. 5. Illustration of the scenario where multicasting is not supported, Multiple subnet across regions, Master for each subnet, Hop and chaining. Formula for bandwidth consumption :~ (M*N)*(M*N + M-1)*(packet size) where M = no of regions & N=no of nodes in each region. (approx.)

This scenario is illustrated in figure 5.

IV. Generalized analysis of delay timing

In this section we derive a generalized formula for the bandwidth consumption in different configurations of the network as discussed in the previous section.

A. Case 1: For n nodes and 1 HAProxy multicaster

For every node in a subnet, it needs to send 1 packet and for each 1 packet there are (n-1) destinations as recipients. So total bandwidth used for 1 node = sending + receiving = 1 + n-1 = n

Therefore, using this for all n nodes in a subnet

Total bandwidth consumption = n + n + .. (n times)

= n^2 = O (n^2)

B. Case 2: For m regions and n nodes per region

Let us consider a multi-region scenario with multiple nodes in each region. To simplify our analysis, we assume the number of nodes is same for all regions and that is equal to n, and the number of regions is m. There is one HAProxy multicaster per subnet.

For a packet to travel from one node in a subnet to another node in a different region, there are the following three components of its journey:

- For the current subnet, the number of hops made by a packet sent from one node is n, since there are n-1 other nodes and the one multicaster unit.
- For a packet to travel to the destination region there are (m-1) * n destination recipients, because there are (m-1) paths to reach to the destination region recipients from the current subnet.
- There are (m-1) paths for m regions from current subnet.

So the total number of hops shall be

= n + (m-1) * n + (m–1) = m*n + m - 1 = m * (n+1) – 1

The above formula corresponds to the bandwidth consumption is for one node sending packets.

Total number of nodes participating in cluster formation are m*n

Therefore total bandwidth consumption for all these nodes =m*n*(m*(n+1)-1)

=m*n*(m*n+m-1)

V. Test setup

In AWS, there are multiple backend hosts behind the HAProxy as Loadbalancer, as shown in the setup on figure 6. These servers have capability to fetch the content from outside and keep a copy with it before sending it back to client. There are N client requests simultaneously sent to the load balancer.

The HAProxy forwards these requests to backend hosts in round robin fashion. Every host in turn checks if the requested information present in its local cache, if found, replies the client through the load balancer. If not found, it fetches the same information from outside world. In case the same information exists in any of the other backend hosts, this cannot be known by the current host. To solve this problem, each host needs to know about other hosts in the same network. To understand about other hosts, it needs to form a cluster of hosts because of which all hosts can share information with others. Therefore, all hosts need to advertise about itself in network so other hosts could listen to it. This advertisement can be done using multicasting technique. In AWS, multicasting is not supported, so hosts cannot form a cluster in AWS environment. Effectively, hosts in AWS environment cannot share information from local cache with other hosts.

This can be solved by running multicasting agent along with HAProxy which helps in hot addition or removal of hosts in network. And Multicasting agent becomes very good reason to form a cluster for hosts in the network as explained earlier.

Once a cluster is formed, all hosts can form a dedicated TCP connection mesh among themselves. The existing local cache of host can be shared across cluster. Every host who receive client request checks its local cache first, if not found checks in cluster cache by sending TCP unicast before it receives it from the outside world.

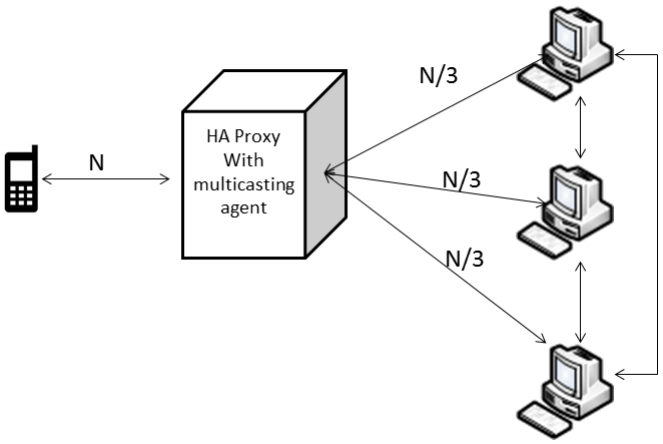

Fig. 6. Test Setup for 3 nodes

VI. EXPERIMENTAL SETUP AND RESULTS

We have performed an experiment to bring ATS nodes in a cluster on AWS, using HAProxy as a load balancer. The multicasting solution we described in this paper was running as part of the HAProxy load balancer. Because of the cluster we get a unified cache, also called clustered cache. Here, the HAProxy is used to do the hot addition and deletion, multicaster will help in bringing all the ATS nodes into the cluster, and individual ATS nodes will participate in forming the clustered cache. We have n number of ATS instances (Apache Traffic Server) forming a single cluster using multicast. This is needed in order to provide a clustered cache. We turn on all the ATS instances in AWS long with LB.

In this experiment we are measuring scalability and reliability of the network. We want to determine whether as the number of packets increases, the delay in sending and receiving the packets increases or stays broadly the same.

For the experiment, we had a single subnet with 2 AWS nodes, and one HAProxy with multicaster, which is part of the load balancer. The AWS instance type we used was m1.xlarge. We sent the packets from node 1 to node 2 via the load balancer. There was some delay in the packet transmission from the nodes to the load balancer, as well as a small delay in the multicaster. The client program on each node just forwards and receives the packets.

TABLE I. DELAY FOR PACKETS SENT IN A SIMPLE NETWORK SETUP WITH TWO NODES AND ONE MULTICASTER WITH LOAD BALANCER

Nth packet sent	Sent time delay (in microsec)	MC unit delay (in microsec)	Receive time delay (in microsec)
1000	678639	20	678620
2000	678561	23	678580
3000	678589	23	678587
4000	678587	24	678610
5000	678595	47	678580
6000	678591	23	678620
7000	678657	22	678640
8000	678663	22	678667
9000	678636	21	678650
10000	678606	21	678630

During the experiment, we sent a number of packets (1000, 2000, 3000 and so on) from one node to the other and measured the delay between the packet sending time on the first node and the packet receiving time at the other node, and time spent during context switch at the multicaster unit. Fig. 6 illustrates the test setup for 3 nodes.

The results of our experiments are displayed in table 1. We can see that there is no appreciable change in delay as the number of packets is increased. Also, we found that the system formed a cluster of n nodes successfully, hot addition and deletion was successfully performed.

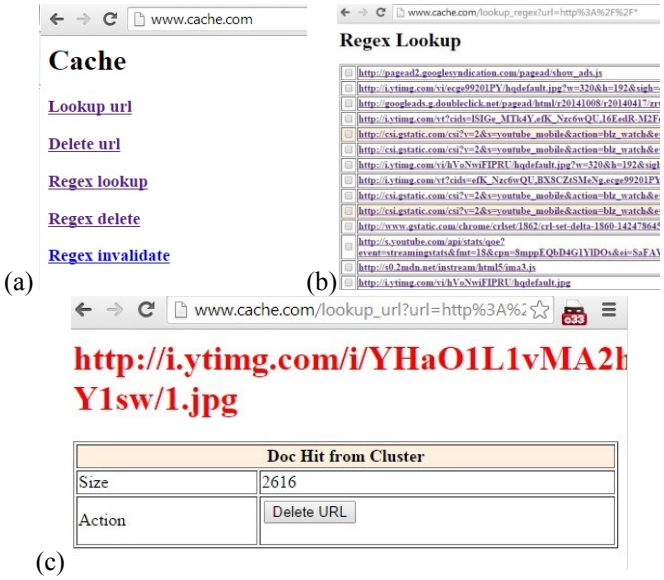

Fig. 7. Second experiment to test the effectiveness of the system to form a cluster of ATS hosts in an unsupported multicast environment. (a) shows cache inspector tool for host. (b) shows the output of a regex lookup of cache inspector tool showing cache objects. (c) Illustrates the output of the cache inspector tool upon performing the lookup operation for an URL. Initially it searches for the URL in the local cache and if the URL is not present, it searches in the clustered cache (i.e. the shared local caches of the individual machines forming the cluster). The figure illustrates a successful search of the clustered cache object, showing that the system successfully forms an ATS cluster.

We performed another experiment to study the effectiveness of our system. The setup for this experiment consisted of two or more ATS (Apache Traffic server) instances, one LoadBalancer (HAProxy), and one Multicaster Unit module which runs as part of HAProxy. The aim of this experiment was to form a cluster of hosts using our solution in unsupported multicast environment.

As mentioned previously, in this experiment we used two or more (maximum of 255) machines which got ATS-apache traffic server running on it. ATS has capability of caching images and text locally. ATS can be set as a proxy server for the client. Browsing traffic from client is routed through the ATS instance. In this test we set only one machine as the proxy IP for the client, which also acts as the load balancer. In our setup the same machine acts as the proxy as well as the load balancer in order to balance the incoming client load. HAProxy as a load balancer does health-checks and balances incoming traffic in round-robin fashion to the backend ATS instances. These ATS instances form a cluster among themselves by multicasting in the same subnet.

As mentioned earlier, the AWS environment does not support multicast subnets by default. So we use our proposed solution to achieve this, along with some necessary configuration changes in ATS. Each instance participating in cluster needs to know the load-balancer IP address. Using our solution, ATS instances form a cluster, after which they make dedicated TCP connections to each other in the cluster.

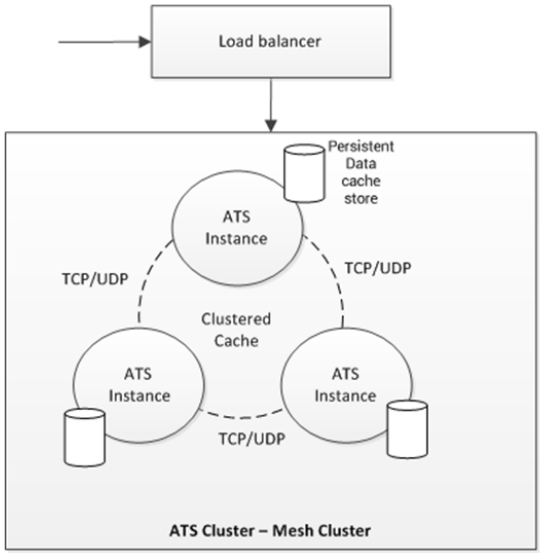

Fig. 8. Setup for the second experiment to test the effectiveness of the system to form a cluster of ATS hosts in an unsupported multicast environment.

These dedicated TCP connections are used for sharing local caches (associated with ATS) in the cluster. Instead of having standalone cache in ATS, it makes use of clustered cache among ATS instances.

When clients send HTTP requests to the proxy, which is the load balancer in our case, it routes its incoming connection to backend ATS in round-robin fashion. When a particular request/connection lands on backend ATS machine, it contacts the origin server to get the HTTP response. ATS caches the response in the cluster and simultaneously responds to the client via the load balancer.

As part of writing its response to the cache, the ATS decides uniquely (as per its algorithm) where to cache in the cluster. It is not necessary that it caches it locally on the same machine. Next time, when the same request lands on the same backend ATS or any other machine in the cluster, it performs a lookup in its local cache before checking it in the cluster cache using the same algorithm to decide where to look in the cluster uniquely. If the search does not succeed in the local and clustered cache, it gets a response from the origin server. There is a mechanism in ATS to view cached objects of the ATS in the client using preconfigured URL (in order to access the cache inspector tool) in ATS. The admin client makes use of this URL to view clustered cached objects.

Figure 7 displays the output of this experiment. It shows that a successful cluster is formed out of ATS instances, by showing that a URL lookup is made successfully from the clustered cache when it cannot find the URL in the local cache. Figure 8 shows the setup of the experiment.

VII. Conclusion and Future Work

In this paper we have analyzed the issue of load balancing in web services which do not support multicasting, using Amazon web service (AWS) as an example.

From the experiments mentioned in the previous section, we can see that conclude that our algorithm can enable a multicast network to be simulated well in different conditions in spite of the firewall restrictions.

In future, we plan to generalize this idea and extend it by implementing our solution at the kernel level.

References

[1] Amazon Web Services aws.amazon.com

[2] http://aws.amazon.com/vpc/faqs/

[3] M. Bencek, S. Buckell, S. Struk, “How to enable broadcast and multicast support on Amazon (AWS) EC2”. URL: https://www.buckhill.co.uk/blog/how-to-enable-broadcast-and-multicast-on-amazon-aws-ec2/2

[4] S. Khatua, A. Ghosh, N. Mukherjee, "Optimizing the Utilization of Virtual Resources in Cloud Environment", Proc. IEEE International Conference on Virtual Environments Human-Computer Interfaces and Measurement Systems (VECIMS), 2010

[5] Jeff Barr, "AWS Elastic Load Balancing Inside of a Virtual Private Cloud", Amazon Web Services Blog, 2011. URL: http://aws.typepad.com/aws/2011/11/new-aws-elastic-load-balancing-inside-of-a-virtual-private-cloud.html

[6] T. Chiba, M. den Burger, T. Kielmann, S. Matsuoka, "Dynamic Load-Balanced Multicast for Data-Intensive Applications on Clouds", in Proc. 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, 2010

[7] http://www.ntop.org/products/n2n/

[8] http://www.chromium.org/spdy/spdy-whitepaper