A Novel Approach to Manage Asymmetric Traffic Flows for Secure Network Proxies

Qing Li

Blue Coat Systems, Inc., 420 N. Mary Ave., Sunnyvale, CA 94085-4121, USA
Qing.Li@BlueCoat.com

Abstract. A transparent secure network proxy intercepts web traffic such as HTTP requests and applies access policies to the intercepted traffic. The proxy will reinitiate a request on behalf of the client when policies permit. Depending on policy configuration, this proxy may masquerade as the client when generating the request. The response from the server may reach the client instead of the proxy due to asymmetric routing, and if so, would be rejected by the client as an invalid response. Consequently the proxy can not complete the original request. This paper presents a new protocol and a comprehensive mechanism that facilitates the formation of a cluster comprised of multiple proxies. This proxy cluster can cover a network that spans a large geographical area, and collaboratively discover and redirect asymmetrically routed traffic flows towards the appropriate member proxy. The protocol and the algorithms presented in this paper can operate in both IPv4 and IPv6 [1] networks.[1]

1 Background and Motivation

Secure network proxies play an essential role in today's enterprise networks. These proxies can enforce access policies, conduct traffic monitoring, and perform content delivery acceleration through caching and WAN optimization. The various security requirements combined with reliability requirements present complex network architectures in which proxies cannot be deployed due to application breakage. The deploy-ability of a secure network proxy is measured by the types of policies the proxy can enforce without impeding applications. In other words, applications must continue to function even when application traffic is subject to processing at the proxy. This section provides a general introduction to the concept of a secure network proxy followed by descriptions of example problems challenging the proxy deployment.

1.1 Introduction

In this paper, a secure network proxy is an appliance that is situated between network nodes (i.e. clients) that make service and content requests, and network nodes (i.e. servers) that offer those requested services and content. Within the secure network proxy appliance, a set of application proxies operate in concert to classify and process the requests according to specific protocols. Examples of application proxies [2] are

[1] This work and its publication are sponsored by Blue Coat Systems Inc., Sunnyvale, CA, USA.

J. Cao et al. (Eds.): NPC 2008, LNCS 5245, pp. 196 – 209, 2008.

HTTP proxy, FTP proxy, Streaming Media Proxy, Peer-to-Peer proxy, SSL proxy, MAPI proxy and CIFS proxy. A secure network proxy typically operates at the application layer (layer-7) of the OSI stack [3]. In practice, since a secure network proxy is deployed at the network perimeter to manage traffic flowing between the Intranet and the Internet, a secure network proxy is commonly known as a *gateway proxy*. The most common gateway proxy is a web proxy that specializes in processing HTTP and HTTPS traffic. In this paper the terms secure network proxy, secure web proxy, web proxy and proxy are used interchangeably to simply the discussions.

A web proxy is termed a *transparent* proxy [4] if its presence is not known to the clients and servers. A web proxy is termed an *explicit* proxy when clients are configured to send all requests to that proxy instead of to the servers directly. A web proxy is said to be deployed *inline* if all traffic generated between the clients and the servers always traverse a path through the proxy.

Once a proxy intercepts a request, the access polices installed in the proxy are executed to determine what actions to apply to the request. The most common actions are *intercept* and *bypass*. Bypassed traffic will pass through the proxy without any modification. In this case the proxy acts as either a router or a bridge. Intercepted traffic is subject to further processing within the proxy by one or more application protocol proxies.

Since a web proxy operates at the application layer, the web proxy may initiate a request to the server on behalf of the client. For example, the proxy intercepts a HTTP request by first terminating the associated TCP connection [5], i.e. completing that TCP connection as if the proxy were the server. Then the subsequent HTTP request is transferred to the HTTP application proxy. If the HTTP application proxy decides to grant that request, the HTTP proxy will then initiate an actual request to the origin server on behalf of the client.

In essence, the proxy created two TCP connections for the original client request: one TCP connection is established between the client and the proxy (also known as the *client connection* or *client inbound connection*), and another TCP connection is established between the proxy and the origin server (also known as the *server connection* or *server outbound connection*). The response received on the server connection is then forwarded onto the client connection. Transformations may be applied to the response by the application proxy before it is forwarded to the client. The proxy may also decide to serve the response from its local cache instead of contacting the origin server.

1.2 Problems and Motivation

Since the server may use the client IP address as a means of authentication, the web proxy may masquerade as the client (sometimes known as *client-spoofing*) [6] by using the client's IP address as the source address of the TCP connection that the proxy initiates. In environments where asymmetric routing is a common occurrence, or due to routing policy or traffic engineering, the server response may not reach the proxy but instead reaches the client. Such environments present a challenge to the extent that deploying transparent proxies may be impossible. Consider the example given in Fig. 1.

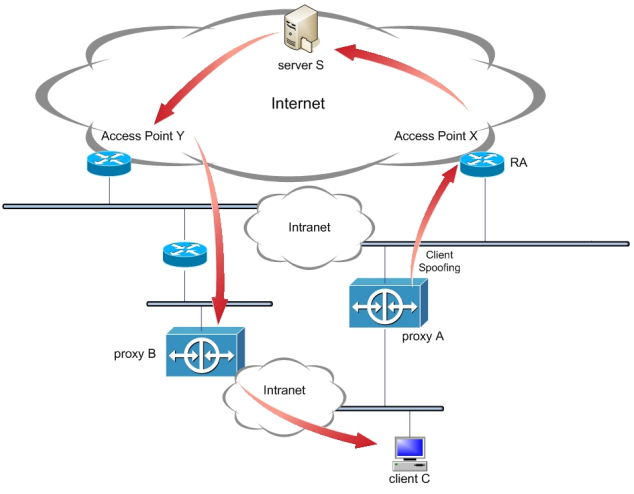

Fig. 1. This figure illustrates a typical asymmetric routing situation in an enterprise network

Fig. 1 illustrates an enterprise network where two access points are available over two different ISPs. In this example, router RA is configured as the default gateway at the client C. Proxy A is inline between C and RA. Proxy A intercepts the request from C, and then initiates another connection to the origin server S. Proxy A exercises client-spoofing because S performs IP address based authentication. This proxy A-to-S connection is through access point X. The return traffic from S to A, however, is routed through access point Y, which traverses proxy B instead of A. The main reason for this asymmetric flow is because which path the server decides to take for reaching the client's IP address (proxy A spoofed that IP address), is determined by the configuration at server S and by the routing policy defined at access point Y. Proxy B may decide to bypass this response because it has no corresponding request state. Client C, however, will reject this response because it does not have an active TCP connection that matches the response from S. More security conscious proxies may decide to reject the response outright by dropping the flow. In either case, the original request will never succeed.

Traffic load balancers and Layer-4 (L4) switches [7] are an integral part of the design in many enterprise networks. The function performed by the load balancer or the L4 switch may impose adverse effects that threaten the effectiveness of a transparent proxy. Consider the example given in Fig. 2. In order to handle a large volume of traffic, a typical design is to deploy a traffic load balancer in front of a farm of proxies that have similar capabilities and processing logics. In this figure, proxy A is called a *downstream proxy* and each proxy in the proxy farm is called an *upstream proxy* [2].

Proxies can operate in conjunction with one another in what is known as *proxy chaining* [2]. This proxy chain performs specialized functions that may require shared state. An example is dictionary compression where two proxies maintain parallel dictionaries generated from traffic passing through this proxy chain. These dictionaries allow the proxies to replace previously seen data sent between them with small tokens. Each token refers to a location in the dictionary where the previously seen data is stored.

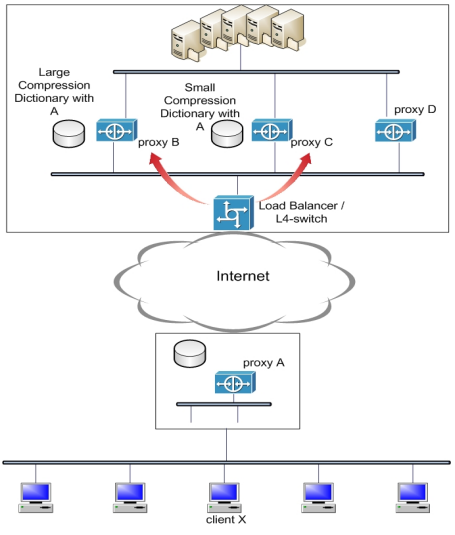

Fig. 2. Load balancers and L4 switches may break upstream and downstream proxy affinity

In this example, a downstream proxy needs to establish a chain with an upstream proxy from the proxy farm when servicing a request. It is preferable the downstream proxy establishes a chain with the same upstream proxy when the downstream proxy services other requests. The goal is to maintain an affinity in the proxy chain so to continue to build that shared state and maximize its usage.

Traffic from the downstream proxy reaches the load balancer first. The load balancer may use the client and server addresses in making its load balancing decision. If the downstream proxy is deployed transparently and exercises client spoofing, then the load balancer will see the client's IP address in the connections initiated by the downstream proxy. Subsequently the load balancer may forward the request to a different upstream proxy. On the other hand, the desired upstream proxy would be chosen correctly without client spoofing because the IP address of the downstream proxy would be visible instead.

Assume in this example that a downstream proxy creates a chain with an upstream proxy to perform dictionary based compression for traffic acceleration. The size and the lifetime of the dictionary are important factors in achieving a good compression ratio. In this example, downstream proxy A and upstream proxy B have built up a large dictionary over time. This dictionary has been consulted each time A serves requests from client X, as such, client X has been enjoying a good response time for its requests. Then at some point in time the server changed its policy to using IP address based authentication. Now proxy A must activate client spoofing resulting in the load balancer deciding to redirect traffic from proxy A to proxy C. Since proxies A and C have had limited exchange, the dictionary constructed thus far is relatively small. Client X would begin to experience a sudden drop in performance that translates into noticeable delays.

1.3 Related Work

Transparent proxies are essentially man-in-the-middle appliances that monitor, log and intercept all traffic flows that traverse through them. The concerns for privacy, copyright and content integrity [4] have only grown in recent years. In the corporate environments, individual user's desire for privacy conflicts with the corporate interests, where visibility and control are the main focuses.

Since the operational goals of the corporate lies in the opposite spectrum of those goals demanded by the users, perhaps for this reason there is a substantive lack of research activities in academia to improve the operational efficiencies and to expand the deployment coverage of these transparent proxies. Yet in the enterprise environments secure gateway proxies are essential and are confronted by numerous challenges presented by the various networks in which these proxies are deployed. Research into the vendor space yields a single vendor product that offers a partial solution, however, that partial solution in terms of algorithms is not disclosed.

The above are the main motivations for this research and the publication of this paper. The goals are to bring these issues to light, and to offer a detailed disclosure on a solution that is fully deployed in production environments in hope to solicit peer reviews, and to invigorate additional research activities in this area.

2 System Architecture

To solve the aforementioned problems, this section presents the details of a protocol that facilitates the formation of a cluster. This cluster is comprised of multiple member proxies that can be multiple hops away from each other. Each member proxy exchanges information about the requests that it has processed with every other member proxy. The clustering protocol is designed to manage the cluster membership as well as to exchange states of processed requests.

2.1 Overview

In the problem illustrated in Fig. 1, if proxy B has the ability to recognize traffic flows that belong to proxy A and can forward to A those traffic flows, then these transparent proxies can be deployed throughout an autonomous system without concerns for asymmetric routing or the presence of load balancers. Proxy A must inform proxy B about the connections that A has processed. Similarly proxy B must inform proxy A of those connections that belong to B. The solution presented in this paper can handle the situation where asymmetrically routed traffic reaches a member proxy before the proxy-to-proxy notification arrives at that member proxy.

The solution is to create a *proxy cluster* that includes A and B and let A and B form a peering relationship. Proxy A and B can signal one another about their respective workload after successfully establishing the peering relationship. Proxies are typically deployed in different subnetworks and these proxies are multiple hops away from each other. The notifications must be delivered reliably. For these reasons proxy peers within the cluster are connected over a full-mesh topology. The connection between any pair of peers is a TCP connection. Fig. 3 visualizes the secure proxy cluster.

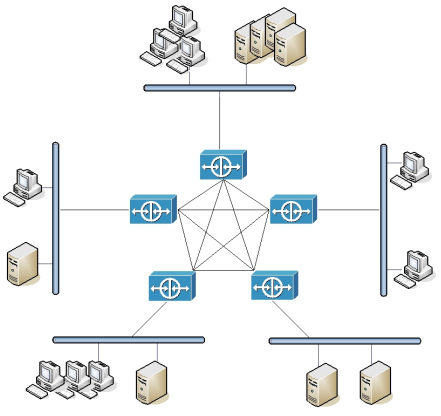

Fig. 3. Member proxies form a cluster with full-mesh TCP connectivities

Section 3 provides additional information on the reasons why the solution is not based on a multicast protocol.

The solution is comprised of the following three main modules:

- cluster control-channel management module
- connection management module
- traffic flow forwarding module

The cluster control-channel management module is responsible for establishing and maintaining peer relationships with every other member in the cluster.

The connection management module is responsible for maintaining the database that describes connections that are processed by local proxy as well as connections that are processed in peer proxies.

The traffic flow forwarding module is responsible for classifying packets against this connection database and forwarding the packets to either local on-box application proxies, or redirecting these packets to peer proxies.

2.2 Cluster Control-Channel Management

In the first implementation of this solution, the cluster membership is statically configured into each proxy. A proxy attempts to establish a TCP connection to every other peer in the membership list. The membership list may be static, but the proxies identified in the list may join and leave the cluster dynamically. A proxy may leave the cluster either voluntarily or involuntarily due to hardware or software failure.

This special peer-to-peer TCP connection is called a *cluster control-channel*, or simply the *control-channel*. A specially chosen TCP port is reserved by the implementation as the service port on which peering requests are made. If successful, the two peers begin to synchronize their respective *connection tables* over the control-channel. In the context of this paper, the connection table includes all of the TCP and UDP connections associated with the requests that have been processed by a proxy.

Note for intercepted requests, the connection table contains both client bound and server bound connections.

The *cluster connection database*, or simply *connection database* is defined as the connection table that contains all of the intercepted and bypassed connections that have been collected from all of the active peers of the cluster. The cluster connection database is described in more detail in Section 2.4.

The types of messages that are exchanged over the control-channel are given in Table 1. The INSTALL_CONNECTION message informs the receiver about those connections that are handled by the sender. The receiver will store these connections in its connection database so that it can recognize packets that are associated with those connections, and forward those packets back to the sender. The REMOVE_CONNECTION message informs the receiver about those connections that have been completed, or those connections that should no longer be bypassed (perhaps due to changes made in the access policy). The receiver will remove those connections from its connection database.

Table 1. Cluster Management Control Connection Message Types

Message Type	Action	Description
INSTALL_CONNECTION	FORWARD	This message installs state about a connection into a peer. Traffic belonging to this connection will be forwarded to the originator of this message.
	BYPASS	This message installs state about a bypassed connection into a peer.
REMOVE_CONNECTION	FORWARD	This message removes state about a forwarded connection from a peer.
	BYPASS	This message removes state about a bypassed connection from a peer.
CLUSTER_PEER_SYNC	PEER_JOIN	This message indicates the sender is in the process of establishing a peering relation with the receiver. This message provides a safeguard against accidental connection being made to the clustering port. This message is also an indication the sender requests a peer to send over its connection table that includes both intercepted and bypassed connections.
	PEER_KEEPALIVE	This message informs a peer the sender is alive.
	PEER_LEAVE	This message informs peers that the sender of this message is exiting the cluster and is no longer an active member of the cluster.
	PEER_REJECT	This message informs a peer that the peering request initiated by the receiver has been denied. This reason for the denial accompanies the PEER_REJECT message.

2.3 Cluster Control-Channel Packet Formats

The control-channel packets have a generic four-byte header, which is shown in Fig. 4. The first two bytes are common, and the content of the subsequent two bytes depends entirely on the message type.

Each packet begins with a *Message Type*. The defined message types are INSTALL_CONNECTION, REMOVE_CONNECTION, and CLUSTER_PEER_SYNC. The *Action Type* can be one of FORWARD or BYPASS. The *Protocol Family* can be either TCP or UDP, which identifies subsequent connection information as to contain either TCP or UDP connections. The *Address Family* specifies whether the IP addresses of a given connection are IPv4 or IPv6 addresses.

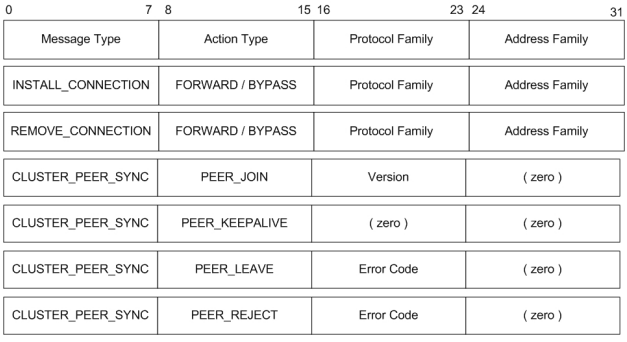

Fig. 4. Control-Channel packet header format

A proxy that intercepts a client request will issue two FORWARD actions to all of its peers. The first FORWARD action installs the client inbound connection and the second FORWARD action installs the server outbound connection in the peers. All traffic that belongs to these two connections which arrives at any other peer will be forwarded to the intercepting proxy.

The BYPASS action is necessary when one proxy receives a client request and the policy indicates the connection is to be bypassed. In this case, instead of redirecting the return traffic that was asymmetrically routed to the intercepting proxy, the intercepting proxy can ask its cluster peers to bypass the traffic, thus forwarding the traffic toward the requesting client or server directly. This direct bypass approach can be considered as an optimization.

The PEER_JOIN action serves as a safeguard against accidental connection made to the special TCP port reserved for cluster peering requests. Once the TCP connection is successful, the peering initiator must issue the PEER_JOIN action as the first exchange. In return, the peer will respond with a PEER_JOIN message. The peer that is waiting for the peering requests on the special socket [8] will close the control connection if the PEER_JOIN action is not the first exchange. In other words, for both ends of the control-channel, the first bytes exchanged must constitute a PEER_JOIN action and its associated data. The number of connect attempts to be made for establishing peering relation to another cluster member before abandoning that peer is configurable by the administrator. Infinite retries is a configurable option to manage the situation where the peer is down for an extended period of time but will eventually recover.

The packet formats may change depending on the *Version* field. Therefore, at the time of establishing the peering relation the version field must be examined by each peer to determine if the peers are compatible. Incompatible peers will result in the failure of peering establishment. The version field is carried in the PEER_JOIN message only.

The PEER_KEEPALIVE action is necessary when a configured period of time has elapsed and there is no traffic on a control-channel. The PEER_KEEPALIVE is sent to each peer to inform those peers about the liveliness of the sender. A response is not necessary because the control channel operates over TCP, which is reliable. Once a proxy deems its peer unreachable, that proxy will remove all connections (whether

bypassed or intercepted) associated with the unreachable peer from its cluster connection database.

A member proxy that voluntarily leaves the cluster issues the PEER_LEAVE action to all other members. The PEER_LEAVE action is a proactive way of informing other peers to immediately remove all connections associated with the sender of the PEER_LEAVE action from their connection databases.

A member proxy transmits the PEER_REJECT action to inform another proxy that its peering request has been denied, for example, because the other proxy is not present in the cluster membership list. The reason for the rejection is included in the PEER_REJECT message.

For the PEER_REJECT message, the *Error Code* field indicates reason for peering failure. Peering failure can occur when a peer is not in the cluster membership list but attempts to connect to other members that are in the list. Another reason for peering rejection is version number mismatch.

After a proxy has established the control channel with another proxy, if an unrecognized message is received on their control-channel, that control-channel will be closed. The main reason is at the point where the bad message is read, the proxy can no longer parse the remaining bytes received on that channel. In this case the proxy will issue a PEER_LEAVE action to its peer with an error code that indicates bad message.

Fig. 5 illustrates an example of a control-channel packet carrying IPv4 connection information, and this connection would be forwarded by the peers. The size of the source and destination ports is fixed. Each INSTALL_CONNECTION and REMOVE_CONNECTION will carry only one set of connection information. In other words, if there are two blocks of connection information to be installed, then there will be two INSTALL_CONNECTION messages.

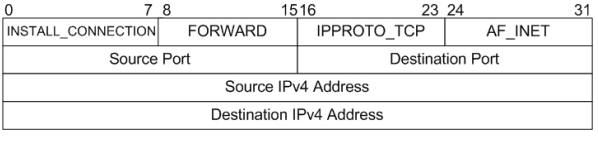

Fig. 5. This is an example of an IPv4 connection installation packet

As can be seen from Fig. 5, the size of the packet can be derived from the packet header. For example, if the message type is INSTALL_CONNECTION, the protocol is TCP, and the address family is IPv4, then the packet is 16 bytes long. On the other hand, if the address family is IPv6, then the packet is 40 bytes in size (4-byte header, 4 bytes for ports, 16-byte source IPv6 address, and 16-byte destination IPv6 address).

Fig. 6 illustrates the possible usage of these message types. The INIT state is the state where a proxy reads and parses the cluster membership list, and determines which peers it should initiate an active peering request to. In the CONNECT state, the peering request initiator responds with a PEER_JOIN and enters the SYNCHRONIZE state. In the SYNCHRONIZE state, the two peers exchange their respective connection tables. The SYNCHRONIZE state is the same as the ESTABLISHED state if the initiator has an empty connection table.

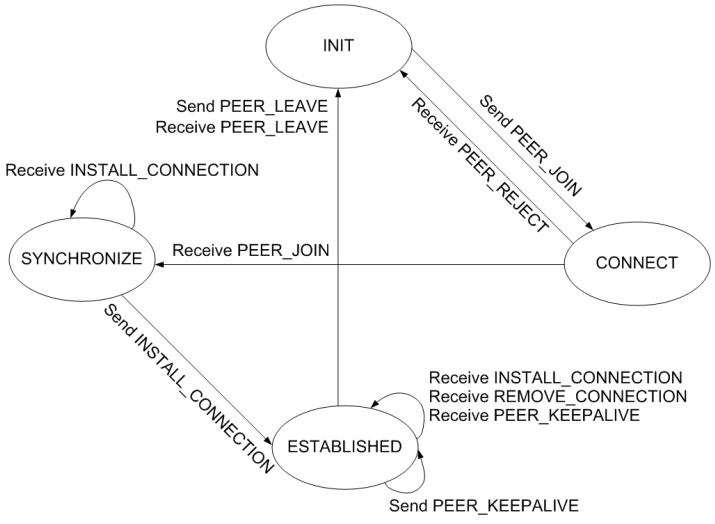

Fig. 6. Example of peering states and associated message exchanges

2.4 Cluster Connection Database Management

At the completion of the cluster establishment, all of the proxies belonging to the same cluster must have the same connection database of all the connections that are intercepted and bypassed by every member proxy. The most efficient way to synchronize the connection table is to exchange local connection table with each peer at the peering time. Once the cluster is fully established, each proxy will maintain a cluster connection database covering all of the active peers. New members can join the cluster and the peer table will be updated accordingly.

Each time a new request is processed by a proxy, that proxy sends the INSTALL_CONNECTION notification to all its peers before it initiates a connection to the server for that request. This approach will reduce possible memory overhead that may be incurred by a peer if the server response reaches a peer before the notification arrives at that peer.

Each proxy unilaterally sends out PEER_KEEPALIVE messages on its own timer, and the other side of the control channel treats this peer as unresponsive if that side does not hear any traffic after some fixed time interval, for example, 3 times the PEER_KEEPALIVE interval. The connections associated with a peer are removed from the local connection database if the peer becomes non-responsive. The number of probes to send and the probe interval are configurable by the administrator on a per peer basis. A proxy will reset the PEER_KEEPALIVE transmission timer each time a message of INSTALL_CONNECTION or REMOVE_CONNECTION type is received. The periodic timer is also reset when a PEER_KEEPALIVE is triggered.

Table 2 illustrates the structure of the connection table at the intercepting proxy, and the corresponding connection database maintained by the peers.

Table 2. Connection Table and Connection Database

Connection Table of the Intercepting Proxy A

Connection	Direction	Responsible Proxy
< src-A, src-PA, dst-B, dst-PB >	Client to Proxy	Proxy A
< src-C, src-PC, dst-D, dst-PD >	Proxy to Server	Proxy A

Connection Database at Proxy B

Connection Information <source-IP, source-Port, destination-IP, destination-Port>	Protocol	Action	Originating Proxy
< src-A, src-PA, dst-B, dst-PB >	TCP	Forward	IP address X
< src-C, src-PC, dst-D, dst-PD >	TCP	Bypass	IP address X
< src-E, src-PE, dst-F, dst-PF >	UDP	Bypass	IP address Y

2.5 Traffic Flow Forwarding Module

Once the connection database is synchronized among all peers, the traffic flow forwarding module within a proxy can begin processing traffic flows that belong to other member proxies.

The traffic flow module must first act as a packet classifier, i.e. the traffic flow module must examine all traffic received by the proxy and match this traffic against the local connection database. Conceptually the connection database is comprised of all connections that have been processed by all member proxies. In the actual implementation, however, connections local to a proxy are not inserted to that database but are kept in a separate table for performance reasons.

For an input packet that matches a connection in the connection database, i.e. for a packet that belongs to a traffic flow processed by another proxy, the traffic flow module will package this off-box packet and transmit it to the proxy identified by the connection database entry (i.e., to the IP address specified in the connection entry). The off-box packet is encapsulated inside an IP-in-IP frame [9]. The IP-in-IP frame has the address of the local proxy as the source address, and the address of the remote proxy as the destination address. The choice of using IP-in-IP encapsulation for transporting off-box traffic is to take advantage of the existing routing infrastructure, and for ease of implementation.

3 Discussion

The clustering protocol could be built on top of a multicast mechanism because multicasting connection state information to all other members simultaneously is more scalable. IP multicast forwarding is not a mandatory requirement and is by default disabled in the majority of the enterprise routers. The connection state information must be exchanged reliably and in bounded time among the peers. There exists reliable multicast protocols, but those solutions are typically too complex to implement. The solution presented in this paper assumes the number of peers would not exceed 20 proxies for all practical purposes. A reliable multicast clustering protocol would be desirable if the number of peers exceed that limit.

When a proxy first encounters a partial flow, e.g. the proxy receives a TCP <SYN,ACK> packet, or when the proxy receives a TCP packet with only the <ACK> bit set, and the proxy cannot find any connection that matches this packet in either the local connection table or the overall connection database. In this situation, the proxy cannot discard any packet belonging to the partial flow. The reason being another member proxy may have processed that flow, and the notification about that flow is in transit and has not reached the local proxy. The longer a proxy can hold these partial packets, the longer the distance is tolerated between a pair of proxies. In other words, the packet hold time determines the diameter of the cluster. Once the hold time expires, the queued packets are subject to re-classification and can be dropped if a matching connection is still missing.

The solution proposed in this paper can be utilized to build intelligent load balancers. Intelligence comes from the fact that offloading decisions (i.e. connection forwarding) can be made by an external module operating at a layer higher than layer-4. Refer to Fig. 2, consider the scenario where a TCP connection request (e.g. as a result of a HTTP request) is received at a member proxy. This TCP <SYN> packet can be handed to the WAN optimization module. The WAN optimization module examines the client and server addresses, and determines another proxy is best at handling this request because that other proxy has built a better compression dictionary. In this case, the WAN optimization module would instruct the connection management module to install this connection as an off box connection, and then subsequently forward this connection to the chosen member proxy. Such a method is sometimes referred to as *connection handoff*.

The traditional load balancer or L4 switches operate at layer-3 and layer-4 of the OSI stack. The load balancer is typically a single point of failure, and the traffic flow is load balanced in one direction. The full-mesh cluster topology as shown in Fig. 3 enables bidirectional load balancing and provides reliability in the overall solution because if one member proxy is inoperable, only a subnetwork that is covered by that failed proxy would be affected. Traffic is not affected by the failure if the network has built-in redundancy and can re-route the traffic around the failed proxy to another member proxy.

Since the connection database contains all of the requests processed by every member proxy, and because each entry in the database identifies a responsible proxy, an administrator can access any member proxy to gain a complete view of all proxied traffic flowing through the entire network. In other words, the solution proposed can serve as an excellent network troubleshooting tool.

Another observation is each member proxy can enforce both the access policies defined in it, and at the same time enforce policies defined in the other member proxies as well. This solution enables a large amount of security and access policies to be divided into smaller subsets and install each subset into one member proxy to enforce. In other words, this solution allows for the implementation of a distributed policy enforcement mechanism.

In the enterprise environment where the clustering solution is deployed, and accordingly to the appliance capability specification, there are approximately 20,000 simultaneous requests active within the appliance at any given time. At a minimum, each connection state holds the protocol type, the address family, connection 4-tuple <source address, source port, destination address, destination port>, the IP address of

the responsible proxy and the action type. For IPv4 each connection state requires roughly 32 bytes, and with a cluster containing 20 member proxies, the size of the connection database is approximately 25.6 MB.

In the same deployment environment, each proxy is subject to approximately 2000 requests per second. Assuming these requests are HTTP requests, each requested object is typically 10K bytes in size, which translates into 20MB per second per link traffic. In other words, the proxy receives 160Mbps from the server and then transfers this same amount of traffic to the client. In the worst situation where all traffic destined to one proxy must always traverse another member proxy, and assume there is a 100ms delay between the two peers, then that other member proxy must buffer 2MB of packets on behalf of its peer.

At 2000 requests per second, approximately 390Mbps is exchanged within the cluster for installing or removing connection states. Each proxy handles roughly 19.5Mbps of cluster protocol exchange in the worst case and with 20 proxies in the cluster.

The solution presented in this paper has been implemented in a system with 4 CPUs, 4GB RAM and multiple Gigabit Ethernet interfaces, which represents a typical high-end appliance. The system memory and network bandwidth requirements in the worst case scenario are easily satisfied.

4 Conclusion and Future Work

In this paper I have described a protocol and system that enable a group of proxies to form a proxy cluster. This cluster can cover a large scale network that can span geographical locations. This proxy cluster acts as a single virtual proxy that can enforce a large distributed set of policies. Without the proposed solution, transparent proxies that perform client spoofing cannot be deployed in environments where asymmetric routing takes place. The solution presented in this paper also enables the construction of better load balancers and application layer switches.

The solution proposed in this paper has been deployed in real-world production environments. Performance measurements were conducted using commercial web performance testing tools against the requirements specification. A formal system performance analysis is in progress.

Acknowledgements. I would like to thank Blue Coat Systems for sponsoring this research work and granting me the permissions for publication. I would like to thank the various reviewers from Blue Coat Systems for their comments and suggestions, in particular Min Hao (Howard) Chen and Yusheng Huang. I would like to thank Ron Frederick for his review and his technical contribution.

References

1. Li, Q., Jinmei, T., Shima, K.: IPv6 Core Protocols Implementation. Morgan Kaufmann, San Francisco (2006)
2. Gourley, D., Totty, B., Sayer, M., Reddy, S., Aggarwal, A.: HTTP The Definitive Guide. O'Reilly, Sebastopol (2002)

3. Tanenbaum, A.S.: Computer Networks, 4th edn. Prentice Hall PTR, Upper Saddle River (2002)
4. Wessels, D.: Web Caching. O'Reilly, Sebastopol (2001)
5. Stevens, W., Wright, G.: TCP/IP Illustrated. The Implementation, vol. 2. Addison-Wesley, Upper Saddle River (1994)
6. Wikipedia, http://en.wikipedia.org/wiki/IP_address_spoofing
7. Syme, M., Goldie, P.: Optimizing Network Performance with Content Switching: Server, Firewall, and Cache Load Balancing. Prentice Hall PTR, Upper Saddle River (2003)
8. Stevens, W.: Unix Network Programming. The Sockets Networking API, 3rd edn., vol. 1. Addison-Wesley, Upper Saddle River (2003)
9. Simpson, W.: RFC 1853, IP in IP Tunneling, IETF (1995)