EtherProxy: Scaling Ethernet By Suppressing Broadcast Traffic

Khaled Elmeleegy
Yahoo! Research

Alan L. Cox
Rice University

***Abstract*—**
Ethernet is the dominant technology for local area networks. This is mainly because of its autoconfiguration capability and its cost effectiveness. Unfortunately, a single Ethernet network can not scale to span a large enterprise network. A main reason for this is broadcast traffic resulting from many protocols running on top of Ethernet. This paper addresses Ethernet's scalability limits due to broadcast traffic.

We studied and characterized broadcast traffic in Ethernet networks using traces collected from real networks. We found that broadcast is mainly used in Ethernet for service and resource discovery. For example, the Address Resolution Protocol (ARP) uses broadcast to discover a MAC address that corresponds to an IP address.

To avoid broadcast for service and resource discovery, we propose a new device, the EtherProxy. An EtherProxy uses caching to suppress broadcast traffic. EtherProxy is backward compatible and requires no changes to existing hardware, software, or protocols. Moreover, it requires no configuration. In our evaluation, we used real and synthetic workloads. Using both workloads, we experimentally demonstrate the effectiveness of the EtherProxy.

I. Introduction

Ethernet is the dominant layer 2 networking technology in many environments including enterprise networks, data centers, and campus networks. Virtually every computer system nowadays has an Ethernet network interface card. Ethernet dominance for the most part, is due to its ease of use and its high performance-to-cost ratio. An Ethernet network is easy to use because equipment can be added to it with little or no manual configuration.

In Ethernet networks, many higher level protocols rely on broadcast traffic. This is mainly used for service and resource discovery. For example, ARP [17] uses broadcast to discover a MAC address that corresponds to an IP address. A Dynamic Host Configuration Protocol (DHCP) [4] client also relies on broadcast to find the DHCP server.

Broadcast traffic poses a scalability problem to Ethernet networks. This traffic increases in large networks having a lot of hosts. Hence, it can reach a point where broadcast traffic consumes a significant part of the network bandwidth. Moreover, the CPUs of end hosts can get overloaded by processing broadcast traffic received through the network.

To handle these problems, a large network is partitioned into smaller Ethernet segments. These segments are connected together using layer 3 routers. This is because a router does not forward broadcast or multicast traffic. Hence, an Ethernet segment is called a broadcast domain. To limit the amount of broadcast traffic in a broadcast domain, the number of hosts in the domain needs to be limited. Table I shows Cisco's recommendation for the number of hosts in a broadcast domain based on the protocols used in the network [13]. Those values are very modest, because of the scalability problem due to broadcast traffic.

An Ethernet network can be segmented physically into separate physical networks, or logically using Virtual Local Area Networks(VLANs). Using VLANs offers more flexibility over physical segmentation of the network. This is because in the VLANs case, segmentation is logical and thus configurable. This is useful, for example, if a user, connected to a segment S, wants to change her location while remaining connected to S. In this case, a network port at the new location can be configured to belong to S, allowing the user to move while remaining connected to S.

Although segmenting Ethernet networks alleviates the scalability problem, it does not come without a cost. In a large network, this would require plenty of error-prone manual labor. First, if VLANs are used, they must be created and configured. Second, subnets need to be created too. Third, address assignment needs to be managed. Fourth, routers connecting network segments need to be configured. Moreover, this approach is costlier than using Ethernet alone, as layer 3 routers are significantly more expensive than layer 2 Ethernet switches. Their high cost is due to their complexity compared to layer 2 switches, e.g. they may be required to handle different protocols like IP, IPX, and AppleTalk.

In addition to all these costs, segmenting the network imposes significant limitations. For example, in data centers Virtual Machines (VMs) migrate from one physical machine to another for load balancing purposes. Migration needs to be transparent to application running on top of these VMs. This becomes difficult if a VM needs to migrate from one subnet to another. This is because the VM needs to maintain its IP address during its runtime, which is difficult to do while crossing subnets. Another limitation of Ethernet segmentation is that there are protocols not routable at the network level, e.g. the Network Basic Input/Output System (NetBIOS) [12] and the Maintenance Operations Protocol (MOP) [3].

	IP	Netware	AppleTalk	NetBIOS	Mixed
Max # hosts	500	300	200	200	200

TABLE I
RECOMMENDED MAXIMUM SIZE OF A BROADCAST DOMAIN BY CISCO BASED ON THE PROTOCOLS RUNNING IN THE BROADCAST DOMAIN.

From all this, we argue that it is desirable to build large LANs by exclusively relying on layer 2 technology (e.g. Ethernet) rather than using layer 3 (e.g. IP routing).

In previous work, to scale Ethernet, people have proposed rebuilding it from the ground up [11], [8]. Some of this work have promised end host backward compatibility, e.g. SEATTLE [8]. Even with end host backward compatibility, these new technologies are not incrementally deployable into existing Ethernet networks. Moreover, Ethernet equipment will be more cost effective than the new equipment needed for the new technologies as Ethernet equipment is mass produced. Hence, even if one is to build a new enterprise network, using Ethernet will likely be more economical than using a new technology. Finally, there is a lot of human expertise in installing and operating Ethernet networks, which is not the case for other technologies.

In contrast, we take a fully backward compatible approach to address Ethernet's scalability problem. We propose a new device, the EtherProxy[1], that can be inserted into an existing Ethernet to suppress broadcast traffic. For protocols that use broadcast, an EtherProxy caches protocol information carried by protocol messages passing through it. Then for each of those protocols, the EtherProxy employs an algorithm that may suppress subsequent broadcast protocol messages with the aid of the cached protocol information. For example, an EtherProxy can cache the ARP response to an ARP request. Subsequent ARP requests for the same IP address can be served directly from the EtherProxy's cache rather than broadcasting the request over the network. EtherProxy is backward compatible and requires no changes to existing hardware, software, or protocols. Moreover, it requires no configuration. This allows a single Ethernet network, with a single broadcast domain, to scale to much larger sizes than before. Consequently, a large network can be built exclusively using layer 2 technology, i.e. Ethernet.

The rest of this paper is organized as follows. The next section describes EtherProxy's design and operation. Section III presents a study of broadcast traffic in Ethernet networks along with an evaluation of the EtherProxy's effectiveness. Section IV discusses other scalability concerns in Ethernet networks. Section V discusses related work. Finally, Section VI states our conclusions.

II. THE DESIGN OF THE ETHERPROXY

The EtherProxy intercepts broadcast packets it recognizes then it either (1) responds to a host's query, sent via a broadcast packet, itself using its cached information and without forwarding the request packet to the network, (2) replaces the broadcast destination address in the packet with the appropriate unicast destination address, cached at the EtherProxy, and then sends the packet over the network, or (3) passes on the broadcast packet as is, if it contains information that needs to be disseminated to all hosts in the network. If the broadcast packet's protocol is not recognized and there is no rule for handling it, then the packet is just passed through.

[1]Not to be confused with Proxy ARP [18]

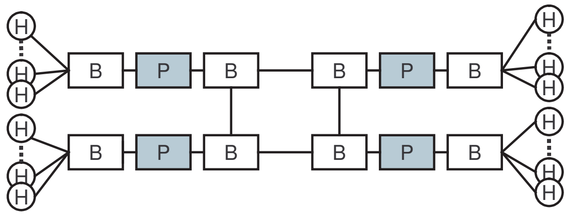


Fig. 1. An example deployment of EtherProxies in an example topology.

Every protocol using broadcast packets can be handled by one of the approaches mentioned above. For example, an EtherProxy can cache responses to ARP requests. Then, using its cached responses, it can directly respond to requests coming from hosts, instead of forwarding the request to the network. For protocols like DHCP, an EtherProxy can replace the destination address of broadcast packets with the appropriate unicast destination address it has cached. Conversely for protocols like the Network Time Protocol (NTP) [10], where NTP servers periodically broadcast time information to synchronize clocks of all hosts in the network, the EtherProxy does not intervene and just passes those packets through.

To be able to suppress broadcast traffic in Ethernet networks, EtherProxies should be placed along the path of this traffic to be able to detect and handle it. It is preferable to deploy EtherProxies close to the edges of the network to quickly intercept and suppress broadcast traffic before it floods the network. Figure 1 shows an example deployment of EtherProxies, where Bs are bridges, Hs are end hosts, and Ps are EtherProxies.

A. *Modules in the EtherProxy*

For every protocol the EtherProxy handles, it uses a specific software module that handles packets of this particular protocol. More modules can be loaded at runtime into the EtherProxy to allow it to handle more protocols. For every packet arriving at the EtherProxy, the packet is inspected to detect the protocol it belongs to. Accordingly, the packet is forwarded to the corresponding module. The EtherProxy also informs the module with the port that this packet arrived at. If no module is found to handle the packet's protocol, the packet is allowed to pass through the EtherProxy without further processing. In this section we will present the design of two modules that are representative of modules handling different protocols and services using broadcast packets.

1) The ARP Module: This module is responsible for handling ARP requests and responses. In this module, the EtherProxy maintains an ARP cache of MAC-to-IP address mappings. This cache can be viewed as an aggregate of every local ARP cache of every host behind the EtherProxy, maintained by hosts' operating systems. When the EtherProxy receives an ARP request from a host A querying about the MAC address of another host with an IP address X, the EtherProxy checks its ARP cache. If a mapping is found, then a response is

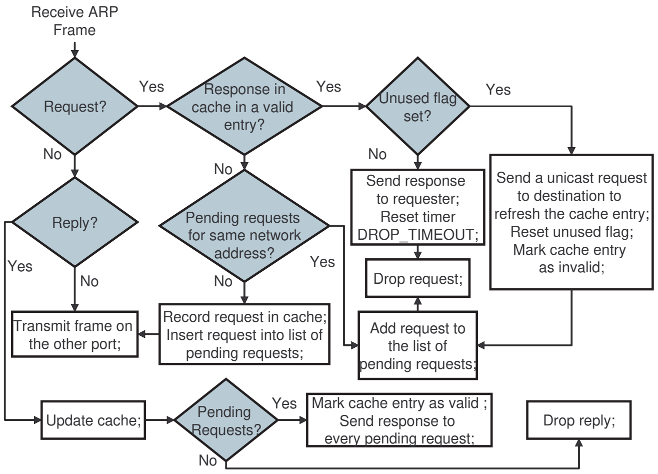


Fig. 2. A flowchart of handling ARP packets in the ARP module.

constructed and directly sent back to A without having the request flooded into the network. Otherwise, the request is broadcast through the network and also is recorded in the EtherProxy's ARP cache as an *incomplete* cache entry. If more ARP requests arrive for the address X from other hosts before the EtherProxy has received a response from the host with the address X, the EtherProxy suppresses those subsequent requests and maintains a list of the hosts issuing those requests. When the request sent by the EtherProxy reaches the host with the address X, it will respond. The EtherProxy will intercept the ARP response, the same way it intercepted the ARP request. Consequently, the EtherProxy includes the response in its ARP cache and thus the complete mapping is recorded in the cache. Then, the EtherProxy constructs and sends responses to all hosts with pending requests for the address X. If the EtherProxy does not receive a response from the host with the address X within a timeout, it can choose to retry sending the host another request, but eventually it would give up dropping this incomplete cache entry.

Figure 2 shows a flow chart of how a received ARP packet would be handled by the ARP module. In the general case, the EtherProxy can receive ARP requests and responses from either upstream or downstream. Requests from downstream are only served by information received from upstream and vice versa. Thus, for every entry in the ARP cache, the EtherProxy needs to record the port the ARP reply arrived at. Those details are omitted from the flow chart for simplicity.

Mappings in the cache expire after reaching a timeout, *REFRESH_TIMEOUT*. This is to handle the case when a mapping for a host becomes invalid because, e.g., the host becomes unavailable. This is analogous to the local ARP cache at hosts maintained by their operating systems. Thus, the EtherProxy does not change the semantics of ARP. To try to prevent those mappings from expiring at the EtherProxy cache, the EtherProxy sends a unicast ARP request to every host corresponding to every mapping that is about to expire. If the host responds, the mapping is then given a new timeout and is allowed to remain in the cache. Otherwise, the mapping will expire and will be dropped from the cache.

It is guaranteed that invalid mappings will be flushed out of the network and never persist. Using the REFRESH_TIMEOUT will prevent stale entries from persisting in EtherProxies' caches. Moreover, stale information can not circle in the network. This is because the Ethernet's forwarding topology is loop free. In addition, ARP information only flows in one direction. This is because an EtherProxy only serves an upstream ARP request from information obtained from downstream and vice versa.

Although unicast ARP requests, used to maintain the ARP cache, are sent at a very low rate, they can still add noticeable extra traffic in the network. This will be especially true in a large network with a large number of hosts and a large number of EtherProxies each maintaining cache entries for a lot of hosts in the network. Thus, it is preferable to eliminate unicast ARP requests used to maintain cache entries that are no longer being used. To detect unused entries, an EtherProxy uses another per ARP cache entry timeout, *DROP_TIMEOUT*, that is reset whenever the ARP cache entry is used to serve an ARP request. This DROP_TIMEOUT can have a very large value, i.e. measured in hours. When this timeout expires, the cache entry is marked as *unused* and no more unicast ARP requests are used to maintain it. Later, if an ARP request arrives from a host requesting the mapping at this unused entry, the EtherProxy sends a unicast ARP request to refresh the entry. If no response is received for this unicast ARP request within a timeout, the EtherProxy sends a broadcast ARP request to reconstruct the cache entry.

Another optimization to reduce unicast ARP requests used to maintain the ARP cache at the EtherProxy is to allow cache entries to last longer in the cache without the REFRESH_TIMEOUT expiring. To do so, the EtherProxy can use the data packets arriving from a host as an implicit indication that this host is alive and having the same MAC and IP addresses cached in the EtherProxy's ARP cache. Consequently, the EtherProxy can reset the REFRESH_TIMEOUT timer in its ARP cache for each cache entry corresponding to a host it receives a data packet from. Thus, cache entries will last longer in the cache without the need of sending explicit unicast ARP requests. The downside of this approach is that the EtherProxy now needs to inspect every data packet that flows through it.

Likely, the ARP cache will be able to accommodate all IP addresses in the network. Thus, no cache replacement would be required, as we will see in Section II-C. However, if the cache can not accommodate all addresses, an LRU replacement policy can be employed among all cache entries.

2) The DHCP Module: DHCP allows a server to dynamically distribute IP addressing and configuration information to clients on a TCP/IP network. This simplifies network administration because the software keeps track of IP addresses rather than requiring an administrator to manage the task.

In DHCP, when a client is initialized, it goes through the following conversation with the DHCP server to receive its configuration information. First, the client sends a DHCPDISCOVER message with a broadcast destination address to discover a DHCP server. The DHCPDISCOVER message includes the client's MAC address. Second, the server responds with a DHCPOFFER message to offer an IP address along with configuration information to the client. Since the client does not yet have an IP address, the DHCPOFFER message is addressed to a broadcast IP address, which is mapped to a broadcast MAC address. The DHCPOFFER message includes the client's MAC address, received in the DHCPDISCOVER message, along with the DHCP server's IP address. Third, the client responds with a DHCPREQUEST message indicating that it wants to accept the offered IP address. The DHCPREQUEST message is also sent to a broadcast address. This is because there could be more than one DHCP server in the network and it is possible that more than one offered IP addresses to the client. When the other DHCP servers receive the DHCPREQUEST message, they know they can now release their offered IP addresses. Similarly, the DHCPREQUEST message includes the client's MAC address, along with the DHCP server's IP address received in the DHCPOFFER message. Finally, the DHCP server responds to the DHCPREQUEST with either a DHCPACK or a DHCPNACK if some error occurs, thus completing the initialization cycle. The DHCPACK/DHCPNACK message still has a broadcast address as the client has not acquired an address yet. Again, the DHCPACK/DHCPNACK message includes the client's MAC address, received in the DHCPREQUEST message, along with the DHCP server's IP address.

In the DHCP module, the EtherProxy intercepts the DHCP messages. It maintains a list of both the MAC and IP addresses of the DHCP servers in the network. When the EtherProxy receives a broadcast DHCPDISCOVER message from a client, it picks one of the DHCP servers in its list and forwards the DHCPDISCOVER message to this server as a unicast packet. For load balancing, the EtherProxy selects DHCP servers from its list of servers in a round robin fashion. When the message reaches the DHCP server, it replies with a broadcast DHCPOFFER message. The EtherProxy intercepts the DHCPOFFER message and replaces its broadcast address with the client's unicast address and then sends the message to the client. The client address is found in the DHCPOFFER message itself. The client then responds with a broadcast DHCPREQUEST message. The DHCPREQUEST message includes, in its body, the IP address of the DCHP server that has sent the DHCPOFFER message. Given the server's IP address, the EtherProxy can get its MAC address as the EtherProxy maintains these mappings. The EtherProxy intercepts this message and replaces the broadcast destination address with the DHCP server MAC address. Finally, the server responds with either a broadcast DHCPACK, or a broadcast DHCPNACK message that includes in its body the IP address offered to the client along with the client's MAC address. This message is intercepted by the EtherProxy, which replaces its broadcast address with the client's MAC address.

Initially, for the EtherProxy to construct the list of the DHCP servers in the network, it broadcasts the first DHCPDISCOVER message it receives from a DHCP client. This makes all the DHCP servers in the network respond

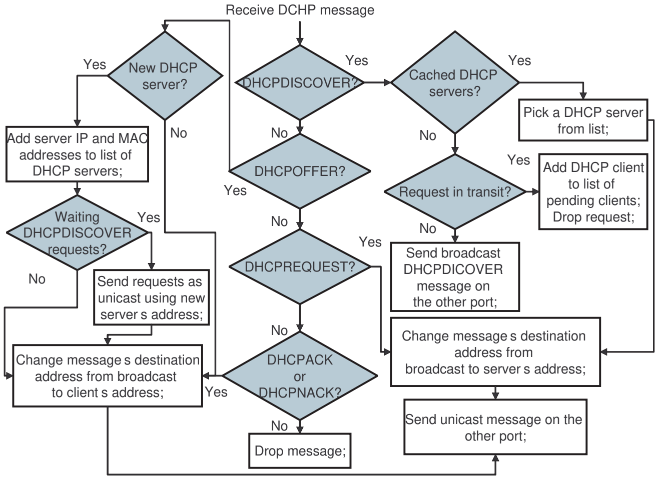


Fig. 3. A flowchart of handling DHCP packets by the DHCP module

with DHCPOFFER messages. Consequently, the EtherProxy learns both the MAC and IP addresses of all DHCP servers in the network. If more DHCPDISCOVER requests arrive from other clients before the EtherProxy receives a DHCPOFFER, it buffers those requests until a DHCPOFFER message arrives. Then, pending DHCPDISCOVER requests are sent as unicast packets to the DHCP server's address that was learned from the received DHCPOFFER message. The EtherProxy maintains the list of DHCP servers by periodically pinging them. If a server does not respond, it is removed from the list. Moreover, the EtherProxy periodically, with a low frequency, broadcasts a DHCPDISCOVER message, it received from a client, through the network. This forces all the DHCP servers to respond, allowing the EtherProxy to update its list of DHCP servers, adding any new DHCP servers in the network. Figure 3 shows a flow chart of how a received DHCP message would be handled by the DHCP module in an EtherProxy.

The DHCP module is similar in functionality to the DHCP relay agent [15], which allows a client running in one broadcast domain to reach a DHCP server running in another domain by relaying the DHCP messages. However, unlike the DHCP module, a DHCP relay agent must be configured with the IP addresses of DHCP servers in the network. Conversely, since the EtherProxy runs in the same broadcast domain of the DHCP server, it can discover the address of the DHCP server using broadcast. Hence, the EtherProxy achieves the best of both worlds. It substantially suppresses broadcast traffic to achieve scalability. Yet, it can configure itself using broadcast as it allows the whole network to run in one broadcast domain.

B. Discussion

As we have seen in this section, EtherProxy's modules are straightforward and easy to implement. Moreover, as we will see in Section III-A, the vast majority of broadcast traffic in a network comes from only a small set of protocols. Hence, it is simple to build an EtherProxy that suppresses most of the broadcast traffic in the network. The rest of this section discusses different issues concerning the EtherProxy.

1) Number of Hosts behind an EtherProxy: An EtherProxy suppresses broadcast traffic resulting from hosts behind it.

However, broadcast traffic from one host behind an EtherProxy, P, reaches all the other hosts that reside behind P. Hence, it is preferable to have the number of hosts behind an EtherProxy within Cisco's guidelines shown at Table I.

2) Handling Misbehaving Hosts: A malicious or a misconfigured host can flood the EtherProxy with broadcast packets. Those packets could be recognized by the EtherProxy forcing it to process them, e.g. ARP requests, hence overwhelming the EtherProxy. Conversely, those broadcast packets can be unrecognized by the EtherProxy, forcing it to let them pass to the network, which can flood the network. To counter this threat, an EtherProxy should employ broadcast filters that are common in many Ethernet switches [2]. A broadcast filter limits the number of broadcast packets received per unit time to a threshold. This threshold can be configurable, but should be preset to a default value. An EtherProxy should impose a broadcast filter per host. Moreover, it can have different filters for different protocols, hence handling different protocols in different ways. This can limit both the recognized and unrecognized broadcast packets flowing into the EtherProxy, protecting it from getting overwhelmed. It can also guard the network against broadcast storms.

3) Interplay Between Multiple EtherProxies: Many EtherProxies may reside in a single network. Hence, a packet may cross more than one EtherProxy during its trip from the source to its destination. An EtherProxy only interferes with broadcast packets. Thus, if the first EtherPRoxy the broadcast packet crosses modifies its destination address from a broadcast address to a unicast one, other EtherProxies will not interfere with the packet. Conversely, if the first EtherProxy does not modify the packet's destination broadcast address, the following EtherProxy will try to handle the packet. This will keep going, until either the packet reaches its final destination or one EtherProxy along the way is able to handle the packet by sending back a response or by changing the packet's broadcast address into a unicast one.

C. Implementation Issues

The functionality of an EtherProxy can be implemented in Ethernet switches. However, to be able to take advantage of the EtherProxy's benefits without having to change the existing infrastructure, it is recommended to have the EtherProxy as a stand-alone device. Hence, it can be simply added to exiting Ethernet networks.

The memory requirements of an EtherProxy depend on the modules it includes and the size of the network it serves. The ARP module's memory requirements scale with the number of entries in its cache. Thus, at the maximum, they scale with the number of hosts in the network. On the other hand, the DHCP module just needs to maintain a list of DHCP servers. In conclusion, the memory requirements of an EtherProxy are limited even if they need to operate in a huge network.

III. EVALUATION

In this section we first characterize broadcast traffic in Ethernet networks. Then, we evaluate the effectiveness of the EtherProxy in suppressing this broadcast traffic.

	Rice	Yahoo!	Emulab
Count of source addr.	128	409	303
Count of destination addr.	1216	456	2604

TABLE III
COUNT OF DISTINCT ADDRESSES OBSERVED IN ARP TRACES OF DIFFERENT NETWORKS OVER A 24 HOUR INTERVAL ON A WEEKDAY.

A. Characterization of Broadcast Traffic in Ethernet Networks

To study the broadcast traffic in Ethernet networks, we collected 24 hour packet traces from three representative networks on weekdays. We collected traces from: (1) a network in the Computer Science Department at Rice University representing a campus network workload, (2) a Yahoo! data center network representing a data center network workload, and (3) the Emulab control network. Each trace is collected from a single VLAN, i.e., a single broadcast domain. Table II shows a breakdown of the broadcast traffic observed in those networks. We notice that the vast majority of the measured broadcast traffic is ARP. ARP traffic constitutes 87.4%, 88.2%, and 85% of the whole broadcast traffic in the traces from Rice, Yahoo!, and Emulab respectively.

In the Rice trace, in addition to ARP, we also observe broadcast traffic from NetBIOS, Netware IPX/SPA, Portmap using broadcast Sun Remote Procedure Call (SUNRPC), DHCP, the Internet Printing Protocol (IPP), the Microsoft SQL server, the X Window System protocol (X11), the X Display Manger Control Protocol (XDMCP), and the Open System Interconnection (OSI) protocol.

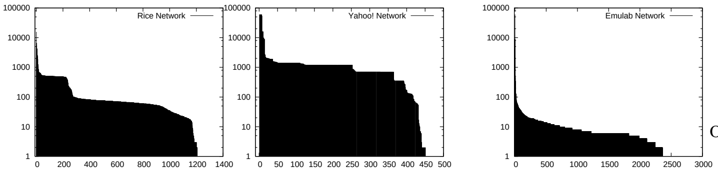
For the Yahoo! trace, other than ARP, it only contains broadcast traffic from the Internet Security Association and Key Management Protocol (ISAKMP), and NetBIOS. While in the Emulab trace, we only see ARP, DHCP, and NetBIOS broadcast traffic.

Since ARP was the major source of broadcast traffic in the three networks we studied, we extracted the ARP traffic and studied it further. From the three ARP traces, we counted the number of source and destination addresses observed. Those numbers are shown in Table III. Table IV shows the per source per second peak and average ARP traffic in the network for all traces. The average load is measured by dividing the total amount of ARP traffic observed in the network by the total duration of the experiment, while the peak load is measured using a one second sliding window over each trace. Then for every trace, the peak and average loads are divided by the number of active sources in that trace. We note that the peak load in the Rice trace, 6.23 request/source/second, is about 300 times more than the average case. In the Emulab trace, the peak load, 2.1 request/source/second, is over 500 times more than the average case. On the other hand, while the Yahoo! network has the highest average load, it does not suffer from major spikes in the ARP traffic. Its peak load is only 0.25 request/source/second. Those loads are small as VLANs have divided the network into small broadcast domains, hence limiting the amount of broadcast traffic per domain.

Figure 4 shows the rankings of destination addresses in ARP requests from the traces of Rice, Yahoo!, and Emulab along with the measured requests count for each address. An

	ARP	IPX/SAP	NetBIOS	SUNRPC	DHCP	IPP	MS-SQL	X11	XDMCP	OSI	ISAKMP
Rice Network	235735	405	19560	9590	1272	3014	4	8	4	122	0
Yahoo! Network	986248	0	237	0	0	0	0	0	0	0	131672
Emulab Network	110586	0	328	0	19154	0	0	0	0	0	0

TABLE II
BREAKDOWN OF THE NUMBER OF BROADCAST PACKETS MEASURED IN DIFFERENT ETHERNET NETWORKS OVER A 24 HOUR INTERVAL ON A WEEKDAY.



(a) Rice Network (b) Yahoo! Network (b) Emulab Network

Fig. 4. Rankings of destination addresses in ARP requests based on counts of ARP requests for each of these addresses and their corresponding requests count in the three traces. The X-axis represents the address rankings. The Y-axis represents the number of requests sent for this address. Note that the Y-axis has a logarithmic scale.

	Rice	Yahoo!	Emulab
Average Load (req/src/sec)	0.021	0.028	0.004
Peak load (req/src/sec)	6.23	0.25	2.1

TABLE IV
CHARACTERIZATION OF ARP TRAFFIC MEASURED IN DIFFERENT NETWORKS OVER A 24 HOUR INTERVAL ON A WEEKDAY.

address's rank is based on how many ARP requests were querying this address in the trace. We note that number of requests per address follow a heavy tailed distribution, where few addresses receive a lot of ARP requests.

B. Effects of Large Volume Broadcast Traffic on Saturating Links' Capacity

Broadcast traffic is forwarded on every link in the Network. Thus, a large amount of broadcast traffic can saturate links, especially the slow ones. From Tables III and IV in the Rice trace, we see that at peak load 128 clients can send 398.7 Kb/s of ARP traffic. Extrapolating this result and assuming that the total amount of ARP traffic grows linearly with the number of hosts, then less than 32,000 hosts in the same broadcast domain can saturate 100 Mb/s network links with their peak load ARP traffic.

C. Effects of Large Volume Broadcast Traffic on End Hosts

In this experiment we studied the effects of having a lot of broadcast traffic on end hosts. To conduct this experiment we used two Emulab nodes having 3 Ghz Xeon processors. On one machine, we ran an ARP traffic generation tool arp-sk, which we optimized to send ARP requests with a higher rate. The machine generating ARPs was sending ARPs, to an unknown IP, as fast as it could, which was 300,000 ARP request/second. On the other machine, we ran different operating systems and measured the corresponding CPU utilization, when subjected to this high volume ARP traffic. Table V shows the measured CPU utilizations for Linux Fedora Core 4 with kernel 2.6.12, FreeBSD 6.2, and Windows XP with service pack 2. We note that under Windows XP, the CPU is fully utilized under this load.

Again, extrapolating the result from Tables III and IV, we can conclude that less than 50,000 hosts in the same Ethernet

	Linux FC4	FreeBSD 6.2	Windows XP
CPU Utilization	23%	35%	100%

TABLE V
CPU UTILIZATION OF A 3 GHZ XEON MACHINE RUNNING DIFFERENT OPERATING SYSTEMS WHEN RECEIVING 300,000 ARP REQUEST/SECOND.

network can produce 300,000 ARP request/second at peak load. Thus, having an Ethernet network with a single broadcast domain and less than 50,000 hosts can saturate the CPUs of machines with 3 Ghz Xeon processors running Windows XP from just processing ARP requests.

Since hosts with different CPU speeds can be connected to the network, then the network should accommodate machines with the slowest speeds. This could be orders of magnitude slower than the 3 Ghz Xeon. Thus, much less than 50,000 hosts can saturate the CPUs of those slow machines. This explains Cisco's conservative recommendation for a broadcast domain size given in Table I.

D. Effectiveness of the ARP Module

To study the effectiveness of the ARP module in suppressing broadcast ARP requests, we studied the hit rates of the ARP cache in the ARP module that were achieved under different workloads. A cache hit corresponds to a suppressed broadcast ARP request as the request will be served from the cache instead of being flooded throughout the network. Hence, the cache hit rate corresponds to the broadcast suppression rate.

In our experiments, we connected an EtherProxy having the ARP module to a client machine running an ARP generator at one side. The ARP generator represents the hosts behind the EtherProxy that send ARP requests for the other hosts in the network in front of the EtherProxy. On the other side of the EtherProxy, resides an ARP responder, which responds to ARP requests by sending back the corresponding ARP responses. The ARP responder represents the other hosts in the network in front of the EtherProxy.

In our experimental setup, the client machine was running the ARP traffic generation tool arp-sk. We modified arp-sk to choose the target IP address in the generated ARP requests based on either a trace file or according to a random distribution. This distribution follows the distribution given in Figure 4(a). The EtherProxy was implemented using the Click modular router [9] and was configured such that entries in its ARP cache expire after 5 minutes; the same value used by Sun Microsystems in Solaris [19]. The ARP responder is implemented using the Click modular router as well. We ran our experiments on the Emulab testbed [20]. In the experiments, we measured the warm cache hit rate at the ARP module under different workloads. To study the effects of sending unicast ARP requests to refresh expiring cache entries on the cache's hit rate, we repeated the experiments with automatic refresh switched on and off.

	Rice	Yahoo!	Emulab
No refresh	61.4%	99.98%	79.6%
Refresh	100%	100%	100%

TABLE VI
HIT RATES OF THE ARP CACHE IN THE ARP MODULE FOR RICE, YAHOO!, AND THE EMULAB TRACES.

1) Evaluation Using Real Workloads: For the first set of experiments, we used real workloads using the ARP traces we collected for Rice, Yahoo!, and the Emulab networks. Table VI shows the measured hit rates for the three traces with automatic refresh switched on and off.

In those experiments, the ARP cache is large enough to accommodate mappings for every IP in the network, i.e. there are no capacity misses. Hence, when cache entries are refreshed by the ARP module the cache does not suffer from any misses and all broadcast ARP requests are eliminated from the network. We notice that when the cache entries are not refreshed there is significant cache miss rate for the Rice and Emulab traces. However, the Yahoo! trace has significantly more active sources and less destinations. Hence, much more ARP requests are sent per address, preventing those address mappings from expiring in the cache. Consequently, the Yahoo! trace experiences high cache hit rate even with automatic refresh switched off.

2) Evaluation Using Synthetic Workloads: To study the effectiveness of the ARP module in very large networks we concocted synthetic workloads using the measurements presented in Section III-A. This is because in practice large networks are segmented into smaller broadcast domains. Hence, it is difficult to get real traces for very large networks with a single broadcast domain.

In our synthetic workloads, we assumed that there are 500 hosts sitting behind the EtherProxy sending ARP requests. We chose the relatively small value for the number of hosts behind the EtherProxy, 500, as a broadcast packet from a host behind an EtherProxy can reach all other hosts behind the EtherProxy. Hence, we used Cisco's recommended value for a broadcast domain in Table I, 500.

We varied the rate at which hosts send ARP requests representing different traffic loads. An ARP request is sent for an IP address selected from a range of IP addresses. This range is fixed for every experiment, but can change across different experiments. The destination IP address is chosen based on a probability distribution function based on the distribution shown in Figure 4(a).

In different experiments, we varied the size of the pool of destination IP addresses representing networks with different sizes. For a host sending ARP requests for other hosts in the network with a given rate, the more hosts in the network, the less ARPs will be sent for the same host. Thus, the bigger the network size, the less reuse of ARP cache entries that there is in the EtherProxy. Consequently, we varied the network size to study the effects of the number of hosts in the network on the cache hit rate, when cache entries are not refreshed.

Table VII shows the measured cache hit rates using different configurations of the synthetic workload. In this experiment we vary the number of hosts in the network from 100 to 50,000. We also vary the rate at which hosts send ARP requests from 60 requests/minute down to one request/minute. Similar to the experiments using real world traces, the ARP cache is large enough to accommodate mappings for every IP address in the network. Hence, when cache entries are refreshed the cache does not suffer from any misses. We notice that when the cache entries are not refreshed there is a significant cache miss rate. This is especially true when requests arrive at a slow rate at the EtherProxy. This is because mapping reuse increases with high request rate, which increases the cache hit rate.

	60 req/min	10 req/min	1 req/min
50K hosts	81.7%	55%	17.2%
10K hosts	94.2%	78.7%	42.3%
100 hosts	99.99%	99.7%	96.1%

TABLE VII
HIT RATES OF THE ARP CACHE IN THE ARP MODULE, WHEN CACHE ENTRIES ARE NOT REFRESHED VIA UNICAST ARP REQUESTS. THE EXPERIMENT WAS REPEATED FOR NETWORKS WITH DIFFERENT SIZES AND FOR HOSTS SENDING ARP REQUESTS WITH DIFFERENT RATES. THE ARP CACHE ENTRY EXPIRES AFTER 5 MINUTES.

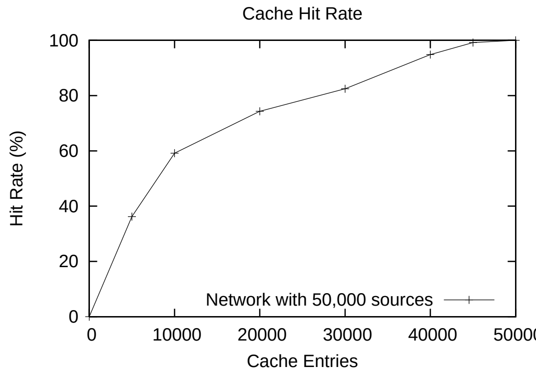


Fig. 5. Measured ARP cache hit rates, when varying the cache size. In this experiment, there were 50,000 sources sending ARP requests with the rate of 1 request/second. Destination IP addresses in the ARP requests were chosen based on the probability distribution shown in Figure 4(a).

3) Sensitivity Analysis of the ARP Cache Size: In this experiment we study the effects of varying the ARP cache size on the hit rate. We use a synthetic workload that was used in Section III-D2. The workload represents 500 hosts behind an EtherProxy sending ARP requests at the rate of 1 request/second. The hosts send those requests to destinations based on a probability distribution function based on the distribution shown in Figure 4(a). The network has 50,000 hosts in front of the EtherProxy, which are the targets for those ARP requests. In this experiment, the EtherProxy's ARP cache uses the LRU replacement policy. Figure 5 shows the measured cache hit rates at the EtherProxy's ARP cache, when varying the cache size. We note that since the workload has a heavy-tailed distribution, a relatively small cache size can achieve high hit rates.

E. Effectiveness of the EtherProxy as a Firewall

This experiment studies the effectiveness of the EtherProxy as a firewall blocking spurious ARP requests. We use the topology in Figure 6. Boxes labeled B are software Ethernet bridges like the ones used in [5]. P is an EtherProxy. H1 and

	Behind EtherProxy	Infront of EtherProxy
Packet Count	1	47477

TABLE VIII
NUMBER OF ARP PACKETS MEASURED ON BOTH SIDES OF THE ETHERPROXY, WHEN AN ARP REQUEST IS INJECTED IN THE NETWORK AT THE BEGINNING OF A 5 SECONDS TEMPORARY FORWARDING LOOP IN THE NETWORK. DURING THE FORWARDING LOOP, THE ARP REQUEST SPINS AROUND THE FORWARDING LOOP, FLOODING THE NETWORK WITH DUPLICATE ARP REQUESTS.

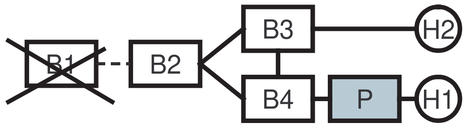


Fig. 6. Topology used to study the effectiveness of the EtherProxy as a firewall blocking spurious ARP requests.

H2 are hosts, where H1 runs a Click ARP responder and H2 is a client that sends ARP requests. H1 represents the rest of the network that may have many end hosts sitting behind the EtherProxy. These end hosts respond to ARP requests arriving at them. We use Emulab to run this experiment. In this experiment, we create a temporary forwarding loop by killing the bridge B1, which causes the bridges to count to infinity, causing a temporary forwarding loop. For more information on count-to-infinity induced forwarding loops, please refer to [6]. This temporary loop lasted for 5 seconds. At the beginning of the forwarding loop, H2 sends a single ARP request for an IP address that the ARP responder responds to. Since the ARP request is a broadcast packet, when it gets trapped in the forwarding loop, it proliferates. Hence, it floods the network with duplicate ARP requests. The first of those ARP requests arrives at the EtherProxy causing it to send an ARP request to the ARP responder at H1. This is because the EtherProxy does not have in its cache a mapping for the IP address requested by the ARP request. After the ARP responder responds to the EtherProxy, a cache entry is formed at the EtherProxy. Subsequent duplicate ARP requests arriving at the EtherProxy are not forwarded to H1, as the EtherProxy sends back a response from its cache directly. This protects H1, or the rest of the network, from the extra spurious ARP requests in the case of this failure. Table VIII shows the number of ARP requests measured at both sides of the EtherProxy. We note that H1 only receives a single ARP request, the first request. Other end hosts in the network - H2 in this case, where we measured the duplicate ARP request packets received - received 47477 ARP request packets.

F. Effectiveness of the DHCP Module

The DHCP module deterministically eliminates all DHCP broadcast traffic from the network by converting broadcast destination addresses to unicast ones. However, it only adds very low frequency broadcast DHCP requests to discover new DHCP servers joining the network.

IV. OTHER SCALABILITY CONCERNS IN ETHERNET

In this section we cover other scalability concerns in Ethernet networks.

A. Flooding Due to Address Learning

Ethernet uses flooding to deliver a packet to a new destination address whose topological location in the network is unknown. An Ethernet switch can observe the flooding of a packet to learn the topological location of an address. Specifically, a switch observes the port at which a packet from a particular source address S arrives. This port then becomes the outgoing port for packets destined for S and so flooding is not required to deliver future packets to S. This information is recorded by a switch in an entry in its forwarding table. A forwarding table entry expires after a timeout if it is not used. This timeout has a default value of 5 minutes. If an entry expires for an address, flooding is used again to relearn the address's topological location.

To study the effectiveness of the address learning in eliminating flooded packets, we used the Rice trace from Section III. This trace was collected using tcpdump on a host connected to the Rice network. We configured the host's network card to run in promiscuous mode. This directs the card to not only accept packets destined to this host, but also accept unicast packets destined to other hosts. Thus, it will accept unicast packets destined for other hosts, but flooded through the network for address learning purposes.

In the trace, we counted 10859 packets that are neither multicast packets nor destined for the host where we collected the trace. Thus, those were flooded packets through the network. We note that, in the same trace, the ratio between measured flooded packets and broadcast packets, from Table II, is 1:25. Hence, address learning is highly effective in alleviating Ethernet's scalability concern due to flooding.

B. Forwarding Table Size in Ethernet Switches

Ethernet switches store learned host locations, along with the corresponding MAC addresses in their forwarding tables. Since MAC addresses are flat and nonhierarchial, the forwarding table size may need to scale with the number of hosts in the network. However, if there is locality in the communication patterns between hosts in the network, forwarding tables will not need to scale with the number of hosts in the network. In fact, studies of enterprise traffic show that communication between hosts exhibit a significant amount of locality [14], [1]. Most hosts only communicate with a few other hosts in the network. Consequently, a commodity switch that can handle 8000 MAC addresses, can serve a medium sized network. Moreover, with improvements in silicon technologies, switches that can handle 512,000 MAC addresses already ship today [7].

C. Traffic Concentration at the Root of the Spanning Tree

The forwarding topology in an Ethernet network is a spanning tree. Thus, traffic due to communication between hosts on different sides of the root of the tree will have to pass through the root. This may become a bottleneck and thus a scalability problem. However, again, if there is locality in the communication patterns between hosts in the network, it will not be common for hosts on different sides of the tree to communicate. Also, links towards the root of the tree could be designed to have more bandwidth, i.e. use a *fat tree* topology. Moreover, the EtherProxy, suppressing broadcast traffic from the network, spares the valuable bandwidth at the root.

V. Related Work

Previous work has argued for the need for scaling Ethernet. Broadcast was identified as a source of the scalability problem and some solutions were proposed. Myers *et al.* [11] argued that the scalability of Ethernet is severely limited because of its broadcast service model. In order to scale Ethernet to a much larger size, they proposed the elimination of the broadcast service from Ethernet and its replacement with a new control plane that does not perform packet forwarding based on a spanning tree and provides a separate directory service for service discovery. In this approach, a host joining the network needs to first register with the directory service via its local switch. Then, it can receive the bootstrapping information by querying the directory service. Hence, unlike the EtherProxy, this approach is not backward compatible.

SEATTLE [8] proposed a new architecture for building layer 2 networks. To handle broadcast, switches participate in a distributed hash table to cache ARP table entries. Switches can also convert broadcast DHCP messages to unicast ones. However, SEATTLE does not provide a general backward compatible approach to handle broadcast traffic used by many other protocols. Moreover, SEATTLE switches do not interoperate with regular Ethernet switches.

Several other previous works have addressed the inefficiency of spanning tree routing in Ethernet. Perlman *et al.* [16] argued that Ethernet has poor scalability and performance and proposed Rbridges to replace the current Ethernet protocols. Routing in Rbridges is based on a link state protocol to achieve efficient routing. However, Rbridges do not address the scalability problem due to broadcast.

EtherProxies can complement Rbridges and SEATTLE. They can be adapted to work with Rbridges or SEATTLE switches to alleviate the broadcast scalability problem.

Other previous work has addressed the reliability problems of Ethernet. EtherFuse [5] provides a backward compatible solution to those problems. An EtherFuse can be inserted into existing Ethernet networks to improve its convergence time and guard against forwarding loops. RSTP with Epochs [6] modifies Ethernet's Rapid Spanning Tree Protocol (RSTP) to eliminate the count-to-infinity problem and consequently improve Ethernet's convergence time.

VI. Conclusions

Broadcast traffic is a major limiting factor in Ethernet's scalability. It is widely used by many protocols running on top of Ethernet. Although segmenting an Ethernet network into separate broadcast domains alleviates the scalability problem, it does not come without a cost. It requires a lot of error-prone manual labor. Moreover, it is not cost effective. To address Ethernet's scalability problem without having to pay the cost of network segmentation, we introduce the EtherProxy. The EtherProxy is a new device that includes a different module for every protocol using broadcast. Modules rely on caching protocol information to suppress broadcast traffic. EtherProxy is backward compatible and requires no change to the existing hardware, software, or protocols.

We studied the broadcast traffic in Ethernet networks using real and synthetic workloads. Our measurements show that only a small set of protocols, in practice, produce the vast majority of broadcast traffic in the network. Hence, it is realistic to have individual modules to handle this handful of protocols. Using the real and synthetic workloads along with a prototype we implemented of the EtherProxy, we performed experiments demonstrating the EtherProxy's effectiveness in suppressing broadcast traffic.

References

[1] W. Aiello, C. Kalmanek, P. McDaniel, S. Sen, O. Spatscheck, and J. V. D. Merwe. Analysis of Communities of Interest in Data Networks. In *Passive and Active Measurement*, Mar. 2005.

[2] Cisco Systems, Inc. Configuring Broadcast Suppression. At http://www.cisco.com/en/US/docs/switches/lan/catalyst6500/ios/12.%1E/native/configuration/guide/bcastsup.pdf.

[3] Digital Equipment Corporation. DIGITAL Network Architecture–Maintenance Operations Functional Specification. DIGITAL Standard 200-1, Document number A-DS-EL00200-01-0000, 1988.

[4] R. Droms. RFC 2131: Dynamic Host Configuration Protocol, Mar. 1997.

[5] K. Elmeleegy, A. L. Cox, and T. S. E. Ng. EtherFuse: An Ethernet Watchdog. In *ACM SIGCOMM 2007*, Aug. 2007.

[6] K. Elmeleegy, A. L. Cox, and T. S. E. Ng. Understanding and Mitigating the Effects of Count to Infinity in Ethernet Networks. *IEEE/ACM Transactions on Networking*, February 2009.

[7] Force10 Networks, Inc. The Force10 EtherScale architecture. At http://www.force10networks.com/whitepapers/pdf/wp_ArchOverview.pdf, 2005.

[8] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *ACM SIGCOMM 2008*, Aug. 2008.

[9] E. Kohler, R. Morris, B. Chen, J. Jannotti, , and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[10] D. L. Mills. RFC 1305: Network Time Protocol (Version 3) Specification, Implementation, Mar. 1992.

[11] A. Myers, T. S. E. Ng, and H. Zhang. Rethinking the Service Model: Scaling Ethernet to a Million Nodes. In *Third Workshop on Hot Topics in networks (HotNets-III)*, Mar. 2004.

[12] NetBIOS Working Group. RFC 1002: Protocol standard for a NetBIOS service on a TCP/UDP transport: Detailed specifications, Mar. 1987.

[13] P. Oppenheimer. *Top-Down Network Design*. Cisco Press, secod edition, 2004.

[14] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A first look at modern enterprise traffic. In *Internet Measurment Conference*, pages 15–28, 2005.

[15] M. Patrick. RFC 3046: DHCP Relay Agent Information Option, Jan. 2001.

[16] R. Perlman, D. Eastlake, S. Gai, and D. G. Dutt. Rbridges: Base Protocol Specification. Internet-Draft (work in progress) draft-ietf-trill-rbridge-protocol-06.txt, IETF, November 2007.

[17] D. C. Plummer. RFC 826: Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware, Nov. 1982.

[18] J. Postel. RFC 925: Multi-LAN address resolution, Oct. 1984.

[19] Sun Microsystems, Inc. Solaris Operating Environment Network Settings for Security. At http://www.sun.com/blueprints/1200/network-updt1.pdf, Dec. 2002.

[20] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, Dec. 2002.