UNIVERSITY OF AMSTERDAM

GRADUATE SCHOOL OF INFORMATICS
System and Network Engineering

Minimizing ARP traffic in the AMS-IX switching platform using OpenFlow

Victor Boteanu <vboteanu@os3.nl>
Hanieh Bagheri <hanieh.bagheri@os3.nl>

July 22, 2013

Supervisor
Martin Pels <martin.pels@ams-ix.net>

University of Amsterdam
Graduate School of Informatics
Science Park 904
1098XH Amsterdam

Contents

List of Figures

Abstract

The AMS-IX ISP Peering LAN connects more than 600 routers. These routers use ARP in order to establish connectivity with each other. With such a large network, it can be expected that there are nodes down at any moment. Even if that is not the case, there are also ARP Requests for IP addresses no longer in use. This causes severe broadcast traffic in the network.

All the nodes that are listening for ARP messages have to further process the packets, meaning that all nodes in the network will spend CPU cycles to process these messages. This is inefficient, as most of the time those messages do not concern them, and some devices even prioritize ARP processing over, for instance, routing protocol jobs.

So far, the solution that AMS-IX has implemented is the ARP Sponge. This tool reduces the amount of broadcast traffic on the peering LAN by replying to ARP requests for dead addresses.

The goal of this research project is to investigate the use of OpenFlow to further bring down the amount of broadcast traffic on the AMS-IX peering LAN.

1 Introduction

1.1 AMS-IX Overview

The Amsterdam Internet Exchange (AMS-IX) is one of the leading Internet exchanges in the world, with an average total traffic load of 1.47 Tb/s (June 2013)*. An Internet exchange is a kind of meeting point for Internet Service Providers (ISPs) where they can connect to each other. This allows ISPs to minimize upstream traffic to higher-tier ISPs.

These connections are peering relations set up using the Border Gateway Protocol (BGP). At the time of this writing, the AMS-IX network has almost 600 connected networks **, with a peak traffic of 2 Tb/s.

Due to the continuous increase in traffic over the years, AMS-IX redesigned their network topology in 2009, now called AMS-IX v4. The peering platform is based on an MPLS/VPLS [2][3] infrastructure, due to the protocol's high scalability and resiliency.

An overview of the current peering LAN is shown in figure 1.1:

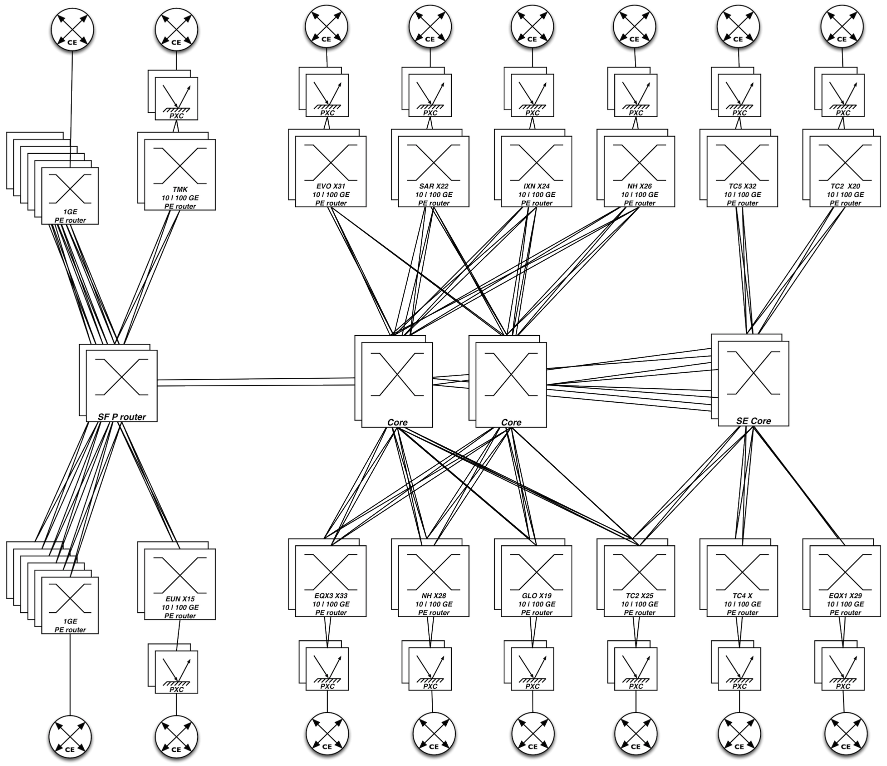

CE
PXC
1GE PE router
TMK 10 / 100 GE PE router
EVO X31 10 / 100 GE PE router
SAR X22 10 / 100 GE PE router
IXN X24 10 / 100 GE PE router
NH X26 10 / 100 GE PE router
TC5 X32 10 / 100 GE PE router
TC2 X20 10 / 100 GE PE router
SF P router
Core
Core
SE Core
1GE PE router
EUN X15 10 / 100 GE PE router
EQX3 X33 10 / 100 GE PE router
NH X28 10 / 100 GE PE router
GLO X19 10 / 100 GE PE router
TC2 X25 10 / 100 GE PE router
TC4 X 10 / 100 GE PE router
EQX1 X29 10 / 100 GE PE router
PXC
CE

Figure 1.1: AMS-IX network overview

The entire network is built around eight core routers (Brocade MLXE-32), and each Provider Edge (PE) router has connections to multiple core routers. This allows load

*https://www.ams-ix.net/technical/statistics

**https://www.ams-ix.net/connected_parties balancing over all core routers and also provides redundancy for the Provider (P) routers. The P routers are at two different locations, with two P routers per location. In case one of the core sites goes down, all the Label Switched Paths (LSPs) will be moved and load-balanced over the remaining core routers.

LSP signalling is done using Resource Reservation Protocol-Traffic Engineering (RSVP-TE), while Label Distribution Protocol (LDP) is used in the control plane to distribute the VPLS labels. Using VPLS allows for the emulation of a LAN in the shared Layer 2 network.

To account for cases in which one of the PE routers fails, Customer Edge (CE) routers are connected to the PE routers through a Photonic Cross-Connect (PXC). A PXC is a network device used to switch optical signals without converting them to an electrical signal. In the case of a PE failure, the PXC will forward all traffic of its connected routers to the other PE, as illustrated in figure 1.2.

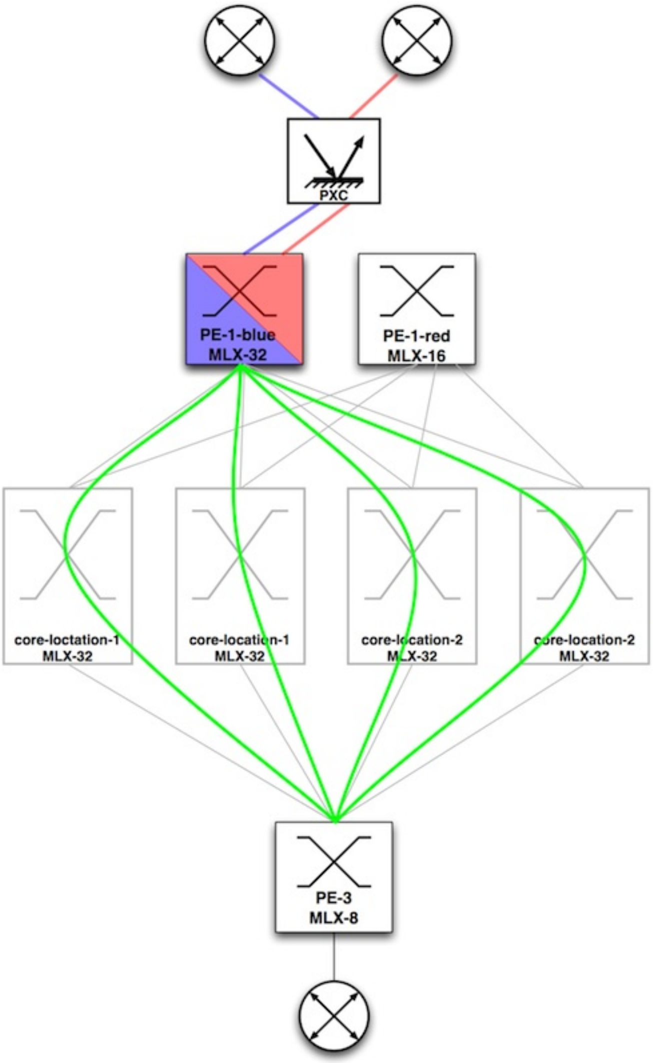


Figure 1.2: Failover in case of PE failure

Current connections to the AMS-IX network use Gigabit Ethernet (GE), 10 GE, 100 GE, or multiples of these values. For Gigabit connections, the access devices are Brocade MLX-8 routers, while Brocade MLX-16 and MLX-32 routers are used for the 10 GE and 100 GE connections.

1.2 The Address Resolution Protocol

The routers connected to each other on the AMS-IX peering LAN use the Address Resolution Protocol (ARP) in order to establish connectivity with each other. ARP is the protocol used for associating a layer 3 address with a layer 2 address. In other words, ARP is used to bind a logical network address (such as IP) to a physical link layer address (such as MAC). ARP is a request-reply protocol: a client router sends an ARP request asking about the MAC address corresponding to a given IP address. The ARP request is broadcast in the network and the node owning that IP address announces its MAC by sending a unicast ARP reply.

Each node has an ARP table, which is a mapping between IP and MAC addresses. When a node wants to send data to another IP[*] address, first it looks up the IP in the ARP table. If there is a hit, the Ethernet frame will be sent towards the matching MAC address; otherwise, an ARP request will be broadcast in the network and the node waits for the ARP reply. When the ARP reply arrives at the node, the MAC address is extracted.

If a node sends an ARP request and does not receive back any reply in a certain amount of time, the same ARP request will be broadcast in the network over and over again, until a node sends an ARP reply for that IP address. In such a large network, it can be expected that there are nodes down at any moment. Even if that is not the case, there are also ARP requests for IP addresses no longer in use. When a client is down and other nodes try to reach it, a huge amount of ARP requests will be flooded in the network, causing an "ARP storm".

Because of the growing number of connected routers, ARP storm is a serious issue in the BGP peering LAN of AMS-IX. When a node is not accessible for a while, every node looking for it will flood an ARP request. So by adding new nodes to the peering LAN, the amount of ARP broadcast will grow exponentially.

All the nodes that are listening for ARP messages have to further process packets with Ethertype 0x0806, meaning that all nodes in the network will spend CPU cycles to process these messages. This is inefficient, as most of the time those messages do not concern them, and some devices even prioritize ARP processing over, for instance, routing protocol jobs. Therefore, although ARP does not use too much bandwidth (60 kbps daily [**]), the efficiency of the network can be improved by reducing the allocated processing power for broadcast traffic.

1.3 The ARP Sponge

To solve the problem of broadcast ARP traffic, a tool called the ARP Sponge was developed at AMS-IX. It is a program that is meant to "sponge" ARP requests for dead IP addresses. In general, if the sponge sees too many ARP requests destined for a node that is not answering, it will start replying for them itself after the requests exceed a threshold (default 1000 unanswered queries with an average rate of 50 or more per minute[†]).

The outcome is that if a router with many peerings happens to go down, all its peers will start sending ARP requests for it, resulting in a broadcast storm. The same situation can arise when a peer leaves the network permanently but its peers have not updated their configurations. By answering ARP requests for these nodes, broadcast storms can be prevented. Figure 1.3 illustrates the ARP sponge's functionality.

By replying with its own MAC address, the ARP sponge black-holes traffic destined for nodes that do not exist anymore in the network. In this way, other nodes will not broadcast ARP requests for that node. In addition to this functionality, the ARP sponge

[*] Here, the term "IP" stands for IPv4. The IPv6 protocol does not use ARP and its functionality is replaced by the Neighbor Discovery Protocol (NDP).

[**] https://www.ams-ix.net/technical/statistics/sflow-stats/broadcast

[†] https://www.ams-ix.net/downloads/arpsponge/3.12.2/arpsponge-3.12.2/arpsponge.txt

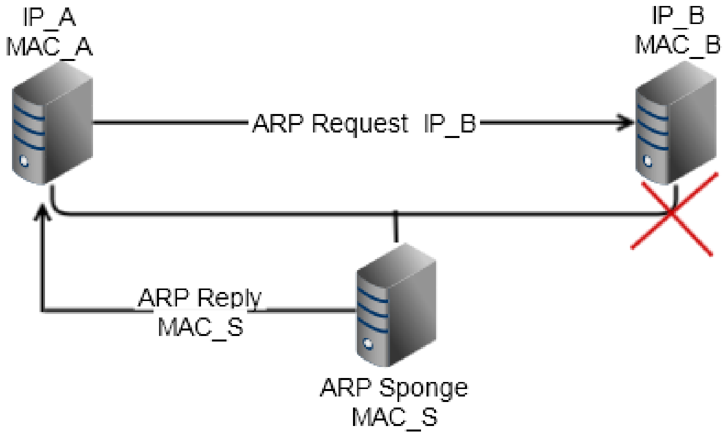


Figure 1.3: ARP sponge behaviour in the case of a down node

also periodically queries addresses to check if their interfaces are back up.

There are three cases in which it will stop sponging an address:

1. If the sponge receives a gratuitous ARP for that address
2. If the sponge sees any ARP or IP traffic from that address
3. If the sponge receives an ARP query for a sponged IP address that seems to come from IP 0.0.0.0 ("ARP WHO-HAS *xx* TELL 0.0.0.0"). This is used by many DHCP client implementations to detect duplicate addresses before accepting an address from the DHCP server). This is mostly a precaution measure.

Other important features of the ARP sponge that are relevant to this project are:

- At startup, it enters a learning state in which it listens to the network, building its own state table. During this state, it does not participate with traffic.
- When it starts sponging an address, it can send gratuitous ARP messages to the other nodes, updating their caches such that they do not have to ARP for it any more.
- It can put addresses in a pending state, in which it probes them for a response. If no response is received, the nodes are declared dead.
- The ARP sponge can periodically sweep the IP addresses of a network in order to account for nodes that do not send gratuitous ARP when they come up.
- Logging

As it will later be shown in Section 2, the state table that the sponge builds can be very useful. The table maps IP addresses to their corresponding MAC addresses, and keeps their state (ALIVE/DEAD/PENDING) and last update time. By using the ARP sponge, broadcast traffic is reduced tenfold, as shown in a previous research paper, which aimed at extending the ARP sponge [4].

1.4 OpenFlow

Network appliance vendors normally do not provide open software for development and management of their commercial hardware. However, researchers have always been interested in testing new ideas and developing experimental protocols and the classical switches and routers do not provide the proper facilities for such activities. In addition, it is difficult for researchers to build their own hardware and software. Even if they succeed in doing so, the result of the tests and experiments might not match the conditions of a production network. OpenFlow is a communication protocol that allows researchers to test their new ideas at scale in a production network.

In a classic switch, the traffic forwarding component (data plane) and the decision

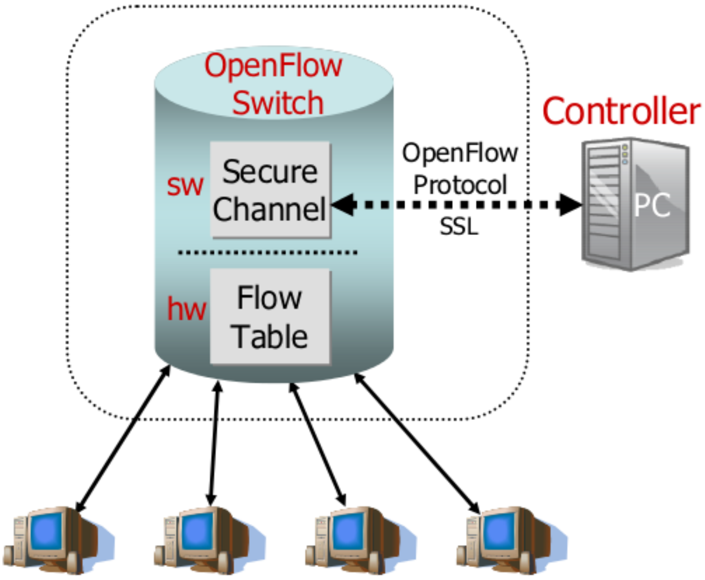


Figure 1.4: OpenFlow architecture [5]

making component (control plane) reside on the same device. Software-Defined Networking (SDN) is a method for separating these two components. OpenFlow is a communication protocol that enables SDN by getting access to the control plane of the switch or router.

In an OpenFlow-enabled switch, the data plane still resides on the switch, but the control plane is moved to a separate machine, called the "OpenFlow controller". The OpenFlow-enabled switch and the controller communicate via the OpenFlow protocol. `packet-received`, `send-packet-out`, `modify-forwarding-table` and `get-stats` are some of the exchanged messages in the OpenFlow protocol. OpenFlow provides an open platform for programming the flow table in an OpenFlow-enabled switch. As shown in figure 1.4 (from the OpenFlow 1.0 white paper), an OpenFlow switch consists of three parts [5]:

- A flow table, with an action for each flow entry of the table.
- A secure channel, which connects the OpenFlow controller to the switch and is used for exchanging packets and commands
- The OpenFlow protocol

Each entry in the flow table has three fields:

- A packet header, which is a set of fields to be matched
- The action that defines the way in which the packets are processed
- Statistics about the number of packets and bytes and also the elapsed time since the last packet is matched

Different switches support different OpenFlow actions. The three basic actions supported by all OpenFlow switches are:

- forwarding all the packets of a given flow to a specific port (or to all ports)
- encapsulating packets from a given flow and forwarding them to the controller
- dropping incoming packets from a specific flow

The OpenFlow controller is in charge of adding/removing flows to/from the flow table. A controller can control multiple switches simultaneously, and a switch can be controlled by different controllers at the same time. When the switch receives a packet, it tries to match the header with a flow entry (in a priority-based ordering). If there is a hit, the specified action will be executed. If it could not find a match, it will send the packet to the controller (in case the default action is forwarding to the controller). Depending on the script running in the controller, it might add some flows to the flow table to tell the switch how to handle similar packets, or it might drop the packet.

OpenFlow version 1.0 was released on December 31, 2009. Since its release, the OpenFlow protocol has been rapidly evolving. Support of groups, multiple tables, MPLS and VLAN, virtual ports, better handling of connectivity loss, are some of the added features to OpenFlow 1.1 (released on February 28, 2011). OpenFlow 1.2 was introduced on December 5, 2011. Some of the important features added from this version on are support of IPv6 and OpenFlow Extensible Match (OXM) for flexible field matching. OpenFlow 1.3 was released on April 13, 2012. This release adds support for more flexible table capability description, IPv6 Extension Header handling, per-flow-meters, per-event connection filtering, Tunnel-ID metadata. OpenFlow 1.3.1, released in August 2012, is the latest version and adds some minor improvements to the previous release.

Different OpenFlow controllers can be used in order to interact with the OpenFlow-enabled switch. Examples of these controllers are:

Open Source Controllers:

- Beacon (Java)
- Floodlight (Java)
- NOX (C++/Python)
- POX (Python)
- Trema (C/Ruby)
- Ryu (Python)
- Open Daylight (Java)

Closed Source / Commercial Controllers:

- Big Network Controller
- ProgrammableFlow
- ONIX

In this research, we aim to use OpenFlow in order to solve the ARP storm problem in the busy BGP peering LAN at AMS-IX. Since the original ARP protocol uses flooding to send ARP requests through the network, we found OpenFlow to be useful for changing the normal behaviour of the switch, while handling ARP messages.

1.5 Related Work

In an RP2 project from 2009, a related issue was analyzed by (at the time) SNE students Niels Sijm and Marco Wessel [4].

In a presentation at MENOG 12, Ivan Pepelnjak has talked about using OpenFlow on Internet Excahnge Points (IXPs)[6]. The discussed problems and proposed ideas in his talk are quite relevant to our research.

A team in New Zealand has deployed an SDN-controlled Internet exchange fabric. It is the first time that SDN is used in a production IXP environment (Citylink IX)[*]. The proposed ideas in this project can be inspiring to our work.

1.6 Research Question

Our main research question is the following:

"Can OpenFlow be used to reduce broadcast ARP traffic in the AMS-IX ISP peering LAN?"

We define these sub-questions:

"Can the ARP protocol be replaced completely by OpenFlow in the core network?"

"Is OpenFlow a scalable solution for this scenario?"

[*] http://pica8.org/blogs/?p=363

2 Proposed solutions

As we mentioned in the previous sections, the switching platform of AMS-IX connects a growing number of routers. The connected peers need to know about each other's MAC address to be able to communicate, so they use the ARP protocol to exchange this information. In this section, different approaches for solving the ARP storm problem are proposed and discussed in detail.

2.1 Solution 1: Forward to OpenFlow controller

In order to reduce the amount of ARP messages, we used the fact that the mapping between MAC and IP addresses of different peers is known beforehand. All the peering information is stored in a database compiled from an XML file. By extracting the IP-to-MAC mapping from the XML file and utilizing OpenFlow, we will be able to decrease the amount of ARP messages significantly.

The idea of the proposed solution in this section is to import the mapping table extracted from the .xml file into the controller and use it to reply to the incoming ARP requests. In other words, we do not use the ARP Sponge anymore and the controller acts as an ARP proxy to answer the ARP requests on behalf of the original nodes.

The steps in this solution are as follows:

1. The mapping table is imported to the controller
2. The controller installs a flow in the switch to forward the incoming ARP messages to the controller.
3. When an ARP request is forwarded from the switch to the controller, the controller looks up the IP address in the mapping table to find the corresponding MAC address.
4. If there is a match, the controller makes an ARP reply packet using the extracted MAC from the table; otherwise, it makes an ARP reply using the MAC address of the controller. This is done to prevent any more ARP requests from being sent. If the node is dead it would not get any of the traffic in either case, so black-holing has no negative effects.
5. The controller sends the ARP reply to the sender of the ARP request

Since the IP-to-MAC mapping table is inside the controller, which is connected to the PE switch, the ARP requests are not broadcast farther than the PEs, so we expect not to see any broadcast messages in the core network. In this way, not only can the controller answer the ARP requests on behalf of the original node, but it can also send the MAC address of a (temporarily) down or (permanently) dead node to the ARP sender. Therefore, we can prevent the problem of broadcast storms when a node is down or dead.

In this solution, the controller plays a similar role to the ARP Sponge present in the current infrastructure. The differences between the two are listed below:

- The ARP sponge starts "sponging" when it detects an unreachable address. By implementing this solution, the ARP requests would be sent directly to the controller instead of being broadcast.

- The ARP Sponge takes some time to find out if a node is dead or to detect if it is back again. This behaviour can cause false positives, where the requests are answered with the sponge's MAC address instead of the real node's. In the proposed solution, independent of the target node being up or down, the OpenFlow controller will send replies to the corresponding ARP requests destined for that node.
- The ARP Sponge uses its own MAC address to answer the ARP requests destined for a down or dead node. This solution normally replies to the ARP request using the MAC address of the node itself. In the case where there is no matching IP, the OpenFlow controller uses its own MAC address to reply to ARP requests.

2.2 Solution 2: Dynamic learning OpenFlow controller

In solution 1, we proposed the idea of exploiting the pre-known peering information to build an IP-to-MAC table. Another solution can be building the same table by learning the table entries from the exchanged ARP traffic in the network.

The steps in this solution are as follows:

1. The controller starts with an empty IP-to-MAC table.
2. The controller installs a flow in the switch to forward the incoming ARP message to the controller.
3. If there is a hit for a given ARP request, the controller makes an ARP reply packet using the extracted MAC address from the table; otherwise, it floods the ARP request to all ports. The controller should keep track of the flooded ARP requests to avoid sending ARP requests for the same IP address before the reply comes back.
4. If the ARP reply is not received in a certain amount of time, the controller assumes that there is no node with the requested IP address, so it replies to that request with its own MAC address. It also adds an entry with the same information in the IP-to-MAC table. This entry should time-out, so the nodes that do not support sending gratuitous ARPs still have the chance to receive ARP requests.
5. Whenever the controller receives an ARP reply or a gratuitous ARP, it updates its IP-to-MAC table.

This solution is more dynamic than the previous solution, because it updates the table according to occurred events in the network (rather than the configuration files being updated). Receiving ARP replies from different nodes, the controller fills up the table gradually. After a learning period, the controller will be able to answer most of the ARP requests using its table. If a node goes down, similar to the previous solution, the controller still answers ARPs on behalf of that node. In case of changes in configuration (adding a node to the network, changing the MAC address of an existing interface), when the node brings its interface up, it usually sends a gratuitous ARP to the controller and causes the proper entries be updated in the table.

Using this method, the table is updated by the real-time events happening in the network. It is an advantage over solution 1, which depends on the correctness of the configuration files. Although some ARP messages are still exchanged, each ARP request can just be flooded once.

The disadvantage of this option is the latency for discovering when a node is back up and it does not support gratuitous ARP. The controller will only be aware of the node's state when another client asks for the node's MAC address.

2.3 Solution 3: OpenFlow-aided ARP proxy on each PE

Because the functionality of the ARP sponge has been thoroughly tested, a solution that basically distributes the sponge over all PEs can be advantageous. For this to work, each PE must have an updated table mapping MACs to IP addresses. This can be done in the switch's ARP table. Since the goal is to eliminate broadcast traffic from the core network, this table should be updated by the OpenFlow controller. Steps for this case are as follows:

1. The OpenFlow controller builds an IP-to-MAC table
2. The OpenFlow controller uses this table to update each PE's ARP cache
3. The switches can now respond to ARP requests based on their own ARP caches
4. The controller keeps the ARP caches up-to-date

As described in previous solutions, the OpenFlow controller could build up its own table; after this step, it will send gratuitous ARP messages for each node in its table to each PE. In this way, all the PEs will have these entries in their ARP caches. The controller should make sure that these are updated before they time out.

When a client would send an ARP request, the switch will have all the information necessary to answer it itself, on behalf of the requested node. As with other proposed solutions, if the client requests an unknown IP (not in the table), the request can either be answered with the device's MAC or sent to the controller. The controller can then either query that IP to see if it exists, or reply with its own MAC address. With this implementation, broadcast ARP is contained between the CE and the PE.

The most important fact to mention regarding this solution is that it will only be implementable if the switches have an ARP proxy feature. The reason for this is that the switches need to be able to answer requests based on their own ARP caches.

2.4 Solution 4: Forward to ARP sponge

This solution aims to make use of the ARP sponge. The motivation behind this solution is the advanced feature-set of the ARP sponge, which can be more easily extended. By forwarding the ARP requests to the sponge, they could be more accurately handled. To avoid overhead, the OpenFlow controller and the ARP sponge can be configured on the same physical machine.

The needed steps in this solution are as follows:

1. The controller installs a flow in the switch to forward every incoming ARP request to the ARP Sponge.
2. The ARP Sponge uses its internal ARP table to answer the ARP requests.
3. The ARP Sponge updates the table using the mentioned methods in section 1.3
4. The ARP Sponge monitors the network and answers the ARP requests destined for the dead nodes using its own MAC address

Since the current infrastructure uses the ARP Sponge, this solution can better integrate with the switching platform of AMS-IX , so it is less likely that this solution causes any undesirable effect on the network. However, because of the needed time for learning process in the ARP Sponge, we cannot completely prevent ARP broadcast(due to the learning period of the ARP sponge).

2.5 Solution 5: Forward to target router

Another solution can be using the ARP table from solution 1 and sending the ARP requests directly to the target node (instead of broadcasting). This is similar to the idea of unicast poll for ARP cache validation [1]. In unicast poll, to make sure that an entry in the ARP cache is still valid, a point-to-point ARP request is sent to the remote host. If after N attempts, no reply is received, the entry should be discarded from the ARP cache. `arping` (from the `iputils` package *) is an ARP level ping, which uses unicast ARP to reduce the amount of broadcast ARP traffic. `arping` starts with sending broadcast traffic and when a reply is received, it will switch to unicast ARP. Keeping this idea in mind, we can use the table from solution 1 and forward the ARP requests to the target. The needed steps in this solution are as follows:

1. The mapping table is imported to the controller
2. The controller installs a flow in the switch to forward the incoming ARP requests to the controller.
3. When the controller receives an ARP request, it looks up the IP address in the mapping table to find the corresponding MAC address.
4. If there is a match, the controller sends a unicast ARP request to the corresponding MAC address; otherwise, it makes an ARP reply using the MAC address of the controller.
5. If the controller receives a gratuitous ARP from one of the clients, the IP-to-MAC table is updated

Using this approach, the OpenFlow-enabled switch reduces the scope of ARP to simple unicast requests, thus lowering the broadcast traffic in the core. Moreover, the received ARP replies from the original nodes update the CAM tables in the PE switches.

2.6 Observations on the proposed solutions

When implementing either of the proposed solutions, consideration should be given to VLAN and VPLS configurations, as well as supported feature sets. Although VLAN tagging (per destination port) is available, support of MPLS label operations is added to OpenFlow from version 1.1 on. Since the current implementation of the Brocade MLX switches is based on OpenFlow 1.0, controlling MPLS operations using OpenFlow is not possible. If Brocade decides to support a newer version of OpenFlow, support of MPLS will be provided.

In a normal network where ARP would be broadcast, CAM tables on all switches would be constantly updated without any other interaction. However, since one of the aims of this project is to reduce and even eliminate broadcast traffic from the AMS-IX core, updating CAM tables becomes an issue.

In the current AMS-IX network, a lot of the peering is done using a route server. In short, a route server allows distribution of prefixes for its BGP peers. This eliminates the need of having a separate BGP session with each peer; instead, a session is needed only with the route server. However, because asymmetric load balancing is also used, there are cases in which traffic is unidirectional from one node to another. This means that the receiving node has some ports which do not have updated CAM tables.

A special tool developed at AMS-IX called `arpkeepalive` is already implemented, in order to solve this problem. It works by periodically sending ARP requests to each CE, telling it to reply to a forged, nonexistent, MAC address. This will cause all the CEs to reply, and all the switches will flood this message since they do not have the destination MAC in their CAM tables. The way in which `arpkeepalive` functions is shown in figure 2.1, in a topology with only three PEs with one customer each. Thus, all the CEs have updated their CAM tables. With this tool in place, CAM

[*] http://linux.die.net/man/8/arping

tables no longer pose a problem for our proposed OpenFlow solutions. Another option would be to implement the same functionality in the OpenFlow controller. Upon establishing a connection to each switch, it would periodically send the same forged ARP requests(with fake reply_to MAC address) to each client in order to update the PEs. One advantage of this option is that it does not depend on the ARP sponge like `arpkeepalive` does (it gets the list of customer IPs from a table generated by the sponge). This could potentially make it easier to update with new information.

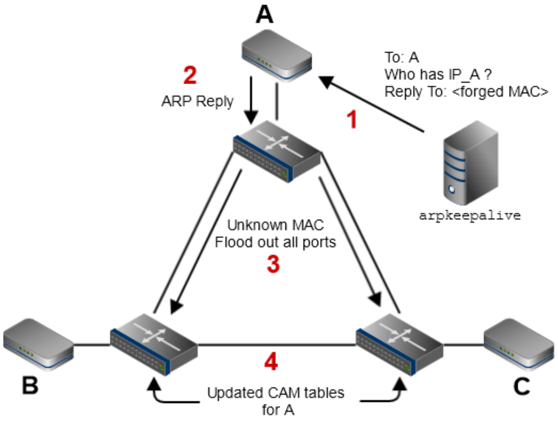


Figure 2.1: arpkeepalive example

In cases where one of the requested nodes is down, it is generally better to reply with the controller's MAC address, instead of that node's address. If a node is indeed down, its MAC address will timeout (60 seconds) from other CAM tables, leaving the node unreachable. If the node is unreachable but all other nodes believe it is still alive, they will continue sending traffic to it, causing (unknown) unicast storms. The best way to prevent this is to periodically check nodes for their state, and update the relevant tables.

Although all solutions depend on the controller, it is even more essential for solutions 1, 2 and 5 since they rely on it to reply to all the ARP requests. If the controller would go down, it would not pose an immediate problem in the case of solutions 3 and 4, because the flow table entry stays installed on the Brocade MLX even thought the connection to the controller is lost. The flow table entry will be removed/replaced/updated once the connection to the controller is back up. However, multiple controllers can be connected to each PE for redundancy.

From a security point of view, defense against MAC address spoofing attacks is an important concern in the AMS-IX platform. Among the proposed solutions, the ones that use the extracted IP-to-MAC mapping from the configuration files can better deal with MAC spoofing. They can use the mapping to validate the origins of the exchanged ARP messages in the network. So solution 1 and solution 3 can be used to mitigate the MAC spoofing attacks.

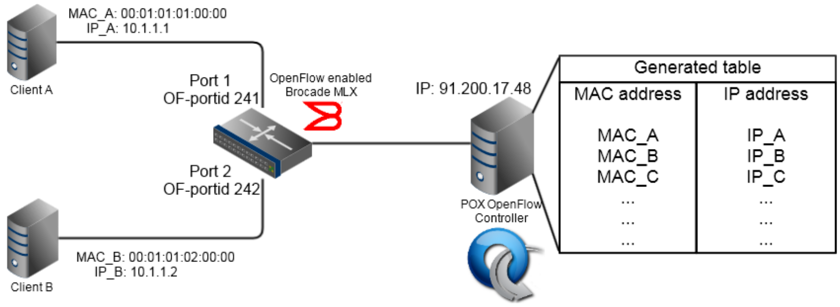
3 Proof of concept

3.1 Setup

The aim of the lab environment is to simulate a portion of the real network. Our experimental setup thus consists of 3 major elements: the clients, the OpenFlow enabled switch, and the OpenFlow controller.

We used two ports on an Anritsu MD1230B[*] traffic generator to simulate traffic from two communicating clients. Each port from the traffic generator connects to one switchport. The switch used was a Brocade MLX series, on which two 1 GbE ports were used. To benefit from the latest features of the switch (hybrid-port mode), we installed version 5.5b of the IronWare OS.

The switch has OpenFlow enabled on both ports, and connects to the OpenFlow controller. We used a Debian virtual machine on which we installed POX as the OpenFlow controller. The topology, and basic network configurations, are shown in figure 3.1.



Generated table	
MAC address	IP address
MAC_A	IP_A
MAC_B	IP_B
MAC_C	IP_C
...	...
...	...
...	...

Figure 3.1: Lab setup

POX is one of many available OpenFlow controllers. It is a Python-based SDN controller, and is part of the NOX project (C++ controller)[**]. We also made use of Mininet[†], a network emulator, for initial testing and debugging.

3.2 Implementation

We chose to implement solution 1 for the proof of concept because of its direct approach, and all the required features were already supported by the Brocade MLX switch. To implement the first proposed solution, we first had to gather all the MAC addresses and corresponding IP addresses of the entire network. AMS-IX does this by keeping an updated database with all the details of the connections (port ID, VLAN, MAC, IP etc.). This database is generated from an XML file, which has a tag for each

[*] http://www.anritsu.com/en-US/Products-Solutions/Products/MD1230B.aspx

[**] http://www.noxrepo.org/pox/about-pox/

[†] https://github.com/minine0t/mininet

such detail. An example for a VLAN definition, is shown in the listing below:

```
<vlan id="501" mode="untagged">
    <mac-address>782b.cb5a.bb68</mac-address>
        <router ipaddr="195.69.145.0" fqdn="rs2.ams-ix.net" asnum="6777">
        <attr id="route-server" value="1"/>
        <peering neighbor="3.14.159.2" />
        <peering neighbor="65.35.89.79"/>
        <peering neighbor="32.38.46.26"/>
        </router>
        <router ipaddr="2001:7F8:1::A500:6777:2" fqdn="rs2.ipv6.ams-ix.net" asnum
            ="6777">
        <attr id="route-server" value="1"/>
        <peering neighbor="2001:7F8:1::A500:1200:1"/>
        <peering neighbor="2001:7F8:1::A500:1200:2"/>
        </router>
</vlan>
```

The idea is to have a table, very similar to a regular ARP table, in which OpenFlow can look up MAC addresses by IP addresses, as shown in figure 3.1. In this way, when the controller receives the ARP request from a client, it can answer that request with the information from the table, and send back an ARP reply. We use a simple python script (Annex B) to generate this table based on the XML file. Although the XML file is updated multiple times per day, our proof of concept used it as a static source. The script we wrote can be extended to regularly check updates in the file, and generate a fresh table.

The OpenFlow controller then installs a flow entry in the OpenFlow table of the switch, matching all packets with the ARP EtherType 0x0806 and destination MAC address FF:FF:FF:FF:FF:FF. It only matches on the broadcast address in order to avoid sending unicast ARP(gratuitous ARP) as well. The action for this flow entry is "send to controller", meaning that any broadcast ARP messages will be sent to the controller. This flow entry is shown in listing 3.1.

Listing 3.1: Switch OpenFlow table entry

```
#sh openflow flows flowid 22586
Flow ID: 22586 Priority: 28672 Status: Active
        Rule:
           Destination Mac :      ffff.ffff.ffff
           Destination Mac Mask:           ffff.ffff.ffff
           Ether type:    0x00000806
        Action: FORWARD
           Out Port: send to controller
```

The controller then looks up the IPv4 address from the ARP request, gets the corresponding MAC address, and constructs an ARP reply that is sent to the client.

By configuring the OpenFlow enabled ports on the switch to work in hybrid-port mode* , any traffic that is not matched to the flow table rules is forwarded to the normal switching fabric. In this way, normal traffic is guaranteed to be unaffected by the OpenFlow rules. Figure 3.2 shows the steps carried out for each ingress packet on the OpenFlow enabled ports of the switch.

3.3 Observations

It is important to mention that the table only maps IPv4 addresses, since IPv6 uses Neighbor Discovery and is out of the scope of this project. Special MAC addresses for LACP (Link Aggregation Control Protocol) links are also listed in the file, but these are answered by the PE itself. For these links, we use the main MAC of the link for the table.

Another observation is the fact that some IPv4 addresses are mapped to two MAC addresses. AMS-IX customers sometimes change their MAC address (hardware changes for example). Because each switch filters ingress traffic based on MAC address, when a customer changes his address, it has to be updated in the access-list on the switch. This means that it also has to be updated in the XML file, so that traffic still goes

*http://community.brocade.com/community/discuss/sdn/blog/2013/04/19/the-practical-path-to-sdn-brocade-openflow-hybrid-port-mode

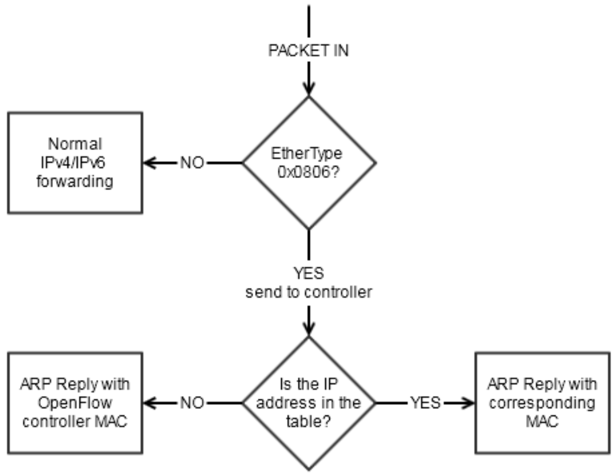

PACKET IN
EtherType 0x0806?
NO
Normal IPv4/IPv6 forwarding
YES send to controller
Is the IP address in the table?
NO
ARP Reply with OpenFlow controller MAC
YES
ARP Reply with corresponding MAC

Figure 3.2: ARP processing flowchart

through while the change is happening. This can pose a problem, because the OpenFlow controller needs to be aware of what MAC is in use. At the time of this writing, the XML lists both the old and the new MAC addresses. The script that we used in this setup just uses the last MAC address introduced in the file, and assumes it is the one currently in use.

Tests were also done using multiple controllers running on the same VM, but on different ports. One controller used port 6633 and the other used port 6634; both controllers were configured in active mode on the switch. In an active connection configuration, the switch will seek the controller and attempt to connect to it; in a passive configuration, it is the controller that seeks switches in order to connect to them. In this setup, the controller which was the last to connect was the one that pushed the flow into the OpenFlow table of the switch.

Although there is no mention of Link Aggregation Group (LAG) support for OpenFlow in the Brocade documentation, we have tested a scenario in which the client is connected through a LAG to the switch. In this scenario, packets coming in through the LAG are matched accordingly and there were no errors. However, two other cases remain to be tested:

- when the LAG is between the controller and the switch
- when the output port for a table entry is a LAG

OpenFlow provides documentation and support for LAGs only in later versions than OpenFlow v1.0.0. The current Brocade implementation (OpenFlow v1.0.0) does support forwarding to a set of ports. This would make the setup less dynamic because the script would have to be changed every time a port would be removed/added to the LAG. Referencing the LAG by an ID would make the controller independent of the LAGs port set and configuration.

4 Conclusion

The main conclusion of this research project is that, indeed, OpenFlow can be used to reduce ARP traffic in the layer 2 ISP peering LAN of AMS-IX. We have presented five possible solutions that can be implemented to reach this goal. The proposed solutions explore different options and allow for a degree of flexibility depending on the latest features supported by the Brocade switches and resources available (multiple controllers, the ARP sponge).

The ARP protocol can still be used for a short learning period or in unicast mode. In either situation, the amount of ARP traffic is much lower than in the current setup. Thus, the load on the CPUs of the network devices is brought down significantly. The ARP protocol still functions normally between the CE and the PE nodes of the network, but the effects of broadcasting are not seen on the network any more.

OpenFlow represents a flexible and scalable solution to the problem of ARP broadcast traffic. Because OpenFlow interacts so deeply with the hardware of the devices, matching and forwarding is done at line speed, having no side-effects on bandwidth. Moreover, with the recent implementation of hybrid-port mode on the Brocade MLX, normal traffic can be easily isolated from OpenFlow rules.

Multiple OpenFlow controllers can be connected redundantly in order to account for failures of the controller. Even in the case in which a single controller is used, the flow table rules remain loaded in the switch until another connection to a controller is set.

5 Acknowledgements

This project could not have been made possible without the help of some people. We would like to thank our coordinator, Martin Pels, for his guidance, patience, and support. Joerg Ammon was of special help as well, giving us a lot of support and insight into Brocade's latest developments regarding their OpenFlow implementation and future plans. The AMS-IX team as a whole was very supportive of our project and provided useful feedback that helped us constantly improve our project (and our minigolf skills).

Bibliography

[1] Braden, R. 1989. *RFC 1122: Requirements for Internet Hosts – Communication Layers*.

[2] K. Kompella, M. Lasserre. 2007a. *RFC 4762: Virtual Private LAN Service (VPLS) Using Label Distribution Protocol (LDP) Signaling*.

[3] K. Kompella, Y. Rekhter. 2007b. *RFC 4761: Virtual Private LAN Service (VPLS) Using BGP for Auto-Discovery and Signaling*.

[4] Marco Wessel, Niels Sijm. 2009. *Effects of IPv4 and IPv6 address resolution on AMS-IX and the ARP Sponge*. M.Phil. thesis, Universiteit van Amsterdam.

[5] Nick McKeown, Tom Anderson, Hari Balakrishnan Guru Parulkar Larry Peterson Jennifer Rexford Scott Shenker Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, **38**(2), 69–74.

[6] Pepelnjak, Ivan. 2013. *Could IXPs Use OpenFlow To Scale?* The Middle East Network Operators Group (MENOG) 12.

A Abbreviations

AMS-IX Amsterdam Internet Exchange

ARP Address Resolution Protocol

BGP Border Gateway Protocol

CAM Content Addressable Memory

CPU Central Processing Unit

DHCP Dynamic Host Configuration Protocol

GE Gigabit Ethernet

ISP Internet Service Provider

IP Internet Protocol

LAG Link Aggregation Group

LAN Local Area Network

MAC Media Access Control

MPLS MultiProtocol Label Switching

- **P** Provider router
- **PE** Provider Edge router
- **CE** Customer Edge router
- **LDP** Label Distribution Protocol
- **LSP** Label Switched Path

NDP Neighbor Discovery Protocol

PXC Photonic Cross-Connect

RSVP-TE Resource Reservation Protocol - Traffic Engineering

SDN Software Defined Networking

VLAN Virtual LAN

VPLS Virtual Private LAN Service

B Code listings

Script for extracting MAC and IP addresses from an XML file, and creating a new file with the MAC-IP table.

```
from BeautifulSoup import BeautifulSoup
import re
import subprocess

x = open('aggregates.xml')
y = BeautifulSoup(x)

all_agg =  y.database.findAll("aggregate")

emptyTable = "touch mac_table.txt"
process = subprocess.Popen(emptyTable.split(), stdout=subprocess.PIPE)
output = process.communicate()[0]

mac_table = open("test_mac_table.txt","w")

for i in range( 0, len(all_agg)):

 # if it actually has a VLAN entry, and thus a mac-address tag
 if y.database.findAll("aggregate")[i].vlan and y.database.findAll("aggregate")[i].vlan.router:

  # filter out those annoying LACP MACs
  if not y.database.findAll("aggregate")[i].vlan.find(type="lacp"):

   # list all MACs from that aggregate
   for j in range(0, len(y.database.findAll("aggregate")[i].vlan.findAll("mac-address"))):
    var1 = y.database.findAll("aggregate")[i].vlan.findAll("mac-address")[j].string
    var2 = var1.split('.')
    var3 = ''.join(var2)
    var4 = re.findall('..',var3)
    final = ':'.join(var4)
    # filter out IPv6 addresses
    if len(y.database.findAll("aggregate")[i].vlan.router['ipaddr']) <= 15:
        mac_table.write(final + " " + y.database.findAll("aggregate")[i].vlan.router['ipaddr'] + "\n")

mac_table.close()
```

OpenFlow controller script, based on the original arp_responder.py script from the POX installation.

```
# Copyright 2011,2012 James McCauley
#
# This file is part of POX.
#
# POX is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# POX is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with POX.  If not, see <http://www.gnu.org/licenses/>.

from pox.core import core
import pox
log = core.getLogger()

from pox.lib.packet.ethernet import ethernet, ETHER_BROADCAST
from pox.lib.packet.arp import arp
from pox.lib.addresses import IPAddr, EthAddr
from pox.lib.util import dpid_to_str, str_to_bool
from pox.lib.recoco import Timer
from pox.lib.revent import EventHalt

from xml.dom import minidom
from BeautifulSoup import BeautifulSoup

import pox.openflow.libopenflow_01 as of

import time


class Entry (object):
  def __init__ (self, mac, static = False):
#     self.timeout = time.time() + ARP_TIMEOUT
#     self.static = static
    if mac is True:
      # Means use switch's MAC, implies True
      self.mac = True
      self.static = True
    else:
      self.mac = EthAddr(mac)

  def __eq__ (self, other):
    if isinstance(other, Entry):
      return (self.static,self.mac)==(other.static,other.mac)
    else:
      return self.mac == other
  def __ne__ (self, other):
    return not self.__eq__(other)


class ARPTable (dict):
  def __repr__ (self):
    o = []
    for k,e in self.iteritems():
      if e.static: t = "-"
      mac = e.mac
      if mac is True: mac = "<Switch MAC>"
      o.append((k,"%-17s %-20s %3s" % (k, mac, t)))

    o.sort()
    o = [e[1] for e in o]
    o.insert(0,"-- ARP Table -----")
    if len(o) == 1:
      o.append("<< Empty >>")
    return "\n".join(o)

  def __setitem__ (self, key, val):
    key = IPAddr(key)
    if not isinstance(val, Entry):
      val = Entry(val)
    dict.__setitem__(self, key, val)

  def __delitem__ (self, key):
    key = IPAddr(key)
    dict.__delitem__(self, key)

  def set (self, key, value=True, static=True):
    if not isinstance(value, Entry):
      value = Entry(value, static=static)
    self[key] = value


def _dpid_to_mac (dpid):
  # Should maybe look at internal port MAC instead?
  return EthAddr("%012x" % (dpid & 0xffFFffFFffFF,))


class ARPResponder (object):
  def __init__ (self):

    core.addListeners(self)

  def _handle_GoingUpEvent (self, event):
    core.openflow.addListeners(self)
    log.debug("Up...")

  def _handle_ConnectionUp (self, event):
    if _install_flow:
      fm = of.ofp_flow_mod()
      fm.priority = 0x7000 # Pretty high
      fm.match.dl_type = ethernet.ARP_TYPE
      fm.actions.append(of.ofp_action_output(port=of.OFPP_CONTROLLER))
      event.connection.send(fm)

  def _handle_PacketIn (self, event):
    squelch = False

    dpid = event.connection.dpid
    inport = event.port
    packet = event.parsed
    if not packet.parsed:
      log.warning("%s: ignoring unparsed packet", dpid_to_str(dpid))
      return

    a = packet.find('arp')
    if not a: return

    log.debug("%s ARP %s %s => %s", dpid_to_str(dpid),
      {arp.REQUEST:"request",arp.REPLY:"reply"}.get(a.opcode,
      'op:%i' % (a.opcode,)), str(a.protosrc), str(a.protodst))

    if a.prototype == arp.PROTO_TYPE_IP:
      if a.hwtype == arp.HW_TYPE_ETHERNET:
        if a.protosrc != 0:

          if _learn:
             log.info("%s learned %s", dpid_to_str(dpid), a.protosrc)
             _arp_table[a.protosrc] = Entry(a.hwsrc)

          if a.opcode == arp.REQUEST:
            # Maybe we can answer

            if a.protodst in _arp_table:
              # We have an answer...

              r = arp()
              r.hwtype = a.hwtype
              r.prototype = a.prototype
              r.hwlen = a.hwlen
              r.protolen = a.protolen
              r.opcode = arp.REPLY
              r.hwdst = a.hwsrc
              r.protodst = a.protosrc
              r.protosrc = a.protodst
              mac = _arp_table[a.protodst].mac
              if mac is True:
                # Special case -- use ourself
                mac = _dpid_to_mac(dpid)
              r.hwsrc = mac
              e = ethernet(type=packet.type, src=_dpid_to_mac(dpid),
                           dst=a.hwsrc)
              e.payload = r
              log.info("%s answering ARP for %s" % (dpid_to_str(dpid),
                str(r.protosrc)))
              msg = of.ofp_packet_out()
              msg.data = e.pack()
              msg.actions.append(of.ofp_action_output(port =
                                                      of.OFPP_IN_PORT))
              msg.in_port = inport
              event.connection.send(msg)
              return EventHalt if _eat_packets else None

    # Didn't know how to handle this ARP, so just flood it
    msg = "%s flooding ARP %s %s => %s" % (dpid_to_str(dpid),
        {arp.REQUEST:"request",arp.REPLY:"reply"}.get(a.opcode,
        'op:%i' % (a.opcode,)), a.protosrc, a.protodst)

    if squelch:
      log.debug(msg)
    else:
      log.info(msg)

    msg = of.ofp_packet_out()
    msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
    msg.data = event.ofp
    event.connection.send(msg.pack())
    return EventHalt if _eat_packets else None


_arp_table = ARPTable() # IPAddr -> Entry
_install_flow = None
_eat_packets = None
_learn = None

def launch (no_flow=False, eat_packets=True,
            no_learn=True, **kw):
  global  _install_flow, _eat_packets, _learn
  _install_flow = not no_flow
  _eat_packets = str_to_bool(eat_packets)
  _learn = not no_learn
  file = open('mac_table_final.txt')

  core.Interactive.variables['arp'] = _arp_table
  for k,v in kw.iteritems():
    _arp_table[IPAddr(k)] = Entry(v, static=True)
    for line in file.readlines():
```

```
    line = line.strip()
  #print line
  col1,col2 = line.split()
  print col1 , col2
  _arp_table[IPAddr(col2)] = Entry(EthAddr(col1), static=True)
  core.registerNew(ARPResponder)
```