A Dynamic Stateful Multicast Firewall

By
Bewketu Tadilo Shibabaw

Supervisor: Dr. Vijay Sivaraman (Room 334)

A thesis submitted for the degree of
Bachelor of Engineering

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

Electrical and Telecommunications Engineering,
The University of New South Wales.

May 2009

Abstract

Security has been one major hindrance to the spread of multicasting and its use. Network managers are still rightly paranoid enabling the technology when a whole array of internal resources can be at risk. Efforts have been made to alleviate some of the concern in the form modified service model namely source specific multicasting. The attractiveness of the original flat service model still remains at large and so do the multitude of routers and hosts that only support this traditional version. The newer protocols have been made backward compatible for this reason. This thesis aims to create a security layer that can sit in between these two models. It draws upon experiences learned from the design of stateful unicast firewalls and the newer protocols to track connections and associations to form its rules of a stateful multicast firewall. It enables both coexistence and smoother transition of these two service models.

Originality Statement

I hereby declare that this submission is the work of my own. This work has not been submitted previously for any other degree and to the best of my knowledge, does not contain materials previously published or written by another person except where due acknowledgment is made.

Signed:_____

Dedicated to my European and Australian friends whom I met in Gondar, Ethiopia. Thank you for taking an extra burden to sponsor my studies in Australia and your collective parenting. May God bless you all!

–Bewketu Tadilo, June 2009.

Abbreviations

DoS/DDoS: Denial of Service/Distrubuted denial-of-service 4

ASM: Any Source Multicasting 13

SSM: Source Specific Multicasting 13

IGMP: Internet Group Management Protocol 6

DNS: Domain Name Server 13

MLD: Multicast Listener Discovery 6

PIM-SM: Protocol Independent Multicasting- Sparse Mode 11

IETF: Internet Engineering Task Force 1

IANA: The Internet Assigned Numbers Authority 3

MSEC WG: Multicast Security Working Group 12

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Multicasting has been supported by IP networks for two decades since Steve Deering did his PhD on it, and wrote the Internet Engineering Task Force (IETF)standard at the end of 1989[**2**]. But, the technology have been little used up until recently. As can be inferred from its name, *multicasting* is a technique for transmitting data from a single source to only *a set of recipeints*. The very idea brings concepts that are not encountered in *broadcasting*, in which a message is sent to *all* computers on the network, and *unicasting* in which a message is passed between a single source and receiver. One of them is how we manage these set of recipients and is discussed in section 2.2 on page 6.

Advances in Audiovisual technologies, that naturally demand high bandwidth, have spurred the recent interest in multicasting in Small and Medium Enterprises (SMEs) and consumer services. This is because we can significantly reduce traffic load if we use multicast efficiently as we will see in section 1.1 on the next page. Nevertheless, it has remained elusive to many Network Managers, Software Developers, System Admins and end users who might otherwise benefit from this technology. Multicasting is heavily used in the stoke market where its model of updating all interested users at the same time fits in perfectly. The defence departments of countries are also big investors to the advancement the technology for better coordination in battlefields. These are areas among others in which multicasting can/is being used

- Clustering
  - load sharing
  - database replication
  - failover clusters
- IPTV and live Internet Audio (radio)
- Interactive gaming
- Software distribution.

It even comes in very handy, if a Network Manager wants to install one program to set of computers on his/her network, including operating systems.

We focus this thesis with medium enterprises in our mind and from a perspective of the Network Manager for that Enterprise. We are on Layer 3 (at the network layer) in the entire thesis.

Multicasting reduces to either unicasting and broadcasting depending on whether *one or all* members of the network are participating in communication respectively.

The use of broadcasting in many cases has been discouraged in favour of multicasting as it is prone to wasting huge resource.

1.1. Why Multicasting?

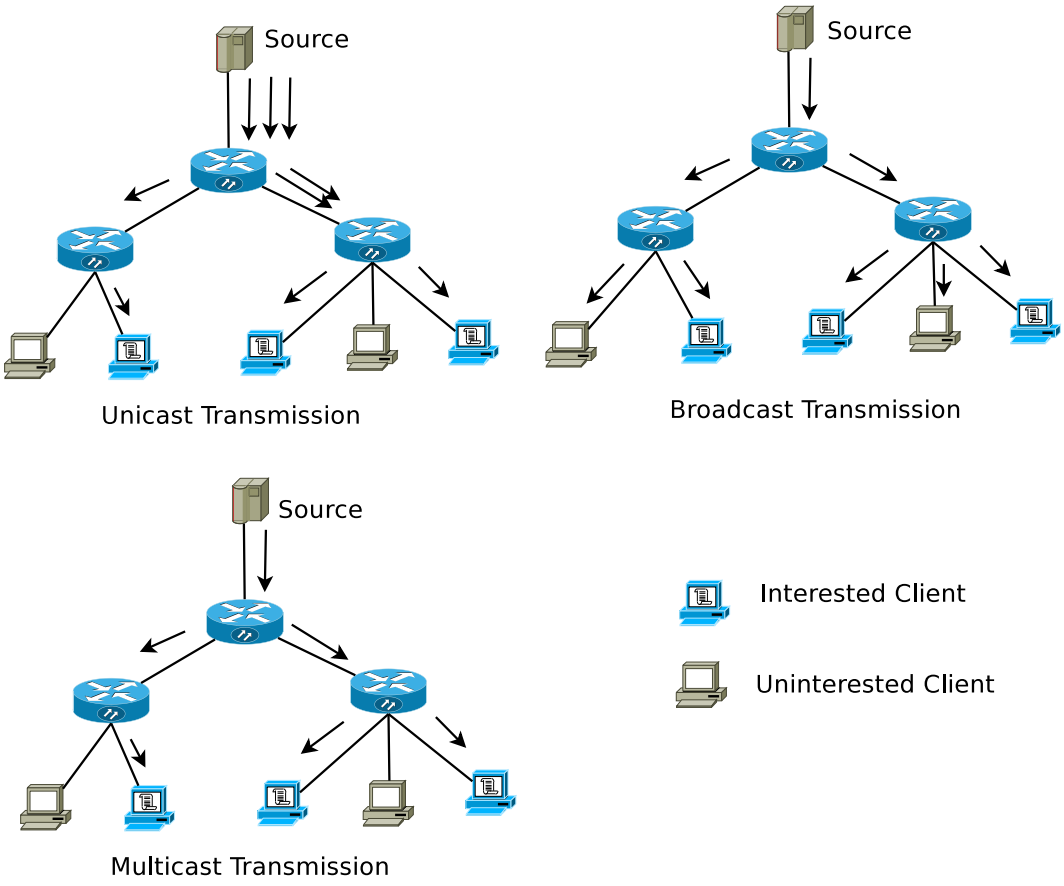


FIGURE 1.1. Data delivery methods

We refer to Figure 1.1 . The source generates a data steam with throughput of 1Mbps, and the data stream is to be received by three recipients. On the upper left-hand side we see a unicast transmission where the source sends three instances of the same data to the receivers. The network link has to carry the same data stream multiplied by three, and so 3Mbps of bandwidth easily consumed. Whereas, the one on the bottom, which is the essence of multicasting, the data is replicated only when it needs to at the junction points. That, in fact, is one of the performance measures of multicasting routing protocols. We can tell from figure 1.1, it saves 2Mbps for the up until the fan-out point and saves another 1Mbps for the network on the rightmost side. It decouples the link bandwidth dependence on the number of hosts wanting that traffic. The disadvantage of broadcasting[1] in relation can

[1]This has caused the creation of violent terms like packet flooding and broadcast storm.

inferred from Figure 1.1- which is uninterested hosts and routers get copy of the data. Though we do not want to proceed with the point- in multicast routing, it is desirable effect that the data travels long distance with the least "fan-out" possible thus saving a lot of bandwidth along its way to its final destination.

1.2. What is Multicast Firewall?

Well it is all good, we can save all the bandwidth with scale of $1/n$ where n is the number of unicast computers in the network that have to receive the traffic. The notion of unicast firewalling scheme breaks down when we are dealing with multicast traffic. In this thesis, when we say firewall[2] what we are referring is the filtering type, which drops or accepts traffic, based on packet header information[3]. A general golden rule for firewall that maintain state, known as stateful firewall, is to *trust an outsider that is contacted by an insider*. Keeping that in mind, we now deal with firewalling multicast. Firewall golden rule

A multicast source is a host that sends packets with the destination IP address set to a multicast group. Anyone can send traffic to that group! A source need not to be a member of the group; sourcing and listening are mutually exclusive. The reason why the firewall golden rule no longer works is the destination address is a group address which is in some sense *virtual*. How can we establish which *actual* outsider our client is trying to contact?

To compound the problem these addresses usually are *volatile*. There have been/are efforts by The Internet Assigned Numbers Authority (IANA) and various standard bodies to get some order on the address allocations and how ISPs should manage these addresses, should be used and various recommendations. For this thesis, we are better off to assume that these addresses as mentioned above are not static. We are also going to be dealing with *access list* control exclusively. The authentication mechanism of sources is mainly the firewall golden rule.

So, let's outline why a firewall maybe needed in controlling access list (whose traffic is authorized to come in) in multicasting:

- To protect the whole internal network from unwanted traffic
- To protect client (insider) group members from unwanted traffic (individual tailored) from outside or within the network and
- Possibly, preventing an unauthorized user within or outside the network from accessing our traffic.

The list is not exhaustive but helps to flesh out what we want achieve. The first two are for network *mantainablity* while the last one is question of *privacy*. Access list is non-trivial especially if dealing with insiders. Some insecurities in multicasting are unavoidable from its designs that ironically make it attractive too. We discuss the last two points more in Chapter 6.

[2]We assume the marketing term firewall, as anti-worm or anti-virus one acquires when they go to sites they should not go to, is nonsensical and not applicable here.

[3]This is in contrast with what are know as application gateway where the actual content is examined via data mining of some sort.

1.3. Problem Statement

From a Network Manager's perspective the biggest worry is Denial of Service/Distrubuted denial-of-service (DoS/DDoS)attack on the network as multicasting amplifies it because multiple resources can be members- (e.g an outsider dumps traffic on the stock exchange network). All innocent group members become victim as opposed to unicast where only one user[4] is affected. To defend that, the usual modus operandi in SMEs is for the Network Manager to simply block all incoming multicast traffic. Then, if someone must join a multicast group, the Network Manager allows the unique source and multicast address manually and traffic starts to come in from that particular source. Now one can imagine what could be the bureaucratic process to achieve that.

The problem statement is : *To design a multicast firewall that relieves the Network Manager from entering static rules per session.*

Being rather prescriptive, our design solution should satisfy the following requirements in order to relieve the Network Manager from doing that:

(1) Block all incoming multicast traffic
(2) Inspect unicast traffic that come out from clients inside our network- (these usually have distinct ports)- to track which unicast source addresses our clients are contacting
(3) Inspect all multicast group subscription of our clients
(4) Allow incoming multicast group traffic from sources
    (a) That have associations with clients from step 2.
    (b) Those clients are also subscribed that group in step 3.

2-4 are restatements of the paper written on this same topic by Vijay et al[**8**]. To elaborate on step 2 a bit, it can, for example, be watching over all TCP connections on Port 80. That is when a user is simply browsing the web. We say more on this on later parts.

1.4. Contribution

Contribution of this thesis involves:

- An algorithm that avoids *as many* memory allocations as possible.
- A data structure that help models *all* entities involved and algorithms along with it
- The overall design

The first contribution needs acknowledgement. My supervisor, Vijay Sivaraman, indicated this is area of bottleneck from previous student implementation. This is to early to discuss more contribution but hopefully we are able to show in Chapter 4 during discussion of detailed design.

[4]We say user for autonomous system, humans as well as host that are being used by humans.

1.5. Document Scope

Questions we address.

- What kind of security model multicast entails and thus the paradigm we follow?
- What is the design solution and its implementation?
- What is the prototype design environment?
- The comparison of the design solution with other implementations.
- How is the performance with the tests conducted so far?

We are quite careful to avoid discussion of multicast routing in most cases. It is quite involved on its own and surely this thesis will only address such issues which we can exploit in the design of our firewall. Often, what we want is to *extract the unicast address of the multicast source efficiently*- an address one of our clients is trying to contact.

Another key fact that one has to know when working on the network stack is it is can never be decoupled how the underlying Operating System handles IP. It requires an intimate knowledge of operating systems networking stack.

Lastly on light tone, I will jestly tell owner of a computer with 32-bit x86 single processor as very lucky person unlike me who owns lowest end of 'fancy' multicore. A newer issue has propped with the advancement of multi-core operating systems too. We have to model our data and our programs to achieve performance taking the fact that there is more than one processor on the computer. Oh before that though, avoiding deadlocks, bus errors and racing are our challenges. This entails questions like do processors share the same memory address space, the same bus and so on? In fact, the author has spent more time dealing with the two above issues than the logical design of this firewall. He had two manually reboot his computer one two many times on segmentation faults that result in computer hangs . We will not discuss those topics in this thesis as they can be digressions.

CHAPTER 2

Background: Protocols, Firewalls and Related works

The purpose of this chapter is to review some prerequisite material in the design of our firewall. We try to filter in only those that are required for dispensing the idea our firewall. These are mainly IP multicast networking concepts. Implementation wise though, we also need to have a firm grasp of the prototyping environment, the encapsulating framework (for our case, these are Linux kernel and Netfilter respectively) and how the life of a packet evolves in those environments. We try to minimize, at least defer, discussion of these latter issues as they are tied to implementation. We focus mainly on IPv4 as our implemented module works on it and the key ideas can be fully elaborated using that version of the IP. We try to shed some light at multicast enabler protocols with special emphasis given to the ones we need for this thesis most. It also introduces us to 'related' work. The discussion of IPv6 diversion and design challenged is a great omission. But we give some pointers to equivalent protocols-like IGMP for Multicast Listener Discovery (MLD) - as we are about to see.

2.1. Protocols in Multicasting

When we talk about a *working* multicasting system, a failure to mention the combination of the enabling protocols involved leaves a big gap of information. Usually, Network Engineers put these separated in hyphen or slash in triplet or so (e.g MSDP/MBGP/PIM in IPv4). This is only to stress multicasting is only a model- not actual technology. So is unicasting but it perfectly maps to IP without any other protocol (needless to say, IP was designed with unicasting model in mind).

2.2. Group Management Protocols

One of the concepts we referred in chapter 1, is this notion of *predefined set of hosts*. Internet Group Management Protocol (IGMP), as the name suggests, is the protocol that helps the *group manager*,which is usually the nearest router, manipulate the group members. A prospect member host must be able to construct IGMP messages if its wants to participate in a multicast group. A Multicast manager, on the other hand, must also be able to *manage* the list. IGMP is encapsulated in IP datagram by setting the protocol field to IPPROTO_IGMP[1] - which is a constant 2. The following steps are involved while managing a group.

MLD is IPv6 counterpart of IGMP.

[1]We use constants like this taken from their RFCs (these are usually in #define codes too)

- The router asks every period (of about 125 seconds) if anyone wants *any* multicast message. By default, every multicast capable device must *always* be subscribed to one address to receive this. This is, for example, 224.0.0.1 in IPv4. Router IGMP Query
- Hosts that want to join a group or keep their membership alive reply (report) with the group address they are still participating or want to start participating. Host IGMP Report
- If the router receives no interest from at least a single host to be in a group, that group is deleted (assuming it exists) and the router notifies his upper peer (if inter-domain multicasting) that it is no longer interested in that group.

The list outlines the baseline procedure that makes this group management *work*. This was exactly what was implemented in version 1 of IGMP. Further on the next iterations various optimization were added which are discussed below. The latest IGMP version number is 3. Version 0 is mere history.

We should take note from first point that a group's age is about 2 minutes which we exploit on later in our Implementation.

2.2.1. The law of backward compatibility. The IETF standards explicitly require the group management 'language' be the lowest IGMP version among members if at least one member has that version as its upper limit support. This entails for the greater versions of IGMP to be backward compatible with their predecessors.

2.2.2. IGMP (v1 and v2). We discuss both IGMP[**3**] versions for comparisons. Version 2 introduces 'Explicit Leave group' message by the host when it no longer wants to receive traffic. This is to decrease link load by issuing a leave message than waiting for the group membership timer to expire; which is followed by the router asking if any one wants to receive traffic. The 'leave latency', the time between when a user stops wanting to receive traffic and the router knows that, will decrease. It is an optimization on the version 1 which simply 'shuts up' when it no longer wants to receive traffic. Thus, the discussion IGMPv2 includes both. We are mainly concerned with report messages of clients as that is where we extract the group a host wants to join. The following is header information of IGMPv2[2]: Host IGMP Leave

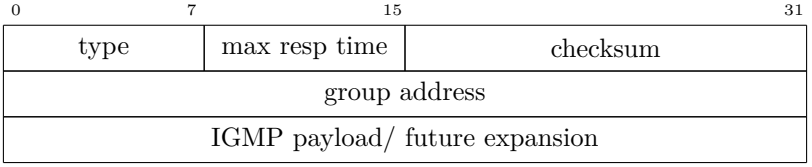


FIGURE 2.1. IGMPv1/v2 Message Format

[2]For least distraction, the encapsulating IP datagram is hidden but it is just above.

The fact is we don't even need to dissect the IGMP header to extract the group it is destined for. It can be found in the IP datagram header. That changes for leave messages; where the destination address of the IP header is set to 224.0.0.2 (the all-routers group). The following is some information about the fields:

- The type describes what kind of IGMP message it is. From it we know if it router query, host leave message or host membership report. We are often concerned with IGMPV{x}_HOST_MEMBERSHIP_REPORT.
- The max resp time is heuristic based constant, which tells the host about when to respond with a report after a router sent query to all hosts. It is about 1 second and can be set to other values in IGMPv2 or later.

2.2.3. IGMPv3. As can be seen on Figure 2.2 [**1**]this version have a radical change from the previous in that the header did change dramatically. It also changes the service model of traditional multicasting by coupling unicasting as the underlying framework. A new idea of 'the user should know where the multicast source is' introduces that one can add:

MLDv2 is the counterpart for IGMPv3 in IPv6

- *Only* those unicast source addresses that one want to hear from. (Include mode)
- All *except* those source addresses that one want hear from. (Exclude mode)

The sudden complexity and bloatness of this design is worth as it is proven by experience it works for one type of model (one to many multicasting). We discuss in detail multicasting models in Section 2.4.

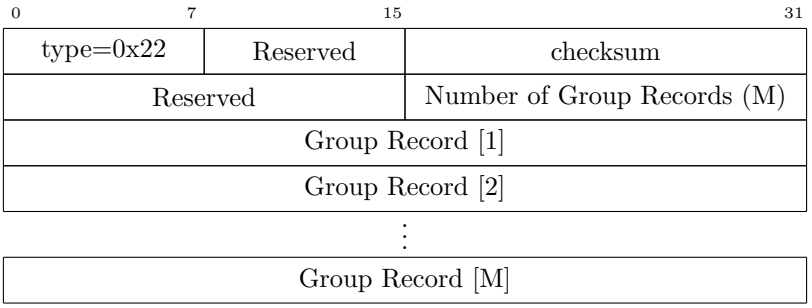


FIGURE 2.2. IGMP Version 3 Membership Report Message

The group record further decomposes to the following. We find the list of sources we want to include/exclude per multicast address.

One of the great things about this protocol is the fact it has breathed fresh air to otherwise sleeping multicasting usage in the industry. It especially works great if we just have a few multicast sources per group. We want to see it from the angle of our problem though. Now, it has the capacity to block static addresses when used in *exclude & include* mode. Yes sure, one can use it to block an annoying user 'on the fly' participating in the multicast group using exclude mode (for example, quite handy when a user is playing Multiplayer online games and wants to block

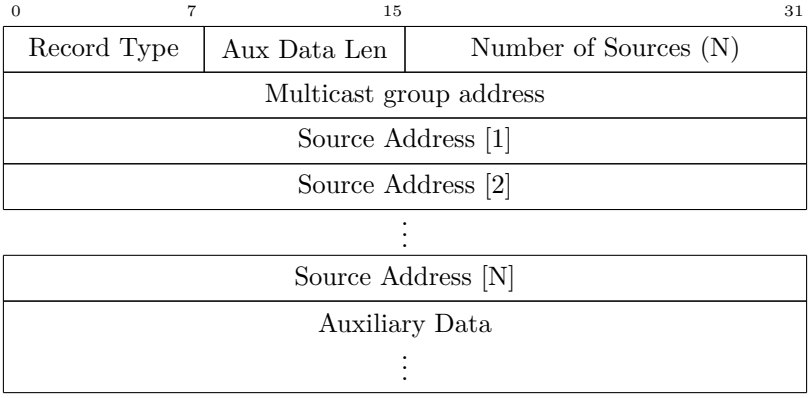


FIGURE 2.3. IGMP Version 3 Group Record Internal Format

someone). In the end, we can only list so much unicast addresses in static form that it cannot be of practical usage for protection against DoS/DDoS attacks when all members are both sources and receivers- the traditional model. Essentially, the verdict is that it is quite a capable protocol if we want a one to many dissemination type communication.[3] But, multicasting has two other extra paradigms-many to one and many to many- and as such it solves 1/3rd of the problem.

There is also argument against it in routing inefficiency when the number of sources increase. The router could easily run out of memory if it is going to keep lots of state information per client. The data path distribution tree is also sub-optimal. In this thesis though, we do not pursue these as our arguments for our design. IGMPv3 is still considered fairly new (in 2009) and we expect to have hosts and routers that only work with the earlier versions. As an example, Linux kernel networking stack integrated it in 2002 and FreeBSD[4] still has not integrated it. Even if it is integrated, more often than not it is working with compatibility mode we discussed above. This adds some weight as why we should pursue such endeavor.

A salient point in relation to backward compatibility is an insider can expose the network for DoS/DDoS attack by reporting using lower IGMP versions. This is because the language of that group is turned into one of the earlier versions and there is no longer source filtering in place.

2.3. A Word About Multicast Routing Protocols

It seems instructional to say a bit about multicast routing as some of the protocols and service models that we are about to discuss have direct relation to them and lest the document may become incoherent. But, the filter module can be done away without much information on the routing protocols.

[3]'radio' style communication where a subscriber/tuner can only receive.
[4]Hard to claim but a little research on the kernel mailing list will show.
http://lists.freebsd.org/pipermail/freebsd-current/2009-March/004128.html

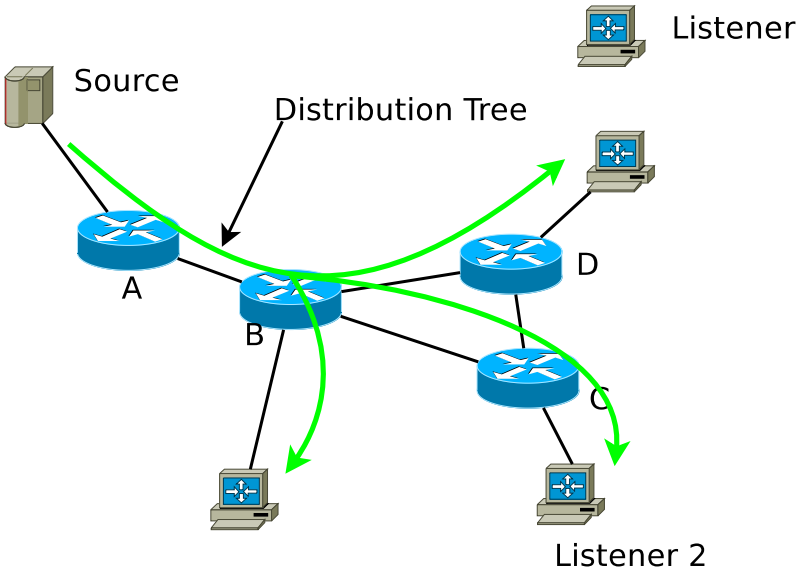


FIGURE 2.4. Multicast Distribution Tree

Because there can be multiple receivers, the traffic path may have several branches as we see in Figure 2.4. The whole data path from start the source to receivers is known as a distribution tree. Data flow through the distribution trees is referred as either upstream and downstream. Downstream is in the direction toward the receivers. Upstream is in the direction toward the source. A downstream interface is an outgoing or outbound interface; and an upstream interface is as an incoming or inbound interface. This is all relative to source.

But let's see how this distribution tree is formed following steps in Figure 2.5. The following steps will take place.

(1) The source sends data packet to specific multicast group. Nobody is yet interested; router A simply discards it.
(2) Listener 2 sends IGMP multicast report to router C.
(3) Router C sends a join (this is done with the routing protocol) on behalf of Listener 2 to router B. But how did it know to ask B, not D? The concept of Rendez-vous point-root router- of the distribution tree comes here. Each multicast capable router (either dynamically or statically) must know the root of the tree. We assumed the source is the root here- which SSM also assumes as we will see soon. Rendez-vous point
(4) Router B to A as in (3) and then the route ABC is established.

As further elaboration of (3) if D was the Rendez-vous point configured in all routers. Source contacts A, A contacts B and, B contacts D and notifies it source is multicasting. Then, listener contacts C, C contacts D. A path $ABDC$ would have been established. This path could be optimal in cases where there maybe another source but we leave that for the moment.

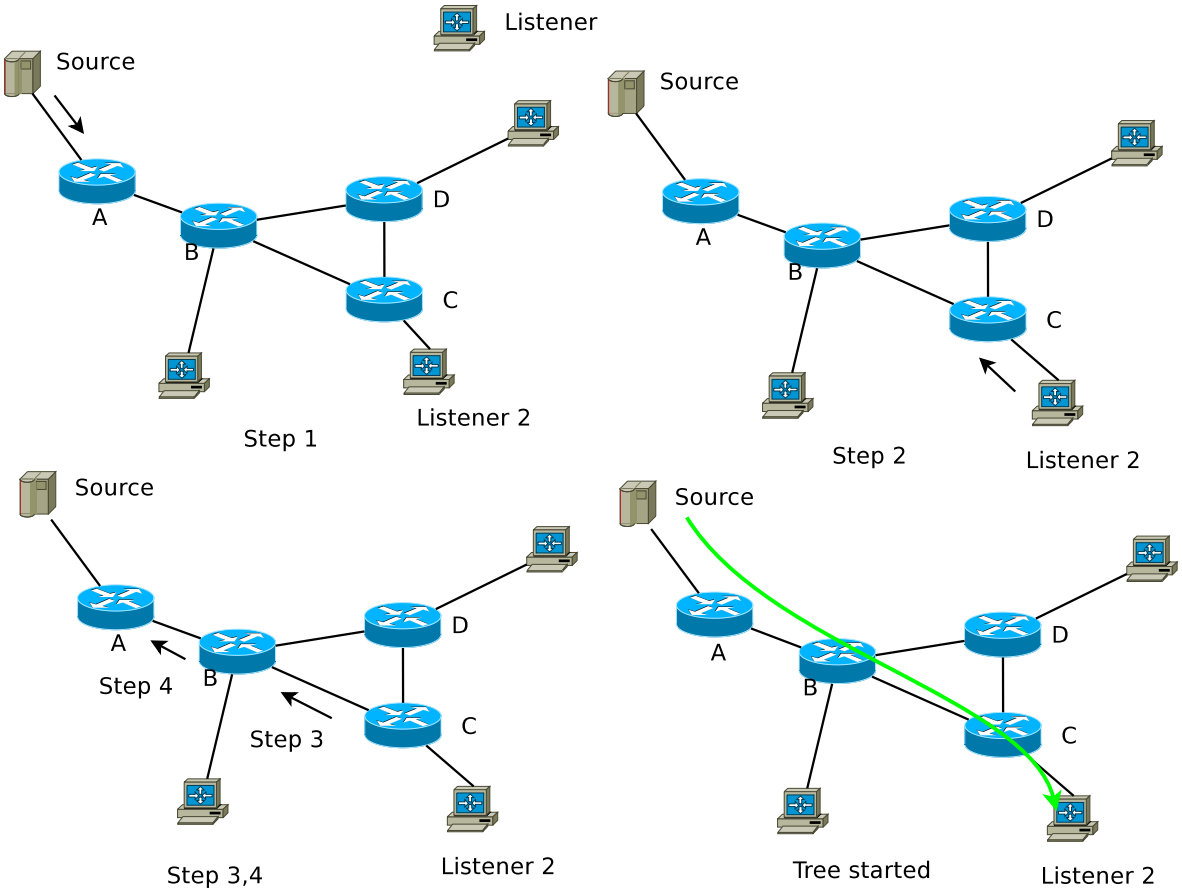


FIGURE 2.5. Multicast Pre-distribution Tree

Each sub network that contains at least one interested listener is a leaf on the tree. When a new listener tunes in, a new branch is built, joining the leaf to the tree. When a listener tunes out, its branch is pruned off the tree. Where the tree branches, routers replicate the data and send a single flow down each branch. Thus no link ever carries a duplicate flow of packets.

The above steps are roughly taken by Protocol Independent Multicasting-Sparse Mode (PIM-SM) and its variants. The SM in PIM-SM refers to the fact the multicast sources are sparse (taken from graph-theory). That is as far we can go with multicast routing as Stevens[**7**] note it can easily consume a book! We refer to [**6**] if this section disappoints.

2.4. Any source vs. Source Specific Multicast

As we stressed earlier, multicasting is a model realized through various underlying protocols. Multicasting itself is traditionally further divided into any source and source specific models[5].SSM is due to Holbrook [**9**]. The any-source part is of

[5]This can be misleading as SSM is subset of ASM. It would have been better to say ASM is divided into...But we stick to the literature.

primary concern to this thesis.

If the user is active participant and does not know all the sources of the groups s/he wants to join, any-source multicasting is appropriate.

In our discussion of IGMPv3, we noted how the user *can opt* to specify sources s/he wants to hear from. Then, it is no wonder that IGMPv3 is subset in realization of SSM model. SSM uses PIM-SM for routing with the simplification that we already know the root of the distribution tree leading a much more strict version called PIM-SSM. That simple assumption eliminates the need for RPTs, RPs, and Multicast Source Discovery Protocol (a lot of the 'hyphens and slashes' we mentioned in Section 2.1), radically simplifying the mechanisms needed to deliver multicast. It somehow becomes *the shortest path* problem between the source and the user and reduces to unicasting routing model. This no longer considers 'group optimization. As mentioned Section2.2.3, support of IGMPv3 is a function of the operating system as of this writing. It may take some time before the majority of hosts on the Internet are enabled for IGMPv3. But also, the restriction of this solution to one model makes it only part of the solution to multicast security we noted earlier. We emphasize the reliance of ASM on the ideas of the earlier versions of IGMP (v1 & v2)- the less strict ones.

By now, we have established our playground, how it differs with its closest ally solution- SSM. We should take SSM in the form of 'related work' to our design solution as we target SSM's super set, ASM.

2.5. Related Works

We have implicitly stated some existing mechanisms like using manual entry of the static addresses in a firewall, and using SSM. We now add some more.

The solutions are roughly divided into two: those that use cryptography mechanisms and those that use various other mechanisms/heuristics to identify the genuine sources-like ours.

2.5.1. IETF MSEC. The IETF Multicast Security Working Group (MSEC WG)MSEC WG[**13**] is the standard body that is working on securing multicasting. The MSEC architecture focuses mainly to using cryptographic mechanisms. It deals with such things as privacy, key management, encryption and is biased towards securing the data than the network. One of its attractive application geared towards our problem is the *shared key* mechanism to authenticate group members. To elaborate it a little more, a member can only join a group if it has the shared key which can be established by the standard challenge-response mechanism. RFC3740 goes into the detail of how it should be done. It is dogged with questions like who is going to be the group manager? How is the membership dynamics? But most importantly do we want to receive traffic from all those who joined the group?

The use of encryption and decryption on every packet has heavy performance hit on real-time data- which multicasting is used for often. Further on this can be found in [**11**].

2.5.2. Other mechanisms. These mechanisms can be seen under one umbrella including ours. The ultimate objective is to let the firewall know 'these are the sources we trust'. For example a solution proposed by [**12**] seeks to implement a private protocol to notify the firewall dynamically the genuine sources that are being contacted. This has very strong assumption on the firewall and the multicast application developer.

Another one is by converging Any Source Multicasting (ASM) to Source Specific Multicasting (SSM). We have thoroughly discussed the models in Section 2.4. It naturally follows if we are using full potential of SSM and are only interested in the service model, we can be secure from DoS/DoS attacks. The following is done to make SSM-incapable hosts (due to IGMPv3) who rely on traditional multicast to join SSM-enabled multicasting.

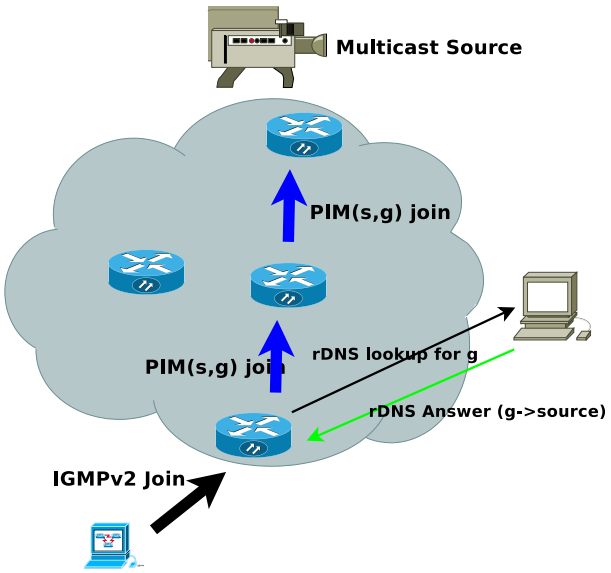


FIGURE 2.6. DNS-based Solution

2.5.2.1. *SSM-Mapping (from Cisco).* The following should not be confused with multicast Domain Name Server (DNS) (mDNS) which is used for LAN resource discovery in line with ousting broadcast resource discovery. SSM-Mapping[**10**] changes the earlier version of IGMP to IGMPv3 'on the fly' thus changing the model. Figure 2.6 tries to show the process.

- Router maps IGMPv2 Joins (in SSM address range) to well- known sources via DNS
- It usually allows only for one, or more, sources per group
- Router maps group to source (sources) – Uses either DNS or static internal database

CHAPTER 3

Design Concepts

If I were stranded on a desert island and could only take one data structure with me, it would be the hash table.

-Peter van der Linden[1]

The purpose this chapter is to introduce to some key concepts that propped in the design of our program. We also walk on the program on a system level what a packet that enters our module will happen to it. Some of the discussion may seem appropriate for the previous chapter but its pertinence is immediate matter to the design we put it alongside here.

3.1. Desired Properties for the Multicast Module

We want to see this from the perspective of two types of people, namely, the Network Engineer and the user- *security & usablity*. Let's further assume as an example: our user is watching Multicast TV programs- on a multi-channel IPTV capable box. This is to illustrate what users want and observe their behaviour and design our solution accordingly. We use the term channel interchangeably with group in the example. So, from the perspective of the user:

(1) We want no delay to traffic, *latency*, and even worse varying delay on traffic, *jitter*. The first one might be tolerable so long as it constant and negligibly small for some applications such as Teleconferencing. This is restating the classical satellite delay problem- what people encounter as conversational gaps and synchronisation problems. After certain threshold- which depends on the application, delay creates confusion among the participants. In short, making sure zero jitter, we still can allow 'tolerable' non-zero latency.
(2) Tolerable delay (for the user) when the user wants to joins a channel.
(3) Tolerable delay (for the user) when a user wants to leave a channel.

The first point is the reason why multicast uses UDP (user datagram protocol), which has the principle of 'an early packet with error is better than a late correct packet' (best-effort delivery). We want to maintain this principle in our multicast firewall design. This entails one specific requirement when inspecting the packet for validity to either drop or accept it at our filter. We need to have a *fast look up* if the source is valid. This justifies the choice of our data structure which we discuss in Section 3.2.

[1]It is actually parody on 'C-compiler', Expert C programming: Deep C secrets

The three list above takes into account the behaviour of the user in our example. One key observation is the following: when a user leaves a channel, it is either to join another one or is leaving for good (finished watching for the day). In the first part there is *genuine leave*, in which case the user actually has left the channel for considerable amount of time. The other leave is *comeback leave*. In this case, the user has left with the intention to come back soon. For example, the user could be avoiding watching commercials on that Channel. Users can be annoyed if they cannot flip back to the previous channel with the back button on the remote. This same behaviour can be easily mapped to perspective of different types of multicast applications. This idea is captured in our algorithm design in Chapter 4. It also plays a role in what should be included our data structures and memory allocation block.

While most of the requirements from the perspective of the Network Engineer has been outlined in Chapter 1, we still want to list some more desired properties. We stress again our multicast firewall is about resource maintainability in the event of attacks (mainly DoS/DDoS). Thus, so much as we want to avoid exposing our network to attacks, we also want to avoid undesired heavy-handedness on internal users.

(1) Our filter should minimize blocking genuine traffic. One special reason that haunts our design to cause false positives[2] is unicast snooping shortfall. In Chapter 6, we outline ways to mitigate this problem of 'network auto-immune disease'.
(2) Our filter should minimize false-negatives. This is of lesser priority than the above. It can be simple 'annoyance' unless orchestrated intentionally attack is taking place. Remember we begin with a completely blocked wall. We are trying to balance usability and maintainability. In this case, the CEO will not ask "why is the teleconferencing program not working?" as the case maybe for the first point.

This leads us to discussion of the data structure of our choice.

3.2. Hashtables

Hashing implements the mathematical concept of a *function*. We choose how our domain changes to the set of values we want it to be transformed to via our *hashing function*. Depending on the input range and output range, hashing has different applications.

Hashtables, which are one part of the application of hashing and the relevant part for this thesis, are about storing values. If we know beforehand we have n distinct inputs (also known as key values) and n associated data values, we store these values in an array of n size by transforming the inputs into range 0-n via our hash function. This is known as *perfect hashing*. In most cases though, we do not know the value and number of the inputs beforehand. So, we simply allocate n units

[2]from firewall point of view, the positives are the ones that are tested to be malicious

of memory of the output types and put a mechanism for *collision* handling when two inputs map to the same address location-bucket. These collisions are often are handled by linked list data structures. Of course, we can use any data structure including a chained hash table. In this thesis, we use linked lists. This is a design choice of simplicity before being fancy without knowing if that is the bottleneck- Rob Pike advices[**18**]. When we have no prior knowledge our hash key and values, we aspire to have hashing function that tries to spread new values *uniformly* across the array. Some of the desirable features (among others) of a hash function for the spreading to happen include:

- Good mixing and combining
- Good speed (input to output)
- Good avalanche effect

The first and third points are to make sure every bit, as in 1/0, of the data has effect on producing the hash value.

Before we digress too much, our main concerns is storing IP addresses. Imagine an internal client with IP address, c, contacts a destination source address, s. In our design, we want to store this association in buckets by using the destination address (which is random) as our key value. For the points we mentioned above, Jenkins Hash[**16**] due to Bob Jenkins is used after doing research. We refer for the explanation of the above points to any Data Structures book .

3.3. A word about Netfilter/Linux

The Netfilter Framework comprises a set of hooks on the Linux networking stack. A hook essentially is a function pointer and gives others the freedom to implement the function themselves- it provides the prototype of the function though[3]. Netfilter is Linux's extensible firewall. Now, Netfilter has access to the packet as argument in hook form at different stages once the packet enters the Linux kernel. It then has the freedom to do stuff with packet including dropping it. There are five different stages where the control of packet is temporally passed to Netfilter:

- PREROUTING: All the packets, with no exceptions, hit this hook, which is reached before the routing decision and after all the IP header sanity checks are fulfilled. Port Address Translation (NAPT) and Redirection that is, Destination Network Translation (DNAT), are implemented in this hook. We can manipulate the data too.
- LOCAL INPUT: All the packets going to the local machine reach this hook[4]. This is the last hook in the incoming path for the local machine traffic.
- FORWARD: Packets not going to the local machine (e.g., packets going through the firewall) reach this hook. In a real system this will be how

[3]This is C's powerful way providing what one calls 'interface in object-oriented paradigm'
[4]As opposed to forwarding traffic.

our module functions. All the packets are checked before forwarding them to one of our clients in the network.

- LOCAL OUTPUT: This is the first hook in the outgoing packet path. Packets leaving the local machine always hit this hook.
- POSTROUTING: This hook is implemented after the routing decision. Source Network Address Translation (SNAT) is registered to this hook. All the packets that leave the local machine reach this hook.

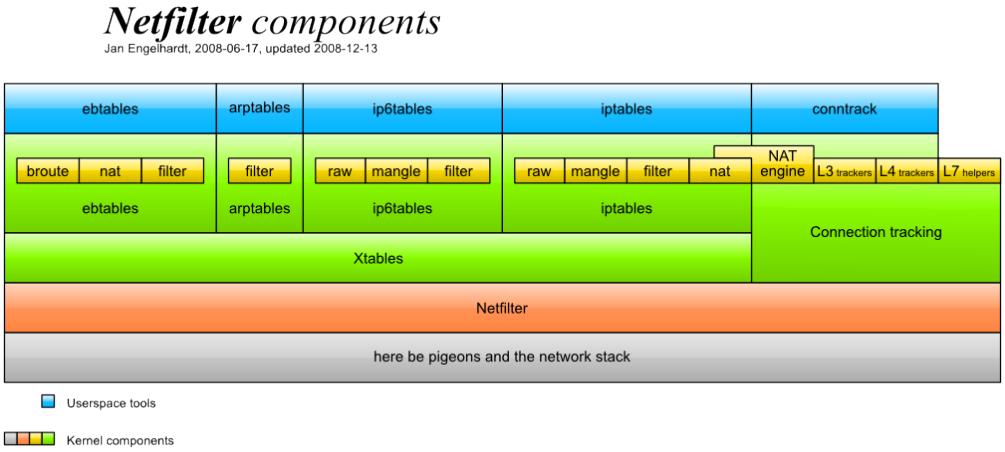


FIGURE 3.1. Netfilter Components (From Jan Engelhardt)

This is a very powerful framework that is capable of doing much more than we can possibly explain here. Then we have Xtables, which is shown on Figure[5] 3.1,on top of it as brick layer. It provides a simpler API so that we can extend Netfilter functionalists by getting access to the packet we want in less complicated way than Netfilter. So, our module is a plugin to this top layer. There is also a code I wrote for userland interaction, this is at Iptables layer, which is mainly based on boilerplate. This Section is very application specific and the appendix might have been a better place.

3.4. System Level Overview of the Module

3.4.1. Our Multicast Firewall. The general concept of the filter was discussed before as 'prescriptive' problem statement. We have also said the way we discover who the sources are is by snooping outgoing unicast traffic. Figure 3.2 tries to show what we are trying to achieve. Darth is the bad guy and Alice is the good

[5]Jan Engelhardt have also graciously provided extensive tutorial on 'Writing Netfilter Modules' and the glue API following changes in kernel. The former has Creative Commons and the latter GPL licenses. All can be found including the picture: http://jengelh.medozas.de/

source. The way we discovered Alice was good source is because someone inside our network has been browsing alice.com. With this, we also expose our strongest assumption in this thesis. That is, somehow that person/application is likely to have the unicast contact of some form with the source. The Figure also shows how Darth's traffic is dropped by our firewall and Alice's accepted.

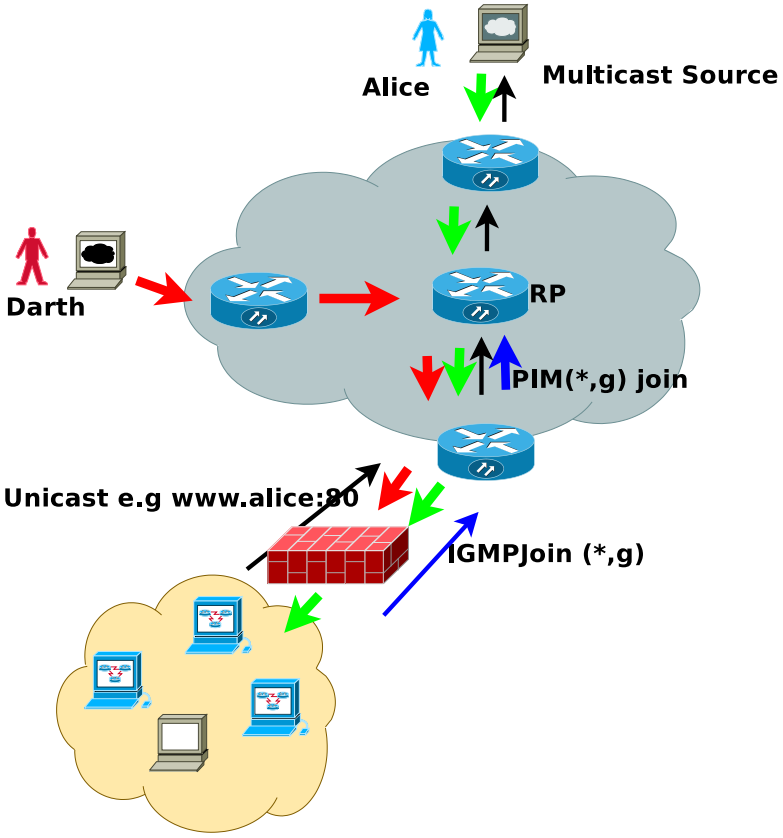


FIGURE 3.2. The dynamic multicast firewall

The formal statement of the above is :

Notation 1. *let* $C = \{c_i\}$ *be the set of internal clients.*

Notation 2. *let* $S = \{s_i\}$ *be the set of external sources.*

Notation 3. *let* $G = \{g_i\}$ *be the set of multicast groups.*

Then the dynamic multicast firewall claims:

Claim 4. *If* c_i *contacts* s_j *and joined* g_i *a traffic from* s_j *to* g_i *is trustworthy.*

We can prove the claim by assuming 'trust chaining[6]' and using transitivity property.

The way we implemented the above claim is:

[6]This is how digital certificates work too.

For outgoing traffic. We keep two arrays of buckets:

(1) One for groups addresses we snoop when clients join groups. Either the destination address is set to group address or the IGMP header has the group address depending on the IGMP version as in 2.2.3.
(2) One for source addresses when clients contact contact sources. As a normal unicast, the destination address (of the IP datagram) is set to the source address.

In both cases, we hash the destination address and put the client's IP address that contacted them.

For incoming traffic. The incoming traffic has both source and group addresses(as the source is the one sending to the group). The IP datagram contains source address as source address and group address as destination address. That means we have both our hash keys. We follow the following steps:

(1) Hash the group address, check if there exists any client at that bucket.
(2) Hash the source address, check if there exists any client at that bucket.
(3) If either one of (1) & (2) is false we drop the packet and finish here.
(4) We cross check those lists until at least one client is common in both. Mathematically, the cardinality of the union of the two sets will be less than sum of the cardinality of each.

The external IP addresses serve the dual purpose of authentication and being hash keys for storing value of the internal clients.

Figure 3.3 shows code flow chart and the ideas mentioned here.

3.5. Data Structures and algorithms

We follow a key concept in Computer Science that 'if we keep certain property when storing a data, we can reap some kind of benefit from it later on. The 'later operations' could be simple data retrieval, search, delete or modify. This is the case with AVL trees, dictionaries, RB-trees and various property-based data structures. The algorithm that is at the core of our module relies on a property we keep in our insertion as we will discuss in Section 4.4. This key property is that we store the last seen client on top. Our data structure is as follows:

```
struct xt\_mcast {
   struct hlist\_node node;
  \_\_be32 ip;  //big-endian ip
  unsigned long timeout;
};
struct mtable {
   struct hlist\_head members[0];
};
```

The `timeout` fields plays that role. The client with largest timeout (the freshest client we saw) would be the head node of the list. Somehow our ways resemble that of operation on a stack.

3.5.1. Code flow Chart and Algorithm. The explanation of the algorithm proceeds in the next Chapter.

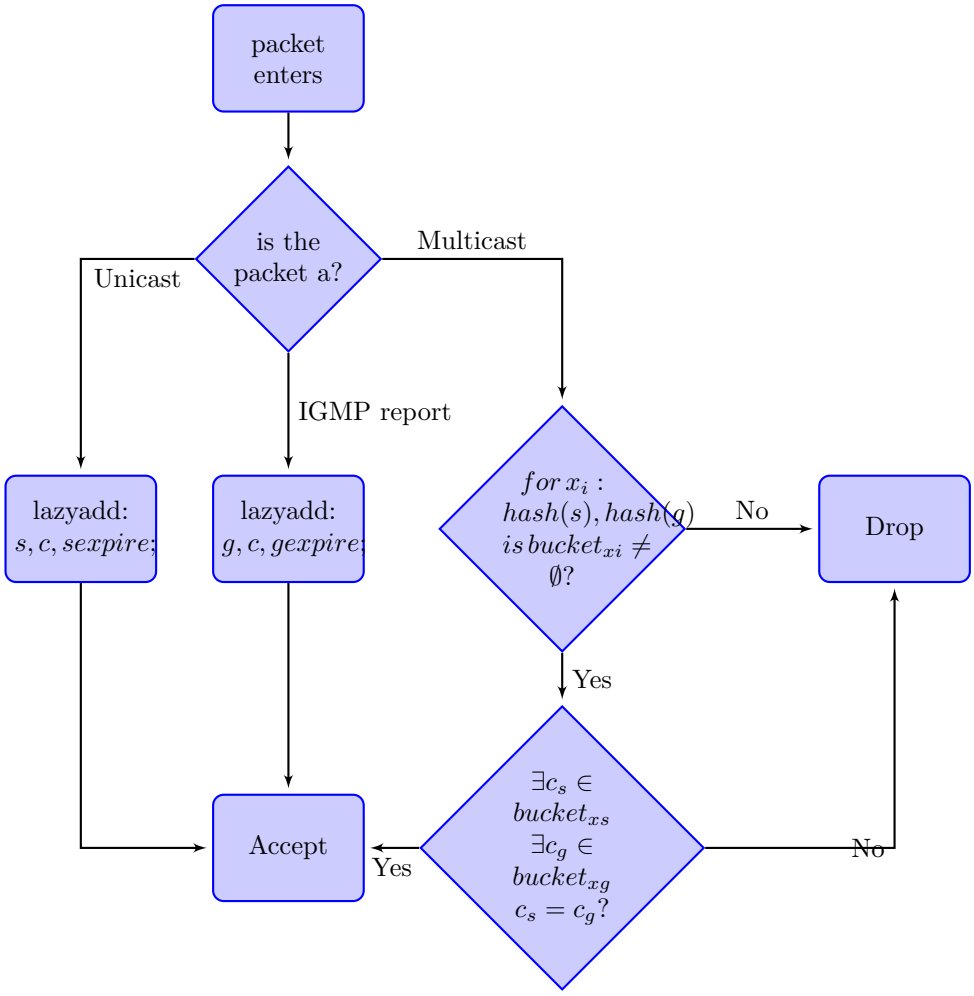


FIGURE 3.3. Code Flow Chart

```
   Input: kip, vip, expire, array
   Result: inserts vip & deletes expired entries at bucket hash[kip]
 1 index = hash(kip);
 2 entry= null;
 3 head= ref = array[index];
 4 while ref is not null do
 5 |   time = ref- > timeout;
 6 |   if now ≤ time then                    /* The node is expired.  */
 7 |   |   entry = ref              /* hold on to the place-memory!  */;
 8 |   |   ref = ref- > next                    /* simply advance it */;
 9 |   |   while ref is not null and ref is not head do
10 |   |   |   temp= ref;
11 |   |   |   free(ref)        /* all are ripe to be deleted-expired */;
12 |   |   |   ref = temp- > next;
13 |   |   end
14 |   else
15 |   |   ref= ref- > next;
16 |   end
17 end
18 if entry is null then          /* if we are not lucky from line 7 */
19 |   allocate(memory,entry);
20 else
21 |   ;
22 end
23 entry- > timout= now + expire     /* should have memory by now */;
24 entry- > ip = vip;
25 if entry is not head then  /* freshest entry must always be head */
26 |   hlist_add_head(entry,array[index]);
27 else
   |                          /* memory was from head- do nothing */;
28 end
```

Algorithm 1: the lazyadd algorithm

CHAPTER 4

Detailed Design

Make everything as simple as possible, but not simpler.

-Albert Einstein

4.1. Design Dilemmas

We start with something we have implicitly deferred for a while. This is the case of IGMP leave messages. They have been left out from the code flow chart of the previous Chapter intentionally. Leave message can be deceptive in the spirit that we discussed as *comeback leave & genuine leave* earlier. The design dilemma posed by that is: *do we go to that particular group hash and remove the client entry from there or simply ignore them?*. If we remove the client to just discover it comes back would be inefficient. IGMP leave messages are, of course, good for routing scalability. In our case though, they can destroy the cache like behaviour we worked hard for by removing the client prematurely. Even though we included that implementation in our module, simplicity and less work bears against it.

Another dilemma is 'how long should a source be valid for?'. That is once it had unicast interaction with one of our clients. There is no good answer. In the module we passed that question to the Network Manager to decide (using the userland Iptables program s/he can set the value on command line). Imagine our IPTV user again and a hypothetical scenario. S/he switches on the box and during booting the box contacts the content providers IP address (maybe during that action the filter notices the contacted IP address). The user is passively watching a movie from then on and no unicast interaction occurs. It would be undesirable to drop the packets while the user is watching.

4.2. Where is the Module and Why?

As can be seen on Figure 4.1, our Filter is located right after the switch. This mainly due to physical reason of getting the whole traffic in one interface. That is, the switch is where the traffic is multiplexed and demultiplexed and as such it is the converging point of outgoing traffic. This is called Bridge firewall- Bridgewall- which is also powerful as it can also do layer 2 filtering (we can, for instance, deny access to a user by MAC address-IP spoofing impossible- who is contacting bad IP addresses).

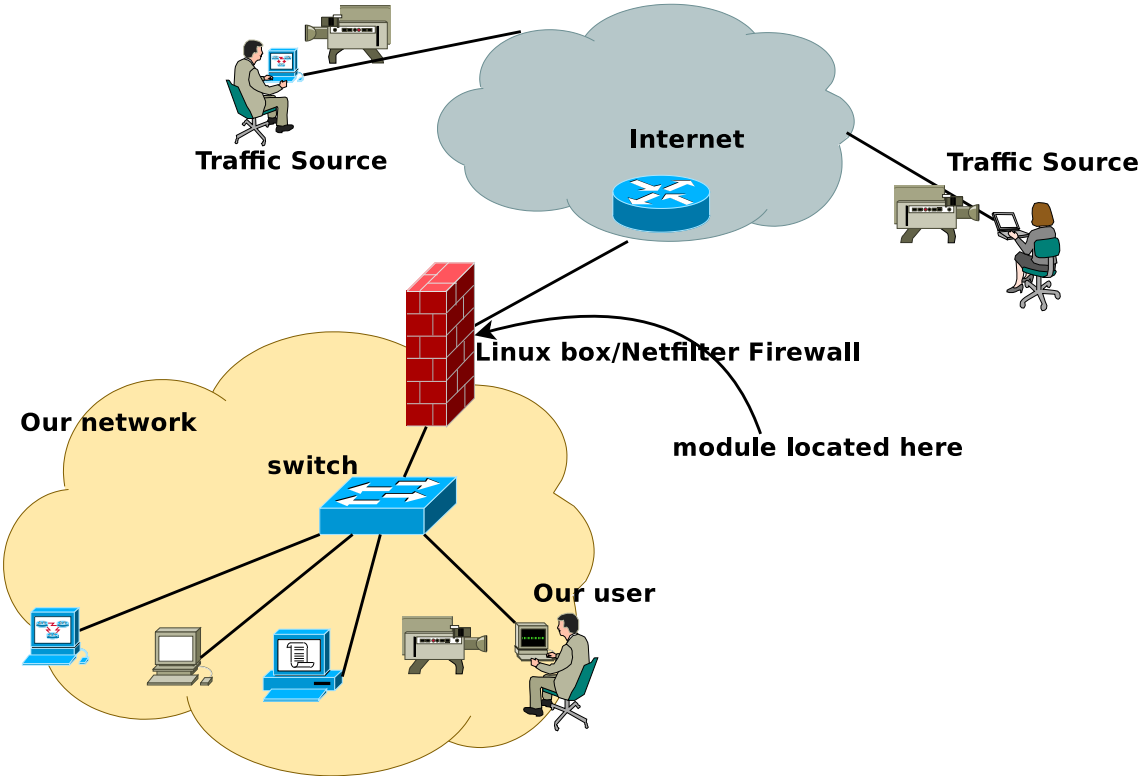


FIGURE 4.1. Module location

4.3. The Memory Management Paradigm

One performance issue that dogged a previous student design solution was a periodic 'call back' function that goes and cleans entries when a timer expires. And later on, goes to ask to allocate new memory when new entries appear. This can keep the processor quite busy with loads and stores to and from main memory while also destroying possible localities maintained by cache[1]. Memory allocation is also an expensive process. And when we request for more memory every now and then, it leads to $O(n)$ linked list complexity (one fragmented memory points to next as standard linked list structure) to access what we want. This is restatement of memory 'fragmentation' problem. There are smart ways of allocating- what the 'slab allocater' does in the Linux kernel. It is not in the document scope to discuss further on this matter.We urge the reader for some faith that less call to any of _malloc family functions is better. If not convinced we refer for discussion memory fragmentation and all other memory related stuff to [**19**].

The fundamental basis behind the design the algorithm about to be discussed is:

- Asking kernel's memory management system (the slab allocator in Linux) for a new block of memory right after freeing one is to be avoided. Instead

[1]This can cause the familiar computer 'crunching' sound- when programs that (ab)use the memory too much are running.

we want to overwrite that with a new entry. This is in the same sense of 'one will not break a plate after eating on it and go out to buy new one when s/he want to eat again'.

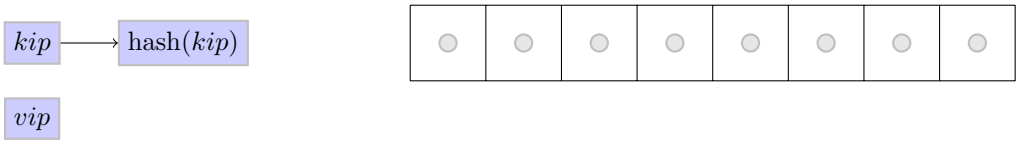
On further resolution the above statement, we can ask when is the right time to free it and how are we going to decide that memory is not needed anymore? As we will see in a bit the right time is when we go to that bucket location and we free it if the timer on that node is expired.

The Lazy Cleaner's Dilemma* (Optional). We introduce a motivation for our design that we call the *lazy cleaner's dilemma*. The cleaner wants to maximize cleanness and orderliness of stuff s/he uses but minimize cleaning work. The cleaning precedes the use; the stuff is considered dirty after some time of its use. We can raise a questions like 'When is it optimal to wash a cup that he used now to be considered dirty'. It is not unusual to see people have another tea on the same cup. Examples of non-optimal solutions: periodic cleaning, cleaning stuff s/he is not likely to use again. I urge you to map this analogies to memory allocation call optimisation.

4.4. The Lazyadd Algorithm (Insert/Delete)

Let's begin with the several assumptions

- We have allocated array of n size (take array size of 8 for illustration),
- Initialised (for each bucket-cell) a head pointer to a a linked list node structure. That is what we call the initialisation step if it referred to below. The full glory of the data structure can be seen in Appendix A.2.2[2].
- Our value IP address *vip* to be stored and hash key IP address,*kip*, just as below:



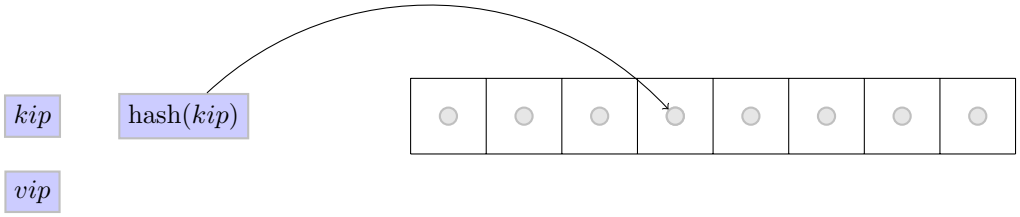
One term that we use is dirty node which we define as a node that a timer associated with it has expired. We can proceed to the steps.

Our algorithm follows roughly these steps to add *vip* in a bucket located by hash[kip]:

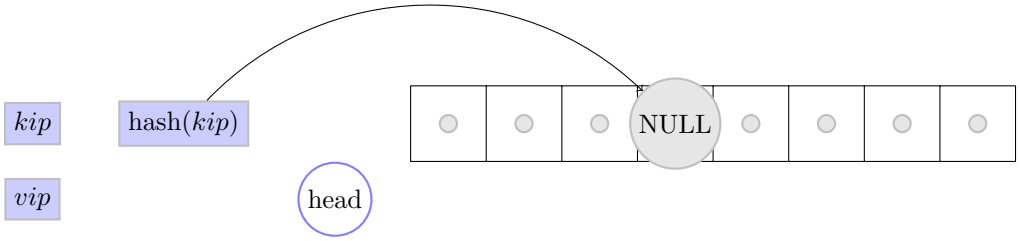
4.4.1. No collision.

[2]The way a linked list is implemented (and how one can utilize that) in the Linux kernel is slightly different from standard Computer Science courses and for us, it is another matter. We refer to the source code /linux/list.h if one is interested. It safe to assume standard linked list data structure with next, prev pointers for each structure to understand these algorithms.

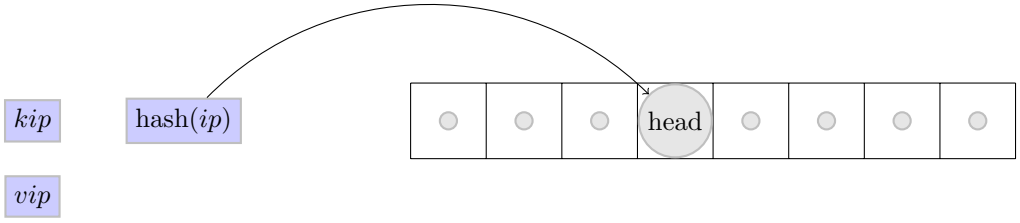
(1) Check the bucket pointed by the hash.



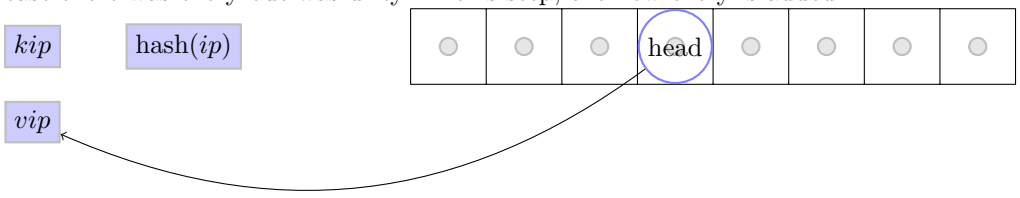
(2) If the head pointer (which is valid during the initialisation step) still points to NO linked list, create a entry node- head node[3].



(3) This is a case when the head node is still there but it is dirty (i.e the timer on it has expired).



(4) From one of the two above steps, we have *memory*. In the first case there was explicit allocation; as there was no entry before, and in the second case there was entry but was dirty. In this step, the new entry is added.

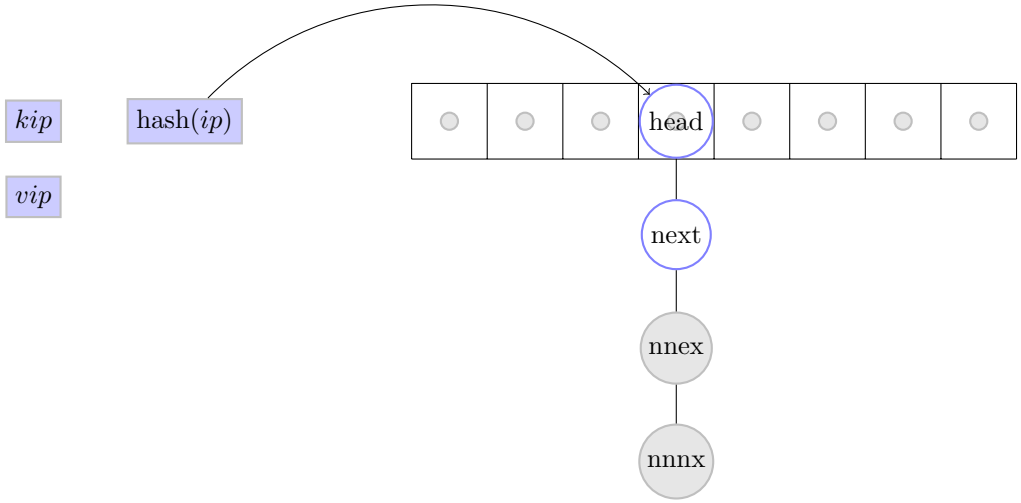


Now let's assume we have had collisions (we have more entries in that bucket), the list is not dirty up until some middle node- the entries are fresh and have not expired. We put this general case so the other cases can thought out or inferred. This leads us to the next section:

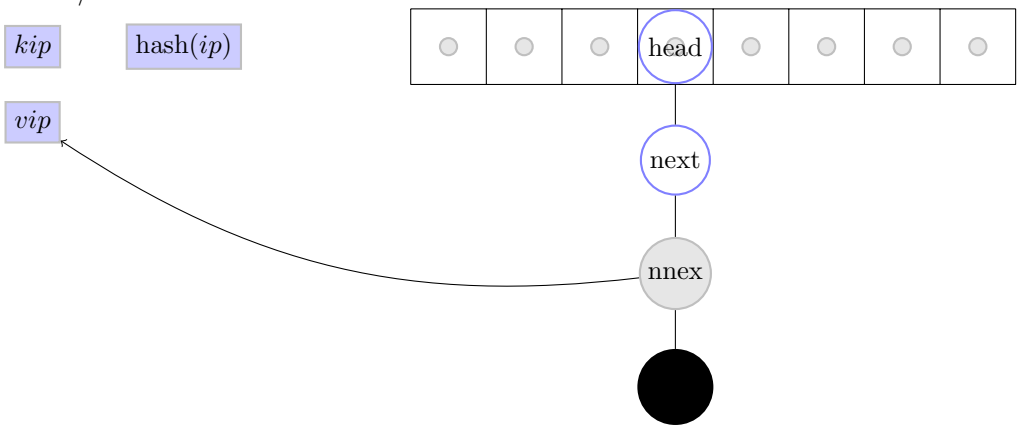
4.4.2. Collision.

[3]This will be the "head node". The head pointer is in the one in the array- small dots. Remember that the picture covers the head pointers when we show nodes- the nodes are sort of "pinned" on it

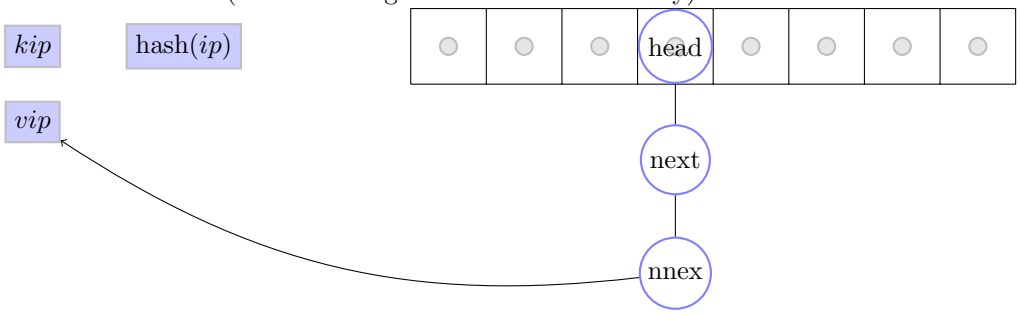
(1) We have a structure that looks like the following



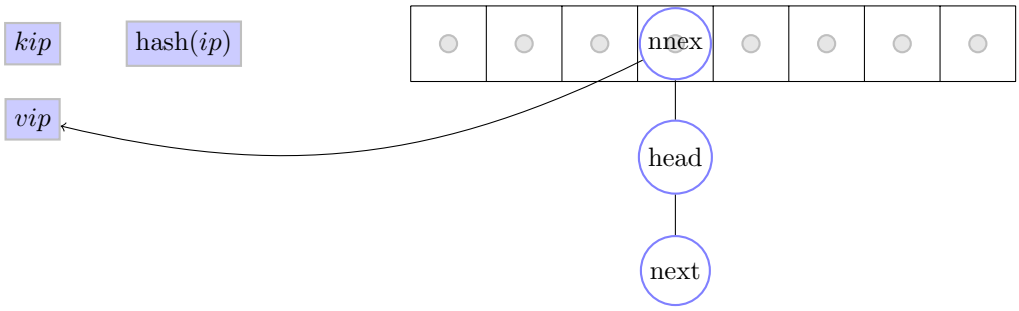
(2) This is a very important step and key part in our algorithm.
   (a) Traverse the list until we hit the first expired- we grab its space- and store the new value.
   (b) Delete/clean the list downwards.



(3) We restore the property which allows us to do step (2) in two steps
   (a) Activate the timer (which no longer makes that node dirty)



   (b) This is the last to expire and so make it the head node (the property). This allows to delete the rest of the list once we find latest expired node as we did in (2b).



This completes the process of the lazy cleaner insert algorithm.

We have to be careful not to delete the head node at step 2.b if the linked list structure is circular. This is the case with Linux kernel implementation of the data structure. Therefore, we need to make sure to hold head node and compare it is not the node we are at, while dropping[4] the expired structures.

4.5. Design Tradeoffs

4.5.1. Performance, Memory, Correctness, Design Complexity. Should we store the source values that our client is trying to contact? What could it possibly mean we are not storing those values? Imagine a scenario where a malicious user's address maps to an address location which has client (that is due to possible collision with genuine source). The probability of that happening can be quite significant depending the property of the hash function. The module does not check this but it is slight modification to the data structure including one more ip address something I missed. We lose extra storage of memory and one more compare instruction for the processor in return for correctness.

The performance (which is packet latency at our module) has direct relation with how many memory cells we allocate to group and client arrays. This is a simple hashtable tradeoff. We see more of it in Chapter 5.

We can also ask if it is worth the design complexity to, for example, abandon linked list and use other faster access complexity structures. The answer at this design stage is no as we have not identified that as bottleneck.

[4]As not to cause another 'gravity theory moment'.

CHAPTER 5

Experiments and Results

Epigram 95. Don't have good ideas if you aren't willing to be responsible for them.

-Alan J. Perlis

5.1. Basics Functionality Setup

5.1.1. Tools needed.

- Linux based system as outlined in README in Appedix A.1.
- Iptables Netfilter Modules as outlined in the README in AppendixA.1.

The easiest way to check the functionality is to start a streaming video on VLC[1] server. This is simply done using GUI of the player. Then, start that session in VLC client. Again this is done using GUI feature of VLC. One can use the shell scripts given in Appendix to test the blocking and unblocking functionalities.

5.2. Resilence Test (Simulation)

5.2.1. Test platform.

Operating System: OpenSuSE11.0 with Linux kernel version 2.6.25
Architecture: x86_64 Pentium D
Memory: 2GB (RAM 2DDR)

5.2.2. What we are measuring. The main performance concern as previously mentioned is the latency. Therefore, what we need to test is how long will a packet stay in our module before a decision is made to either drop or accept that packet. For that, a userland simulator is rewritten which is based on the main kernel module. There was a time and resource constraint to test the module on real systems.

For the sake of getting pattern on how increasing number of clients, groups and sources contacted influence, the numbers are made large. This is in contrast with real networks where hundreds of clients might only be. The source code is provided in Appendix B.

As can be infered from the source code, we have made the following assumptions:

- The space is kept at only 20 buckets (array)- again this is for the sake getting a pattern not because of lack of memory.

[1]This a very popular media player and can be found on www.videolan.org

Clients	200	500	600	700	800	900	1800	3000	4000	10,000
Time (μs)	2	3	4	5	8	9	24	45	60	150

TABLE 5.1. Number of clients vs raw processing time

Groups	10	20	50	100	200	500	1000	2000
Time (μs)	5	8	10	10	11	11	10	7

TABLE 5.2. Number of groups vs raw processing time

- Every client is participating in at least one random group.
- Every client contacts some random external host as its source.

Then what we try to simulate is basically the searching (at peak time when none of the groups or sources have expired). It would also be interesting to see what the effect of expiring time has. We may have to defer that to our recommendation. We have not made any assumption whether the source is malcious or not. We average the time it takes for a packet to pass through or be dropped over three hundred source requests (a source delivering a multicast packet) .We run that again on the terminal 10 times and average the result.

5.2.3. Number of clients as variable. We keep number of groups at 100 and number of sources 10000

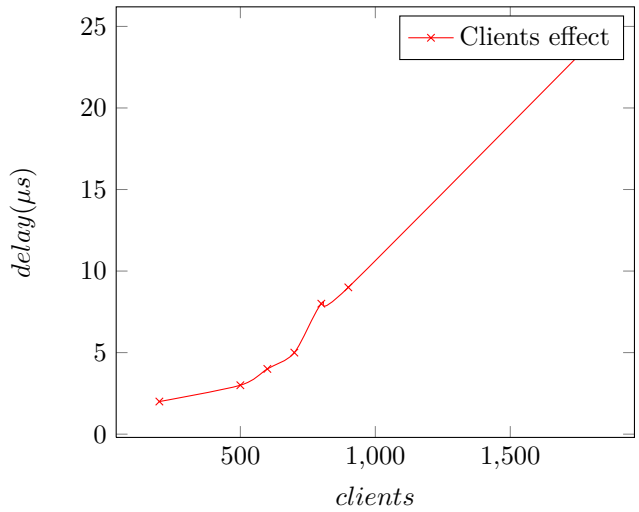


FIGURE 5.1. Effect of Number of Clients on latency

5.2.4. Number of groups as variable. We keep the number of clients at 1000 and the sources at 10000. The assumption being one client contacts to ten possible unicast addresses at a time (e.g web browsing different sites.)

Sources	1	10	20	50	100	200	500	1000	10000
Time (μs)	5	6	10	8	11	10	10	7	6

TABLE 5.3. Number of groups vs raw processing time

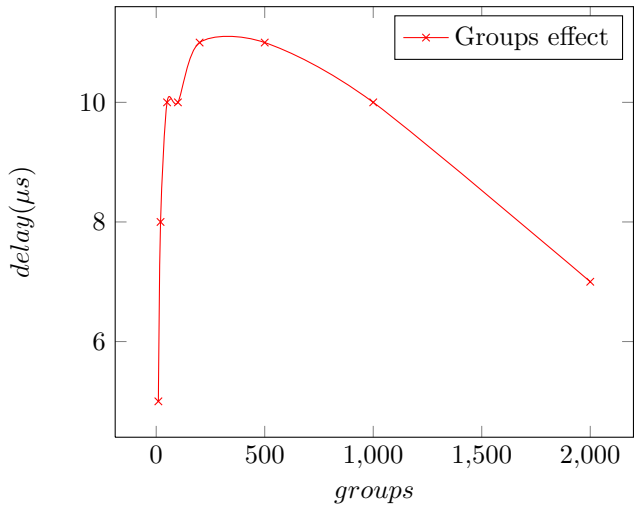


FIGURE 5.2. Effect of Number of Groups on latency

5.2.5. Number of sources as variable. We keep the number of groups 100 and clients at 1000.

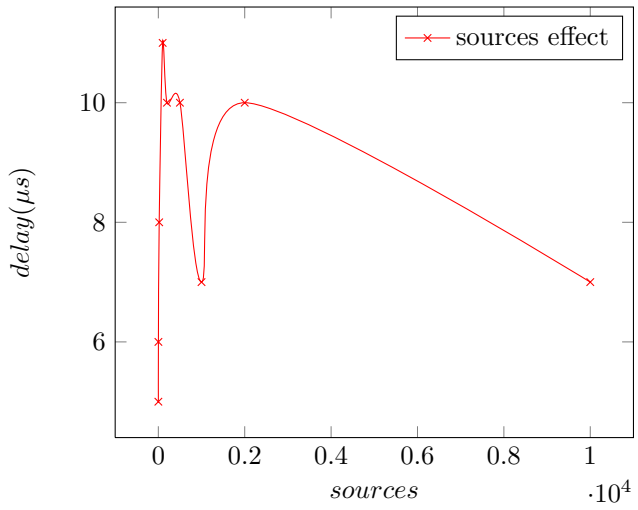


FIGURE 5.3. Effect of Number of Sources on latency

5.3. Results Interpretation

The results shows the module's dependence on internal clients number- as would be expected. The more clients we have the more states we have to keep. In other words, it is our clients that cause the state information creation by initiating those

contacts. If we have more clients, they are likely to contact more groups and group sources.

More interestingly though, the more groups we have outside, the more likely our clients will contact different groups. This causes a faster hash access to the clients. Remember we use the group and source IP addresses as hash keys to store the clients and then do a cross check. If these values become diverse, which is what happens when we increase their domain, their hash value is likely to differ too and thus causing less collisions. This is what is apparent on the the latter two graphs and on my tests. This aligns perfectly with the design challenge of the dynamic multicast firewall, which does not know what the source IP or group IP are before and considers them random. Be warned though, this assumption of randomness may not be true in practice. And also, the actual network load (not our management) maybe unbearable when so many sources and groups exist.

Lastly, even if the results put are in the order of μs, we need to bear in mind multimedia applications (e.g IPTV) that mulitcasting is used for are in the order of $Mbps$. The result can be that we are not processsing as fast or faster than the bitrate which is undesired for the application. We would be introducing artifact bandwidth throttling by our module. For instance, if we are watching real-time video, the implication is we no longer can watch it properly not because of lack of bandwidth but the fact that the module has clogged it. Therefore, the machine's raw processing power and other features are also important in these type of situations and need considerations.

CHAPTER 6

Discussion

6.1. Mitigating Unicast Starvation

The strongest assumption we made as discussed was that of unicast interaction of sources and clients. Here are some ways that may be able to help mitigate those problems:

- If a user/multicast application is not getting its/his traffic, see what ports, protocols and unicast control message the application use and add those to the monitoring list for unicast snooping.
- It is often helpful to monitor web browsing (but it may increase the state)- it is again up to the Network Manager to decide by adjusting the memory required and the expire time of sources.
- If other means do not work , explicit pings to the unicast sources are always easy fixes.

6.2. Another Case Against NATs

NATs are haunted by controversies but we add one more of our own. Imagine the following scenario: a source from NATted network is a valid and one of clients tries to contact it. Due to the nature of the network on the side, our module will assume all sources from the NATed network are valid. The problem is artefactual and we do not consider it in our design as based on Engineering principles.

6.3. Why Access List Security can be Hard Multicast?

Multicasting has a seemingly very strange semantics. If two processes open the same port as a multicast socket and then join two different multicast groups then traffic for both multicast groups is forwarded to either process. This means that application will get surprising data that they did not ask for. Applications will have to filter these out in order to work correctly if multiple applications run on the same system.

The intent was that multicast address socket behavior would be just like unicast address behavior, with the exception that multicast addresses are interface-specific. If we add a new unicast address to the machine, and the binding on the socket doesn't restrict it, you will receive packets from any of the addresses on the machine. That's what INADDR_ANY means. Your multicast application may also receive unicast traffic on that port too.

Some operating systems supporting Multicast have since changed to only supplying multicast traffic to a socket that was selected through multicast join operations. (application will no longer receive traffic from multicast groups that it did not subscribe to.) That does not totally avoid the problem except violating the RFC which is explained in the second paragraph. The problem is anyone on the network, or within multicast routing domain, may reuse both-multicast group and port- for a different machine and the application will receive them. The other machine will also receive the traffic destined for the application.

Why did we go to 'unnecessary' detail above? It should be clear the *virtualness* of multicast address makes it easy for anyone to get others people data and fill garbage on the group address. In security point of view, there is a breach if someone has access to the data even if they cannot decrypt it.

CHAPTER 7

Recommendation, Future work

There can be ambitious work that can be done to better this work which include:

- Testing and using another data structure as collision handlers (static arrays for example). This is in line with claims[**14**] that linked list spoils the CPU cache behaviour.
- To migrate the code to using RCU (Read Copy Update)[**15**] mechanism on the data structure instead of traditional locking we used. This is new elegant paradigm of synchronization of multiprocessors though patented by IBM.

And yes, the Author has done some work on the inclusion of IPv6. Part of the code can be found in the link given the Appendix . In fact, my wish was I complete the work in protocol independent way. It is fairly easy to infer what protocol has been used from either Netfilter or raw packet buffer. I discourage the use of protocol-dependent 'tricks' unless it is explicitly used for that particular protocol. This is a simple restatement of the principle of portability.

CHAPTER 8

Conclusions

In Chapter 1 we looked at the benefits of multicasting and what kind security threat the use of it entails in SMEs. The main framework of thinking we emphasized is ultimately the traffic is coming from source with real IP address. The whole design challenge in this security issue is how we can do the mapping of the group virtual address to the real unicast source address. The dynamic stateful multicast paradigm works on the premise 'if one contacts the group they are also likely to contact the source they are seeking'. This, we said, is the strongest assumption of the thesis. But, it is also a realistic assumption.

In Chapter 3 and4, we looked at the algorithms and data structures we modelled and implemented this idea with. We model it as to provide with a good analytical view the problems that are we are faced. The way it can be conceived is as a security glue between SSM and ASM. It tries to feel the crack that can be opened when one steps to or remains in ASM service model.

We emphasized the design challenges, observations and algorithms than our particular implementation on the Linux Kernel. They are of greater importance if anyone wants construct similar endeavour and as life cyle of kernel, as one facet of technology, changes rapidly.

APPENDIX A

Module Code

The compatibility source code for different versions of Kernel from the link: http://github.com/bewket/adhoc-multicast/tree/master. It is taken from Jan Englehardt's (http://jengelh.medozas.de/) xtables-addons which seeks to provide that.

A.1. README

```
        This module provides a filter for incoming multicast traffic by doing the f/g:
1. Snooping Subscibe (or join) messages and storing the client by its group hash.

2. We also snoop mulitcast source address (unicast type). This is the part where one sort of NEEDs

to pierce a hole for the multicast (which can be seen in the second last line) source-

its unicast address.

We store the client by the multicast source hash.

3. When multicast trafffic come we hash both group and source address and

Checking if the client exists in both hashes.

You need iptables >= 1.4.3 (x_tables module)

and kernel-source >= 2.6.17

Tools: socat-or netcat-, vlc

For testing purpose do the following (need to be root) :

# sh test.sh

Start vlc (www.videolan.org) stream video  on 238.234.234.1

Start extra vlc. Try to open the video on open network and entering the streaming

address. It should NOT open.
```

You will have to choose UDP in both cases.

Then do the following (after changing the 192.x.x.x to your ip)

echo "hi" | socat - udp-sendto:192.168.1.4:4444

and it should now open

You can download the code by using git. At the terminal type (this

$ git clone git://github.com/bewket/adhoc-multicast.git

A.2. The filter Module

A.2.1. xt_MCAST.h.

```
#ifndef _XT_MCAST_TARGET_H
#define _XT_MCAST_TARGET_H
#include <linux/types.h>
struct xt_mcast_target_info {
    uint32_t msrc_size;
    uint32_t mgrp_size;
};
#endif /* XT_MCAST_TARGET_H */
```

A.2.2. xt_MCAST.c.

```
#define MSRC_SIZE    20 //src->cli_head_nodes
#define MGRP_SIZE    20 //grp->cli_head_nodes
#define MGRP_EXPIRE   120 //seconds
#include "compat_xtables.h"
#include <linux/skbuff.h>
#include <linux/module.h>
#include <linux/jiffies.h>
#include <net/ipv6.h>
#include <net/ip.h>
#include <linux/ip.h>
#include <linux/igmp.h>
#include <linux/list.h>
#include <linux/jhash.h>
#include <linux/netfilter.h>
#include <linux/netfilter/x_tables.h>
#include "xt_MCAST.h"
//you can compare addresses directly without htonl
```

```
 //if (ipv6_addr_cmp(&iph->saddr, &new_addr) == 0)
static u_int32_t jhash_rnd;
struct xt_mcast {
    struct hlist_node node;
    __be32 ip;
    unsigned long timeout;
    //unsigned long  expire; //this is in jiffies (allowed per group)
};
struct mtable {
    struct hlist_head members[0];
};
static struct mtable *msrc_hash;
//client list indexed by hash(src)->cli_ip
static struct mtable *mgrp_hash;
//client list indexed by hash(gid)->cli_ip
static unsigned int source_size = MSRC_SIZE;
static unsigned int group_size = MGRP_SIZE;
static unsigned int source_expire = 2 * 60 * 60;
//2-hours (has to
static DEFINE_MUTEX(mcast_mutex);
static DEFINE_SPINLOCK(mcast_lock);
static struct hlist_node *get_headnode(struct mtable *t, uint32_t h)
{
    return t->members[h].first;
}
static unsigned int
init_new_entry(struct xt_mcast *entry, __be32 ip, uint32_t h,
       struct mtable *t, uint16_t expire)
{
    //we are not going to simply add it.
    //go to the list if there is someone who has been expired
//start from the head downwards
    //if the head is expired simple update the head with this
//we want to avoid as many memory alloc calls as possible
    struct hlist_node *pos, *temp;
    struct xt_mcast *temp_entry;
    struct hlist_node *ref = get_headnode(t, h);
    entry = NULL; //just in case
    spin_lock_bh(&mcast_lock);
    hlist_for_each(pos, &t->members[h]) {
temp_entry = hlist_entry(pos, struct xt_mcast, node);
```

```
if (temp_entry != NULL && time_after(jiffies, temp_entry->timeout)) {
//timeout
    entry = temp_entry;
    //the first time this we want to delete all but the first expired
    //go until you find the first timeout and delete downwards.
    pos = pos->next;
//don't delete pos- just advance it
    while (pos != NULL && pos != ref) {
temp_entry = hlist_entry(pos, struct xt_mcast, node);
temp = pos;
pos = NULL; //No pter reshuffle here
pos = temp->next;
kfree(temp_entry);
    }
    break;
}
    }
    if (entry == NULL)
entry = kmalloc(sizeof(struct xt_mcast), GFP_ATOMIC);
    else
printk("Updating entry\n");
    entry->ip = ip;
    entry->timeout = jiffies + expire * HZ;
    hlist_add_head(&entry->node, &t->members[h]);
    spin_unlock_bh(&mcast_lock);
    return NF_ACCEPT;
}
static inline uint32_t mcast_hash(__be32 name, unsigned int size)
{
    return jhash_1word(name, jhash_rnd) & (size - 1);
}
static struct hlist_node *lookup(__be32 hkey, __be32 ip, struct mtable *t)
{
    struct xt_mcast *entry;
    struct hlist_node *n = NULL;
    if (t == NULL) {
printk("Table is NULL\n");
return NULL;
    }
    //use of the mutex should be appropriate here-Noisy Kernel unless new!!
    spin_lock_bh(&mcast_lock);
    hlist_for_each_entry(entry, n, &t->members[hkey], node) {
```

```
if (entry->ip == ip) {
    spin_unlock_bh(&mcast_lock);
    return n;
}
    }
    spin_unlock_bh(&mcast_lock);
    return NULL;
}
static unsigned int igmp_report(__be32 cli, __be32 grp)
{
    struct xt_mcast *entry = NULL, *head_entry;
//aka head
    uint32_t h = mcast_hash(grp, group_size);
    struct hlist_node *head_node = get_headnode(mgrp_hash, h);
    struct hlist_node *entry_node;
    printk("IGMP REPORT\n");
    printk(NIPQUAD_FMT "->" NIPQUAD_FMT "\n", NIPQUAD(grp), NIPQUAD(cli));
    if (head_node == NULL)
return init_new_entry(entry, cli, h, mgrp_hash, MGRP_EXPIRE);
    //go fish
    entry_node = lookup(h, cli, mgrp_hash);
//here there is grp alrdy but is the client in it.
    if (entry_node == NULL) {
return init_new_entry(entry, cli, h, mgrp_hash, MGRP_EXPIRE);
    }
    spin_lock_bh(&mcast_lock);
    //good the client is in the grp.
    head_entry = hlist_entry(head_node, struct xt_mcast, node);
    entry = hlist_entry(entry_node, struct xt_mcast, node);
    entry->timeout = jiffies + MGRP_EXPIRE * HZ;
    //we need to also update the same entry in the source list
    if (entry->ip == head_entry->ip)
goto out;
    //it the head. don't do anything
    hlist_del(entry_node); //shuffle and fix
    hlist_add_head(&entry->node, &mgrp_hash->members[h]);
    //we want this node to be head so we need to del:
    //NOTE: when deleting a grp if it the head pointer
  out:spin_unlock_bh(&mcast_lock);
    return NF_ACCEPT;
}
/* this probably will increase complexity and superflous somehow
```

```
 * we can just ignore this and use
 * our clean first and insert mechanism in fact this  removes
 * temporal locality of our entries
 */
static unsigned igmp_leave(__be32 cli, __be32 grp)
{
    uint32_t h = mcast_hash(grp, group_size);
    struct hlist_node *entry_node;
    struct xt_mcast *entry, *head_entry; //aka head
    struct hlist_node *head_node = get_headnode(mgrp_hash, h);
/* entry is NULL => "I am leaving this grp"-
*yet there is no such grp or doesn't have me
 * shouldn't be possible */
    if (head_node == NULL)
return NF_ACCEPT;
    entry_node = lookup(h, cli, mgrp_hash);
    if (entry_node == NULL) //weird
return NF_ACCEPT;
    entry = hlist_entry(entry_node, struct xt_mcast, node);
    //some clients may just join and leave straight away after becoming last_seen
    //Note: Also IGMPv1 don't issue leave
    //The simple solution here is to make the head of the group last_seen.
    //which often is true
    head_entry = hlist_entry(head_node, struct xt_mcast, node);
    entry = hlist_entry(entry_node, struct xt_mcast, node);
    spin_lock_bh(&mcast_lock);
    if (head_entry->ip == entry->ip && entry_node->next != NULL)
//promote the successor to be head
hlist_add_head(entry_node->next, &mgrp_hash->members[h]);
    //else it is the last entry
    hlist_del(entry_node);
    spin_unlock_bh(&mcast_lock);
    kfree(entry);
    return NF_ACCEPT;
}
//Incoming traffic MCAST traffic
static unsigned int mcast_handler4(const struct iphdr *iph)
{
    __be32 msrc_ip = iph->saddr;
    __be32 mgrp_ip = iph->daddr;
    uint32_t hs = mcast_hash(msrc_ip, source_size);
//h(msrc_ip)->cli_ip
```

```
    uint32_t hg = mcast_hash(mgrp_ip, group_size);
//h(mgrp_ip)->cli_ip
    struct xt_mcast *msrc2cli, *mgrp2cli;
    struct hlist_node *n, *n2;
    //check if there exists any client(s) that are mapped to msrc_ip
    //while looking hash them and check if they have
    spin_lock_bh(&mcast_lock);
    hlist_for_each_entry(msrc2cli, n, &msrc_hash->members[hs], node) {
hlist_for_each_entry(mgrp2cli, n2, &mgrp_hash->members[hg], node) {
    if (msrc2cli->ip == mgrp2cli->ip) { //success
spin_unlock_bh(&mcast_lock);
return NF_ACCEPT;
//remember we won't update anything here (
    }
}
    }
    spin_unlock_bh(&mcast_lock);
    return NF_DROP;
}
static unsigned
int igmp_handler(const struct iphdr *iph, const struct sk_buff *skb)
{
    const struct igmphdr *igmph = igmp_hdr(skb);
    const struct igmpv3_grec *rec3;
    switch (igmph->type) {
    case IGMP_HOST_MEMBERSHIP_REPORT:
    case IGMPV2_HOST_MEMBERSHIP_REPORT:
return igmp_report(iph->saddr, iph->daddr);
    case IGMPV3_HOST_MEMBERSHIP_REPORT:
//this special V2 equivalent case we can handle per rfc3376
if (ntohs(igmpv3_report_hdr(skb)->ngrec) == 1) {
//ONLY one grp
    rec3 = igmpv3_report_hdr(skb)->grec;
    if (ntohs(rec3->grec_nsrcs) == 0) //No src list
switch (rec3->grec_type) {
case IGMPV3_CHANGE_TO_EXCLUDE:
    return igmp_report(iph->saddr, rec3->grec_mca);
case IGMPV3_CHANGE_TO_INCLUDE:
    return igmp_leave(iph->saddr, rec3->grec_mca);
}
}
return NF_ACCEPT;
```

```
    case IGMP_HOST_LEAVE_MESSAGE:
return igmp_leave(iph->saddr, igmph->group);
//from IGMP_V2 to delete a group (ehnancement on Time interval timer)
    default:
return NF_ACCEPT;
    }
    return NF_ACCEPT;
}
static inline struct mtable *init_table(uint32_t nmembers)
{
    uint32_t i;
    struct mtable *t;
    t = kzalloc(sizeof(*t) + nmembers * sizeof(t->members[0]), GFP_KERNEL);
    for (i = 0; i < nmembers; i++)
INIT_HLIST_HEAD(&t->members[i]);
    return t;
}
static void delete_list(struct hlist_head *head)
{
    struct xt_mcast *entry;
    struct hlist_node *pos, *temp;
    pos = head->first;
    while (pos != NULL) {
entry = hlist_entry(pos, struct xt_mcast, node);
temp = pos;
pos = NULL; //No pter reshuffle here
pos = temp->next;
kfree(entry);
    }
}
static inline void delete_table(struct mtable *t, uint32_t nsize)
{
    uint32_t i;
    if (t == NULL)
return;
    spin_lock_bh(&mcast_lock);
    for (i = 0; i < nsize; i++)
delete_list(&t->members[i]);
    spin_unlock_bh(&mcast_lock);
    kfree(t);
}
/**OUTGOING traffic only thus client = ip_hrd(skb)->saddr;
```

```
 * mcast_src_addr= ip_hdr(skb)->daddr;
 */
static unsigned int unicast_handler(const struct iphdr *iph)
{
    __be32 mcli_ip = iph->saddr;
    uint32_t hs = mcast_hash(iph->daddr, source_size);
    struct hlist_node *mcli_node;
    struct xt_mcast *mcli = NULL;
    mcli_node = lookup(hs, mcli_ip, msrc_hash);
    if (mcli_node == NULL)
return init_new_entry(mcli, mcli_ip, hs, msrc_hash, source_expire);
    spin_lock_bh(&mcast_lock);
    mcli = hlist_entry(mcli_node, struct xt_mcast, node);
    mcli->timeout = jiffies + source_expire * HZ;
    spin_unlock_bh(&mcast_lock);
    return NF_ACCEPT;
}
static unsigned int mcast_tg4(const struct sk_buff **pskb,
      const struct xt_target_param *par)
{
    const struct sk_buff *skb = *pskb;
    const struct xt_mcast_target_info *info = par->targinfo;
    const struct iphdr *iph = ip_hdr(skb);
    if (info) {
if (info->msrc_size > MGRP_SIZE)
    source_size = info->msrc_size;
if (info->mgrp_size > MGRP_SIZE)
    group_size = info->mgrp_size;
    }
    if (iph == NULL)
return NF_ACCEPT;
    if (ipv4_is_multicast(iph->daddr))
//broadcast also mcast so: skb->pkt_type==PACKET_MULTICAST check??
    {
if (ip_hdr(skb)->protocol == IPPROTO_UDP)
    return mcast_handler4(iph);
//there maybe NF_DROPs
if (ip_hdr(skb)->protocol == IPPROTO_IGMP)
    return igmp_handler(iph, skb);
//always NF_ACCEPT just for snooping
    } else if (ip_hdr(skb)->protocol == IPPROTO_UDP) {
//the heartbeat protocol can be ICMP, or any other so long as we
```

```
// get its ip header to extract msrc.
return unicast_handler(iph);
//this needs to make fast construction of <s,c> pair
    }
    return NF_ACCEPT;
}
static struct xt_target mcast_tg_reg __read_mostly = {
    .name = "MCAST",
    .revision = 0,
    .family = NFPROTO_IPV4,
    .target = mcast_tg4,
    .targetsize = XT_ALIGN(sizeof(struct xt_mcast_target_info)),
    .me = THIS_MODULE,
};
static int __init mcast_tg_init(void)
{
    mgrp_hash = init_table(group_size);
    msrc_hash = init_table(source_size);
    get_random_bytes(&jhash_rnd, sizeof(jhash_rnd));
    return xt_register_target(&mcast_tg_reg);
}
static void __exit mcast_tg_exit(void)
{
    delete_table(mgrp_hash, group_size);
    delete_table(msrc_hash, source_size);
    xt_unregister_target(&mcast_tg_reg);
}
module_init(mcast_tg_init);
module_exit(mcast_tg_exit);
MODULE_LICENSE("GPL");
MODULE_ALIAS("ipt_MCAST");
MODULE_DESCRIPTION("Xtables: Mcast packet filter");
MODULE_AUTHOR("Bewketu Tadilo<xxxx@xxxx.edu.au>");
```

A.2.3. libxt_MCAST.c (for userland interaction to set values).

```
#include <getopt.h>
#include <stdio.h>
#include <string.h>
#include <xtables.h>
#include <linux/netfilter/x_tables.h>
#include "xt_MCAST.h"
static void mcast_tg_help(void)
```

```
{
    printf("MCAST target options:\n"
   "  --multicast-source-size value    expected number of multicast sources\n"
   "  --multicast-group-size value    expected number of multicast groups \n");
}
enum {
    MCAST_OPT_MSRC_SIZE,
    MCAST_OPT_MGRP_SIZE,
};
static const struct option mcast_opts[] = {
    {"multicast-source-size", 1, NULL, MCAST_OPT_MSRC_SIZE},
    {"multicast-group-size", 1, NULL, MCAST_OPT_MGRP_SIZE},
    {.name = NULL},
};
static int mcast_tg_parse(int c, char **argv, int invert,
  unsigned int *flags, const void *entry,
  struct xt_entry_target **target)
{
    struct xt_mcast_target_info *info = (void *) (*target)->data;
    unsigned int num;
    switch (c) {
    case MCAST_OPT_MSRC_SIZE:
if (*flags & (1 << c)) {
    xtables_error(PARAMETER_PROBLEM,
  "MCAST: can't specify `--multicast-source-size' twice");
}
if (!xtables_strtoui(optarg, NULL, &num, 1, UINT32_MAX)) {
    xtables_error(PARAMETER_PROBLEM,
  "Unable to parse `%s' in "
  "`--mulitcast-souce-size'", optarg);
}
info->msrc_size = num;
*flags |= 1 << c;
break;
    case MCAST_OPT_MGRP_SIZE:
if (*flags & (1 << c))
    xtables_error(PARAMETER_PROBLEM,
  "MCAST: can only specify `--multicast-group-size' once");
if (!xtables_strtoui(optarg, NULL, &num, 1, UINT32_MAX)) {
    xtables_error(PARAMETER_PROBLEM,
  "Unable to parse `%s' in "
  "`--mulitcast-souce-size'", optarg);
```

```
}
info->mgrp_size = num;
*flags |= 1 << c;
break;
    default:
return 0;
    }
    return 1;
}
static void mcast_tg_check(unsigned int flags)
{
}
static struct xtables_target mcast_tg_reg = {
    .version = XTABLES_VERSION,
    .name = "MCAST",
    .revision = 0,
    .family = PF_INET,
    .size = XT_ALIGN(sizeof(struct xt_mcast_target_info)),
    .userspacesize = XT_ALIGN(sizeof(struct xt_mcast_target_info)),
    .help = mcast_tg_help,
    .parse = mcast_tg_parse,
    .final_check = mcast_tg_check,
};
static __attribute__ ((constructor))
void mcast_tg_ldr(void)
{
    xtables_register_target(&mcast_tg_reg);
}
```

A.2.4. Scripts.

A.2.4.1. *Makefile.*

```
top_srcdir      := ..
srcdir          := .
prefix          := /usr/
exec_prefix     := ${prefix}
libdir          := /usr/lib
libexecdir      := ${exec_prefix}/libexec
xtlibdir        := ${libexecdir}/xtables
xtables_CFLAGS  :=
CC              := gcc
CCLD            := ${CC}
LCFLAGS          := -g -O2
```

```
LDFLAGS         :=
regular_CFLAGS  := -D_LARGEFILE_SOURCE=1 -D_LARGE_FILES -D_FILE_OFFSET_BITS=64
-D_REENTRANT -Wall -Waggregate-return -Wmissing-declarations
-Wmissing-prototypes -Wredundant-decls -Wshadow -Wstrict-prototypes
-Winline -pipe -DXTABLES_LIBDIR=\"${xtlibdir}\"
kinclude_CFLAGS :=  -I /lib/modules/2.6.25.20-0.1-default/build/include
AM_CFLAGS      := ${regular_CFLAGS} -I${top_srcdir}/include ${xtables_CFLAGS}
${kinclude_CFLAGS}
AM_DEPFLAGS     = -Wp,-MMD,$(@D)/.$(@F).d,-MT,$@
obj-m += compat_xtables.o xt_MCAST.o
all: lib
make -C /lib/modules/‘uname -r‘/build M=‘pwd‘
clean:
make -C /lib/modules/‘uname -r‘/build M=‘pwd‘ clean
rm -rf libxt_MCAST.so
install:
cp *.so  $(xtlibdir)
cp *.ko /lib/modules/‘uname -r‘/extra
lib:  libxt_MCAST.so
lib%.so: lib%.oo
${CCLD} ${AM_LDFLAGS} -shared ${LDFLAGS} -o $@ $<;
lib%.oo: ${srcdir}/lib%.c
${CC} ${AM_DEPFLAGS} ${AM_CFLAGS} -D_INIT=lib$*_init -DPIC -fPIC
    ${LCFLAGS} -o $@ -c $<;
```

A.2.4.2. *iptables rule (test.sh).*

```
#!/bin/sh
IPT=/usr/sbin/iptables
modprobe x_tables
$IPT -F
rmmod  xt_MCAST
rmmod  compat_xtables.ko
insmod compat_xtables.ko
insmod xt_MCAST.ko
$IPT -I OUTPUT -p udp --dport 4444 -j MCAST
#snoop ONLY igmp we don’t care about outgoing Multicast traffic
$IPT -I OUTPUT -p igmp -j MCAST
$IPT -I INPUT -d 224.0.0.0/4 ! -p igmp -j MCAST
#apply policy on incoming mcast traffic which is IGMP (it could be proxied)
```

APPENDIX B

Userland Simulator

To get the following work one needs /linux/list.h which can be dowloaded (by easy google search) or in /usr/src/linux/include/linux/list.h if one has source installed.

- Remove all the prefetch statements
- Remove all the kernel-specific constants
- put containerof implementation in it as follows and can include it in user (as opposed to kenel) application by #include :

```
#define container_of(ptr, type, member) ({ \
const typeof( ((type *)0)->member ) *__mptr = (ptr);
\
(type *)( (char *)__mptr - offsetof(type,member) );})
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

B.1. Header File

```
#ifndef _MTEST_H
#define _MTEST_H
#include <stdint.h>
struct xt_mcast {
    struct hlist_node node;
    __be32 ip;
    unsigned long timeout;
    //unsigned long  expire; //this is in jiffies (allowed per group)
};
struct mtable {
    struct hlist_head members[0];
};
#define time_after(a,b) \
 ((long)(b) - (long)(a) < 0)
#define time_before(a,b) time_after(b,a)
static unsigned int
init_new_entry(struct xt_mcast *entry, __be32 ip, uint32_t h,
       struct mtable *t, uint16_t expire);
static inline uint32_t mcast_hash(__be32 name, unsigned int size);
static struct hlist_node *lookup(__be32 hkey, __be32 ip, struct mtable *t);
```

```
static struct hlist_node *get_headnode(struct mtable *t, uint32_t h);
static unsigned int igmp_report(__be32 cli, __be32 grp);
static unsigned int mcast_search(__be32 msrs_ip, __be32 mgrp_ip);
static unsigned igmp_leave(__be32 cli, __be32 grp);
static inline struct mtable *init_table(uint32_t nmembers);
static void delete_list(struct hlist_head *head);
static inline void delete_table(struct mtable *t, uint32_t nsize);
static unsigned int unicast_handler(__be32, __be32);
#endif /*_MTEST_H*/
```

B.2. Main.c

```
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <linux/types.h>
#include <stdlib.h>
#include <netinet/ip.h>
#include <netinet/in.h>
#include <netinet/ip6.h>
#include <getopt.h>
#include "list.h"
#include "mtest.h"
#include <linux/netfilter.h>
#include "jhash.h"
#include <sys/time.h>
#define MSRC_SIZE    20 //src->cli_head_nodes
#define MGRP_SIZE    20 //grp->cli_head_nodes
#define MGRP_EXPIRE   120
//2-minutes about the same timetime router will issue Group Query.
#define MCAST_GRP_ID   10
#define ARRAR_SIZE(array) (sizeof(array)/sizeof(array[0]))
static void print_hash_list(struct hlist_head head);
void print_usage(FILE * stream, char **argv);
static inline void simulate(int, char **);
static struct mtable *msrc_hash;
//client list indexed by hash(src)->cli_ip
static struct mtable *mgrp_hash;
//client list indexed by hash(gid)->cli_ip
static unsigned int source_size = MSRC_SIZE;
static unsigned int group_size = MGRP_SIZE;
static unsigned int source_expire = 2 * 60 * 60;
//2-hours
```

```
static unsigned int *rand_src, *rand_grp;
static unsigned int *rand_cli;
static struct timeval now;
static u_int32_t jhash_rnd;
// all insert functions are f(keyip,cli)  format
int main(int argc, char **argv)
{
    int i;
    if (argc < 2)
print_usage(stderr, argv);
    jhash_rnd = time(NULL);
    mgrp_hash = init_table(group_size);
    msrc_hash = init_table(source_size);
    simulate(argc, argv);
    // for(i = 0; i < 10; i++)
    //    igmp_report(i,MCAST_GRP_ID);
    //print_hash_list(mgrp_hash->members[mcast_hash(MCAST_GRP_ID,group_size)]);
    //unicast_handler(10,12);
    //mcast_search(MCAST_GRP_ID,9);
    delete_table(mgrp_hash, group_size);
    delete_table(msrc_hash, source_size);
    return 0;
}
static struct option long_opt[] = {
    {"ngrp", 1, 0, 'g'},
    {"ncli", 1, 0, 'c'},
    {"nsrc", 1, 0, 's'},
    {"percentag", 1, 0, 'r'},
    {NULL, 0, NULL, 0}
};
static char *short_opt = "g:c:s:r:";
static inline void simulate(int argc, char **argv)
{
//15-groups
//15-clients
    int i;
    srand(time(NULL));
    int c, sizec, sizes, sizeg, per;
    long mtime, seconds, useconds;
    struct timeval start;
    struct timeval end;
    long average = 0;
```

```
    while (1) {
c = getopt_long(argc, argv, short_opt, long_opt, NULL);
if (c == -1)
    break;
switch (c) {
case 'c':{
sizec = atoi(optarg);
rand_cli = malloc(sizec * sizeof(int *));
for (i = 0; i < sizec; i++)
    rand_cli[i] = rand();
    }
    break;
case 's':{
sizes = atoi(optarg);
rand_src = malloc(sizes * sizeof(int *));
for (i = 0; i < sizes; i++)
    rand_src[i] = rand();
    }
    break;
case 'g':{
sizeg = atoi(optarg);
rand_grp = malloc(sizes * sizeof(int *));
for (i = 0; i < sizes; i++)
    rand_grp[i] = rand();
    }
    break;
case 'r':
    per = atoi(optarg);
    break;
default:
    print_usage(stderr, argv);
}
    }
    for (i = 0; i < sizec; i++) {
igmp_report(rand_cli[i], rand_grp[rand() % sizeg]);
unicast_handler(rand_cli[i], rand_src[rand() % sizes]);
    }
    //again here now we simulate a random source contacts our network
    //here we measure how long does it take
//to block it to let it go with a relation to the our memory
    //do 300 random searches printing the raw time it took.
    for (i = 0; i < 300; i++) {
```

```
gettimeofday(&start, NULL);
mcast_search(rand_src[rand() % sizes], rand_grp[rand() % sizeg]);
gettimeofday(&end, NULL);
//seconds = end.tv_sec - start.tv_sec;
useconds = end.tv_usec - start.tv_usec;
average += useconds;
//  mtime=  (seconds *1000 + useconds/1000.0)+ 0.5;
//printf("Elapsed: %ld us\n",useconds);
    }
    printf("Averag elapsed %ld useconds\n", average / 300);
}
static void print_hash_list(struct hlist_head head)
// this is array of hashtables
{
    struct xt_mcast *entry;
    struct hlist_node *pos;
    hlist_for_each_entry(entry, pos, &head, node)
printf("ip: %u\n", entry->ip);
}
static unsigned int
init_new_entry(struct xt_mcast *entry, __be32 ip, uint32_t h,
       struct mtable *t, uint16_t expire)
{
    struct hlist_node *pos, *temp;
    struct xt_mcast *temp_entry;
    struct xt_mcast *place;
    struct hlist_node *ref = get_headnode(t, h);
    hlist_for_each_entry_safe(place, pos, temp, &t->members[h], node) {
gettimeofday(&now, NULL);
if (place != NULL && time_after(now.tv_sec, place->timeout)) {
    entry = place;
    pos = pos->next;
//don't delete pos- just advance it
    while (pos != NULL && pos != ref) {
temp_entry = hlist_entry(pos, struct xt_mcast, node);
free(temp_entry);
temp = pos;
pos = NULL; //No pter reshuffle here
pos = temp->next;
    }
    break;
}
```

```
    }
    if (entry == NULL)
entry = (struct xt_mcast *) malloc(sizeof(struct xt_mcast));
    entry->ip = ip;
    gettimeofday(&now, NULL);
    entry->timeout = now.tv_sec + expire;
    hlist_add_head(&entry->node, &t->members[h]);
    return NF_ACCEPT;
}
static inline uint32_t mcast_hash(__be32 name, unsigned int size)
{
    return jhash_1word(name, jhash_rnd) & (size - 1);
}
static struct hlist_node *lookup(__be32 hkey, __be32 ip, struct mtable *t)
{
    struct xt_mcast *entry;
    struct hlist_node *n;
    struct hlist_node *temp = n;
    //use of the mutex should be appropriate here-Noisy Kernel unless new!!
    hlist_for_each_entry_safe(entry, n, temp, &t->members[hkey], node)
if (ip == entry->ip)
return n;
    return NULL;
}
static struct hlist_node *get_headnode(struct mtable *t, uint32_t h)
{
    return t->members[h].first;
}
static unsigned int igmp_report(__be32 cli, __be32 grp)
{
    struct xt_mcast *entry = NULL, *head_entry;
//aka head
    uint32_t h = mcast_hash(grp, group_size);
    struct hlist_node *head_node = get_headnode(mgrp_hash, h);
    struct hlist_node *entry_node;
    if (head_node == NULL)
return init_new_entry(entry, cli, h, mgrp_hash, MGRP_EXPIRE);
    //go fish
    entry_node = lookup(h, cli, mgrp_hash);
//here there is grp alrdy but is the client in it.
    if (entry_node == NULL) {
return init_new_entry(entry, cli, h, mgrp_hash, MGRP_EXPIRE);
```

```
    }
    //good the client is in the grp.
    head_entry = hlist_entry(head_node, struct xt_mcast, node);
    entry = hlist_entry(entry_node, struct xt_mcast, node);
    gettimeofday(&now, NULL);
    entry->timeout = now.tv_sec + MGRP_EXPIRE;
    //we need to also update the same entry in the source list
    if (entry->ip == head_entry->ip)
return NF_ACCEPT;
    //it the head. don't do anything
    hlist_del(entry_node); //shuffle and fix
    hlist_add_head(&entry->node, &mgrp_hash->members[h]);
    //we want this node to be head so we need to del:
    //NOTE: when deleting a grp if it the head pointer
    printf("IGMP REPORT END\n");
    return NF_ACCEPT;
}
static unsigned int mcast_search(__be32 msrc_ip, __be32 mgrp_ip)
{
    uint32_t hs = mcast_hash(msrc_ip, source_size);
//h(msrc_ip)->cli_ip
    uint32_t hg = mcast_hash(mgrp_ip, group_size);
//h(mgrp_ip)->cli_ip
    struct xt_mcast *msrc2cli, *mgrp2cli;
    struct hlist_node *n, *n2;
    //check if there exists any client(s) that are mapped to msrc_ip
    //while looking hash them and check if they have
    hlist_for_each_entry(msrc2cli, n, &msrc_hash->members[hs], node) {
hlist_for_each_entry(mgrp2cli, n2, &mgrp_hash->members[hg], node) {
    if (msrc2cli->ip == mgrp2cli->ip) { //success
return NF_ACCEPT;
//remember we won't update anything here (
    }
}
    }
    return NF_DROP;
}
static unsigned igmp_leave(__be32 cli, __be32 grp)
{
    uint32_t h = mcast_hash(grp, group_size);
    struct hlist_node *entry_node;
    struct xt_mcast *entry, *head_entry; //aka head
```

```
    struct hlist_node *head_node = get_headnode(mgrp_hash, h);
/* entry is NULL => "I am leaving this grp"-
yet there is no such grp or doesn't have me
 * shouldn't be possible */
    if (head_node == NULL)
return NF_ACCEPT;
    entry_node = lookup(h, cli, mgrp_hash);
    if (entry_node == NULL) //weird
return NF_ACCEPT;
    entry = hlist_entry(entry_node, struct xt_mcast, node);
    head_entry = hlist_entry(head_node, struct xt_mcast, node);
    entry = hlist_entry(entry_node, struct xt_mcast, node);
    if (head_entry->ip == entry->ip && entry_node->next != NULL)
//promote the successor to be head
hlist_add_head(entry_node->next, &mgrp_hash->members[h]);
    //else it is the last entry
    hlist_del(entry_node);
    free(entry);
    return NF_ACCEPT;
}
static inline struct mtable *init_table(uint32_t nmembers)
{
    uint32_t i;
    struct mtable *t;
    t = malloc(sizeof(*t) + nmembers * sizeof(t->members[0]));
    for (i = 0; i < nmembers; i++)
INIT_HLIST_HEAD(&t->members[i]);
    return t;
}
static void delete_list(struct hlist_head *head)
{
    struct xt_mcast *entry;
    struct hlist_node *pos, *temp;
    pos = head->first;
    while (pos != NULL) {
entry = hlist_entry(pos, struct xt_mcast, node);
temp = pos;
pos = NULL; //No pter reshuffle here
pos = temp->next;
free(entry);
    }
}
```

```
static inline void delete_table(struct mtable *t, uint32_t nsize)
{
    uint32_t i;
    if (t == NULL)
return;
    for (i = 0; i < nsize; i++)
delete_list(&t->members[i]);
    free(t);
}
static unsigned int unicast_handler(__be32 mcli_ip, __be32 msrc_ip)
{
    uint32_t hs = mcast_hash(msrc_ip, source_size);
    struct hlist_node *mcli_node;
    struct xt_mcast *mcli = NULL;
    mcli_node = lookup(hs, mcli_ip, msrc_hash);
    if (mcli_node == NULL)
return init_new_entry(mcli, mcli_ip, hs, msrc_hash, source_expire);
    mcli = hlist_entry(mcli_node, struct xt_mcast, node);
    gettimeofday(&now, NULL);
    mcli->timeout = now.tv_sec + source_expire;
    return NF_ACCEPT;
}
void print_usage(FILE * stream, char **argv)
{
    fprintf(stream, "\n" "Usage: %s -c <ncli> " "-s <nsrc> \n", argv[0]);
    fprintf(stream,
    "\n"
    "Opitions:\n"
    "       -c, --ncli     number of clients\n"
    "       -s, --nsrc       expected number of sources\n"
    "       -r, --percentage multicast connection percentage (from the above)\n"
    "\n");
    exit(0);
}
```

Bibliography

[1] Brad Cain, Steve Deering, Isidor Kouvelas, Bill Fenner, and Ajit Thyagarajan. Internet Group Management Protocol, Version 3. Request For Comments (RFC) 3376, October 2002. http://www.ietf.org/rfc.html. 2.2.3

[2] S. Deering. Host Extensions for IP Multicasting. Request For Comments (RFC) 1112, August 1989. http://www.ietf.org/rfc.html. 1

[3] W. Fenner. Internet Group Management Protocol, Version 2. Request For Comments (RFC) 2236, November 1997. http://www.ietf.org/rfc.html. 2.2.2

[4] Steve Deering, Bill Fenner, and Brian Haberman. Multicast Listener Discovery (MLD) for IPv6. Request For Comments (RFC) 2710, October 1999. http://www.ietf.org/rfc.html.

[5] S.DEERING and DAVID R. CHERITON, ACM Transactions on Computer Systems, Vol. 8, No. 2, May 1990,

[6] Radia Perlman, Interconnections: Bridges and Routers, Addison-Wesley Professional Computing Series 1999. 2.3

[7] W.R. Stevens Unix Network Programming,Volume 1, Third Edition, Prentice Hall 2004. 2.3

[8] S. Li, V. Sivaraman, A. Krumm-Heller, C. Russell, "A Dynamic Stateful Multicast Firewall", IEEE International Conference on Communications (ICC), Glasgow, Scotland, Jun 2007. 1.3

[9] H. Holbrook, B. Cain “Source-Specific Multicast for IP”, draft-ietf-ssm-arch-07, 4 Oct 2005 2.4

[10] SSM-Mapping, http://www.cisco.com/en/US/docs/ios/ipmulti/configuration/guide/imc_ssm_mapping.pdf 2.5.2.1

[11] Paul Judge and Mostafa Ammar, Security Issues and Solutions in Multicast Content Distribution: A Survey, IEEE Network. January/February 2003. 2.5.1

[12] L. Oria. Approaches to Multicast over Firewalls: An Analy- sis. Technical Report HPL-IRI-1999-004, HP Labs, Aug 1999. http://www.hpl.hp.com/techreports/1999/HPL-IRI-1999-004.html. 2.5.2

[13] IETF Multicast Security (MSEC) working group. http://www.ietf.org/html.charters/msec-charter.html. 2.5.1

[14] Shai Rubin, David Bernstein, and Michael Rodeh, IBM Research Lab in Haifa and Computer Science Department – Technion (Israel Institute of Technology)- 1999 http://pages.cs.wisc.edu/~shai/CC99.pdf 7

[15] Paul E. McKenney. Using RCU in the Linux 2.5 kernel. Linux Journal, 1(114):18–26, October 2003. 7

[16] Bob Jenkins,A hash function for hash Table lookup,http://www.burtleburtle.net/bob/hash/doobs.html 3.2

[17] Paul Russel and Harald Welte, “Netfilter Hacking How-to”: http://www .netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.txt.

[18] B. W. Kernighan and R. Pike, The Practice of Programming, Addison-Wesley, 1999. 3.2

[19] Ulrich Drepper,What Every Programmer Should Know About Memory, Red Hat, Inc. November 21, 2007.

4.3