

Netfilter:

Making large iptables rulesets scale

Netfilter Workshop 2008
d.29/9-2008

by

Jesper Dangaard Brouer <jdb@comx.dk>
Master of Computer Science
ComX Networks A/S

The logo for ComX Networks, featuring the word "comx" in a lowercase, sans-serif font. The letters "com" are in a light gray color, and the letter "x" is in a bright orange color.

ComX Networks A/S

Who am I

- Name: Jesper Dangaard Brouer
 - Edu: Computer Science for Uni. Copenhagen
 - Focus on Network, Dist. sys and OS
 - Linux user since 1996, professional since 1998
 - Sysadm, Developer, Embedded
 - OpenSource projects
 - Author of
 - ADSL-optmizer
 - CPAN IPTables::libiptc
 - Patches accepted into
 - Kernel, iproute2 and iptables

Presentation overview

- You will learn:
 - About a Danish ISPs extreme use of iptables
 - How to avoid bad routing performance
 - *Traffic categorization is performance key*
 - How iptables rulesets are processed in userspace
 - How to use userspace processing as an advantage
 - *Improvements to make iptables scale*

ComX Networks A/S

- I work for ComX Networks A/S
 - Danish Fiber Broadband Provider
 - variety of services (TV, IPTV, VoIP, Internet)
- This talk is about our Internet product
 - Netfilter is a core component:
 - Basic Access Control
 - Bandwidth Control
 - Personal Firewall

Physical surroundings

- ComX delivers fiber based solutions
 - Our primary customers are apartment buildings.
 - Ring based network topology with POPs (Point Of Presence)
 - POPs have fiber strings to apartment buildings
 - CPE box in apartment performs
 - service separation into VLANs

The Linux box

- The iptables box(es), this talk is all about
 - placed at each POP (near the core routers)
 - high-end server PC, with *only two netcards*
- Internet traffic:
 - from several apartment buildings,
 - layer2 terminated via VLANs on one netcard,
 - routed out the other.
- Cost efficient
 - but *needs to scale to a large number of customers*
 - goal is to scale to 5000 customers per machine

Issues and limitations

- First generation solution was in production.
 - business grew and customers where added;
 - several scalability issues arose
- The two primary were:
 - Routing performance reduced (20 kpps)
 - Rule changes where slow

I was hired to rethink the system

Overview

- Presentation split into two subjects
 - 1) Routing performance
 - Solved using effective traffic categorization
 - 2) Slow rule changes
 - Solved by modifying iptables to use binary search

Issue: Bad route performance

- The first generation solution,
 - naive approach: long list of rules in a single chain
- Routing performance degradation problem:
 - It all comes down to traffic categorizing
 - binding packets to a customer
 - where a customer can have several IP-addresses

Need to find a scalable categorization mechanism

Existing solutions

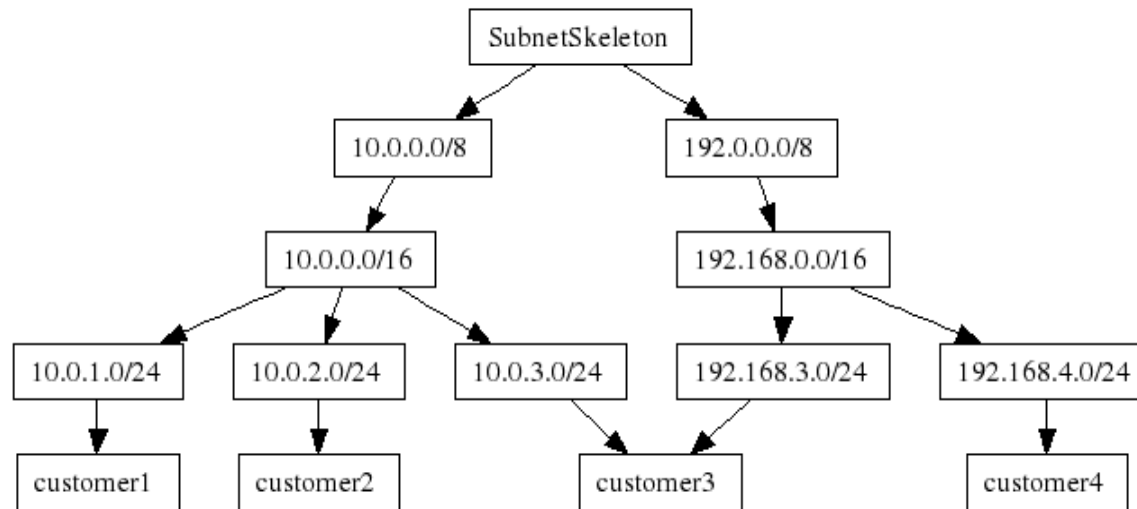
- Looking for existing solutions
 - for solving the categorization task
- Ended up using standard iptables chains
- **nf-hipac**, universal solution,
 - Optimize ruleset for memory lookups per packet
 - Did not work with current kernels
- **ipset**
 - Sets of IP, can be matched, given action

The categorization tasks

- With the kind of categorization needed,
 - why did I ended up using standard iptables chains?
 - **Access Control**
 - simple open/close solution
 - could use ipset
 - **Bandwidth Control**
 - requires an individual shaper per customer
 - cannot use ipset
 - **Personal firewall**
 - most complicated: individual set of rules per customer
 - cannot use ipset

Solution: SubnetSkeleton

- The solution was to build a search tree;
 - for IP-addresses, based on subnet partitioning,
 - using standard iptables chains and jump rules.

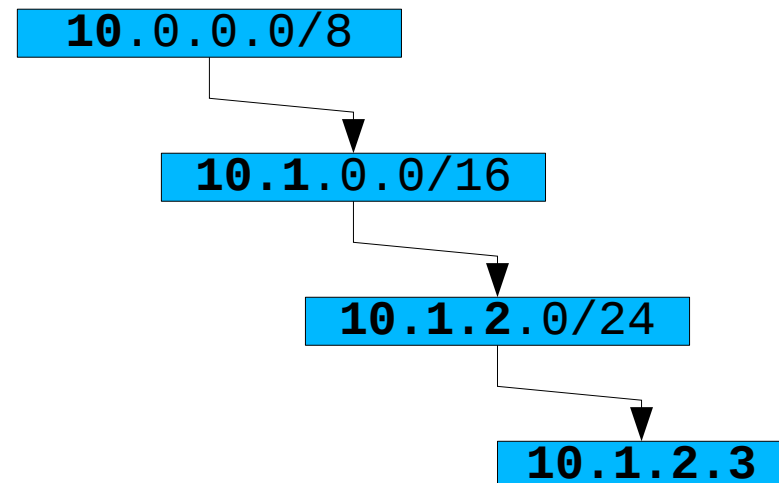


SubnetSkeleton: Algorithm

- Algorithm, predefined partitioning of IP space;
 - based on a user-defined *list of CIDR prefixes*
 - Depth of tree, determined by CIDR list length.
 - Max number of children, bits between CIDRs (2^n)
- Creates tree by bit masking the IP with the CIDR list

– Example:

- CIDR list = [/8, /16, /24]
- IP: 10.1.2.3

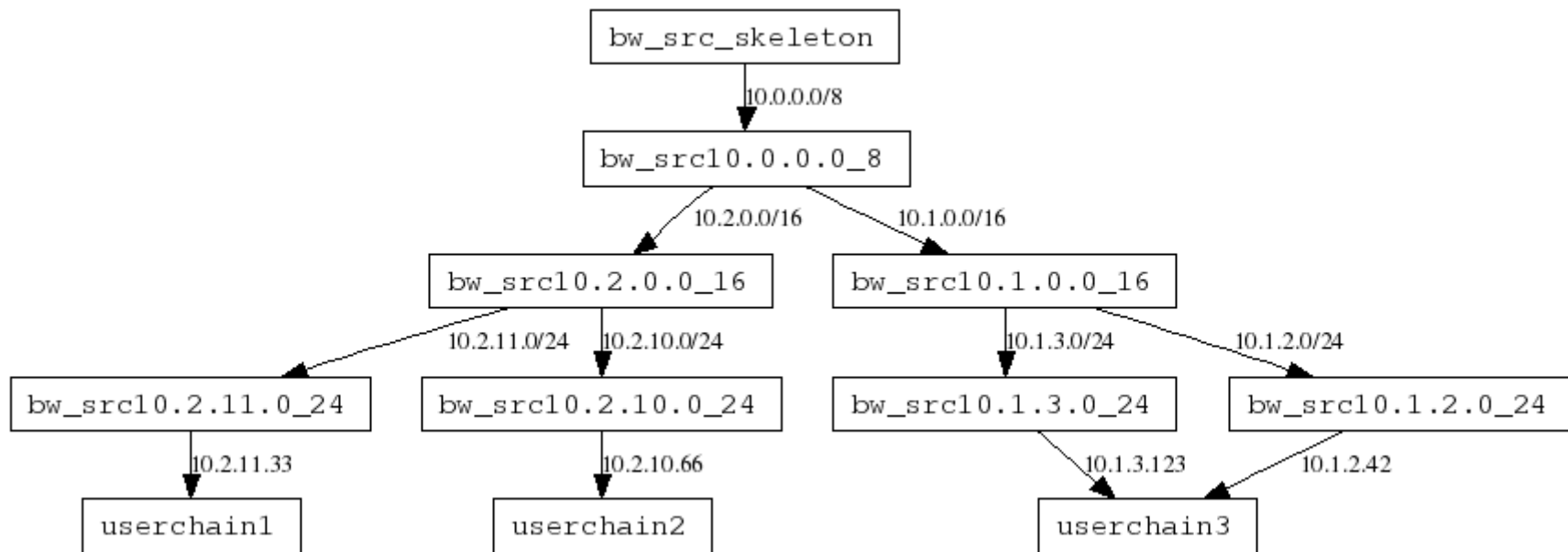


SubnetSkeleton: CIDR partitioning

- Choosing CIDR list is essential.
 - Base it on IP-space that needs to be covered.
 - E.g. our IP-address space, limited to AS number
 - AS31661 = 156.672 IPs.
 - Largest subnet we announce is a /16.
 - CIDR list: [8, 18, 20, 22, 24, 26, 28]
 - /8 needed as our subnets vary on first byte,
 - "0-8", $2^8=256$ children, but only 4 different subnets
 - Between "8-18": $2^{10} = \text{Max } 1024$ children.
 - But know /16 ($2^2=4$)
 - Between, rest 2 bits, thus max 4 children in nodes.
 - Last, "28-32": ($2^4=16$) max 16 direct IP matches.

SubnetSkeleton: iptables

- Expressing the tree using iptables:
 - Each *node* in the tree is an iptables **chain**.
 - *child* pointers in a *node* are **jump** rules.
 - A *leaf* has IP specific jump rules to a user-defined chain
 - *leafs* are allowed to jump to the same user-defined chain
 - *children* (**jump** rules) are processed linearly, in **chain**



Perl - IPTables::SubnetSkeleton

```
#!/usr/bin/perl
use IPTables::SubnetSkeleton;

my @CIDR = (8, 16, 24); # prefix list

my $name = "bw";      # Shortname for bandwidth
my $table = "mangle"; # Use "mangle" table

my $subnet_src = IPTables::SubnetSkeleton::new("$name", "src", $table, @CIDR);

# Connect subnet skeleton to build-in chain "FORWARD"
$subnet_src->connect_to("FORWARD");

# Insert IP's to match into the tree
$subnet_src->insert_element("10.2.11.33", "userchain1");
$subnet_src->insert_element("10.2.10.66", "userchain2");
$subnet_src->insert_element("10.1.2.42", "userchain3");
$subnet_src->insert_element("10.1.3.123", "userchain3");

# Remember to commit the ruleset to kernel
$subnet_src->iptables_commit();
```

Full routing performance achieved

- Full route performance achieved
 - *When using SubnetSkeleton*
 - HTB shaper seems to scale well
 - Better conntrack locking in 2.6.25,
 - reduced cpu load to half, Thanks Patrick McHardy!
 - Parameter tuning
 - Increase route cache
 - Increase conntrack entries
 - remember conntrack hash bucket size (/sys/module/nf_conntrack/parameters/hashsize)
 - Adjust arp/neighbor size and thresholds

Back to subject:
Slow ruleset changes

Issue: iptables command slow

- The next scalability issue: Rule changes slow!
 - Rebuilding the entire ruleset could take hours
- Discover *how iptables works*:
 - Entire ruleset copied to userspace
 - After possibly multiple changes, copied back to kernel
- Performed by a IPTables Cache library "libiptc"
 - iptables.c is a command line parser using this library
- Profiling: identified *first* scalability issue
 - *Initial ruleset parsing*, during “pull-out”
 - Could postpone fix...

Take advantage of libiptc

- Take advantage of pull-out and commit system
 1. Pull-out ruleset (*one initial ruleset parsing penalty*)
 2. Make all modification needed
 3. Commit ruleset (to kernel)
 - This is how ***iptables-restore*** works
- Extra bonus:
 - Several rule changes appear atomic
 - Update all rules related to a customer at once
 - No need for temp chains and renaming

Perl - IPTables::libiptc

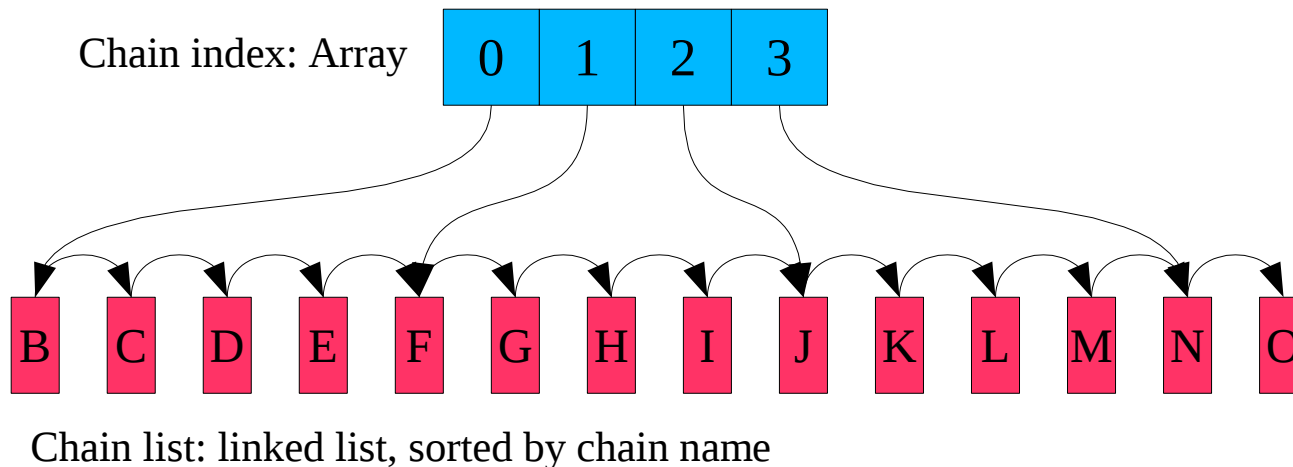
- Cannot use iptables-restore/save
 - SubnetSkeleton must have is_chain() test function
- Created CPAN IPTables::libiptc
 - Chains: Direct libiptc calls
 - Rules: Command like interface via iptables.c linking
 - iptables extensions available on system, dynamic loaded
 - No need to maintain or port iptables extensions
 - Remember to commit()
- Using this module
 - I could postponed fixing "initial ruleset parsing"

Next scalability issue: Chain lookup

- Slow chain name lookup
 - `is_chain()` testing (internal `iptcc_find_label()`)
 - Cause by: linearly list search with `strcmp()`
- Affects: almost everything
 - Rule create, delete, even listing.
 - Multiple rule changes, eg. `iptables-restore`, `SubnetSkeleton`
- Rule listing (`iptables -nL`) with 50k chains:
 - Takes approx 5 minutes!
 - After my fix: reduced to 0.5 sec.

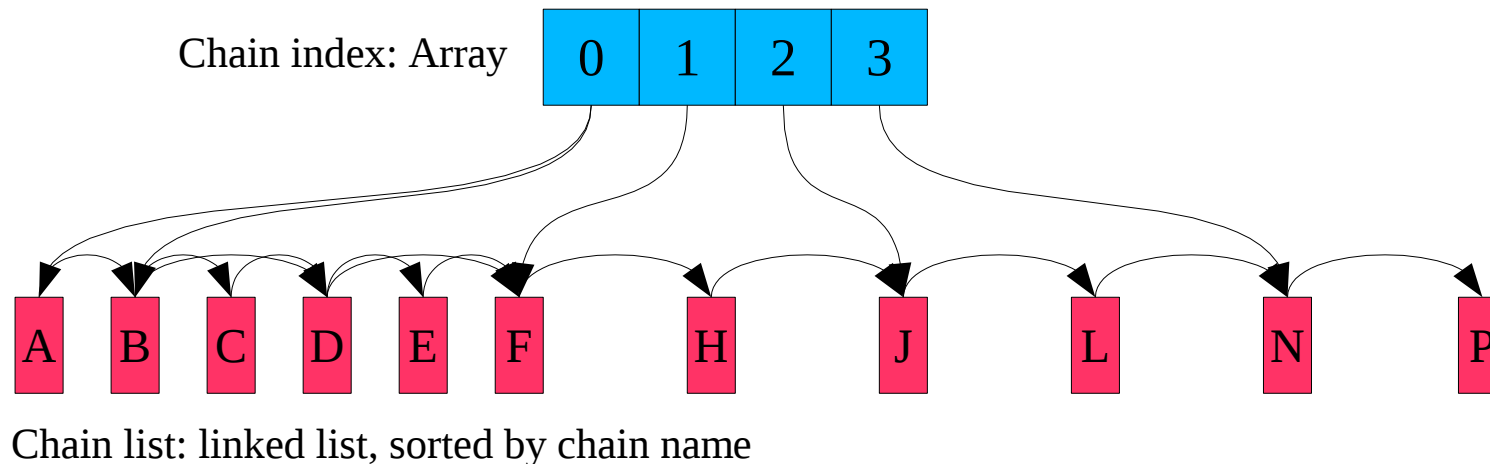
Chains lookup: Solution

- Solution: binary search on chain names
 - Important property chain list is sorted by name
 - Keep original linked list data structure
- New data structure: "Chain index"
 - Array with pointers into linked list with a given spacing (40)
- Result: better starting points when searching the linked list



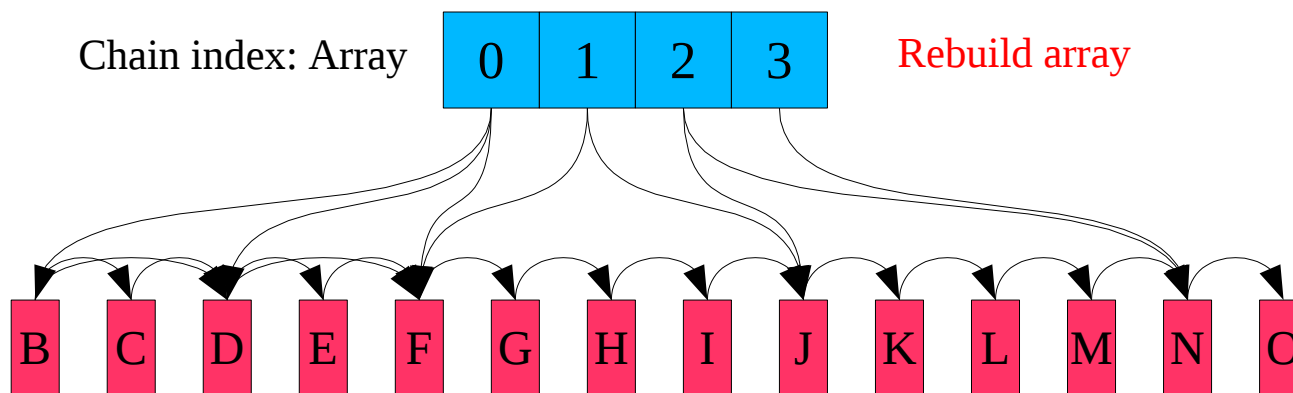
Chain index: Insert chain

- Handle: Inserting/creating new chains
 - Inserting don't change correctness of chain index
 - only cause longer lists
 - rebuild after threshold inserts (355)
 - Inserting before first element is special



Chain index: Delete chain

- Handle: deletion of chains
 - Delete chain *not* pointed to by chain index, no effect
 - Delete chain pointed to by chain index, possible rebuild
 - Replace index pointer with next pointer
 - Only if next pointer not part of chain index



Chain list: linked list, sorted by chain name

Solving: Initial ruleset parsing

- Back to fixing "initial ruleset parsing".
 - Did have a fix, but was not 64-bit compliant (2007-11-26)
- Problem: Resolving jump rules is slow
 - For each: Jump Rule
 - Do a linearly, offset based, search of chain list
- Solution:
 - Reuse binary search algorithm and data structure
 - Realize chain list are both sorted by name and offsets
 - Ruleset from kernel already sorted

Summary: Load time

• Personal firewall

- Reload all rules on a production machine
 - Chains: 5789
 - Rules: 22827

Number of calls 74659
Total time used 1.92 sec
Average per call 0.00002567 sec

| action | calls | time | per call |
|----------------|--------------|-------------------|-------------------|
| set_policy | 1 | 0.00007701 | 0.00007701 |
| append_rule | 8399 | 0.49619532 | 0.00005908 |
| insert_rule | 4463 | 0.24729586 | 0.00005541 |
| flush_entries | 4726 | 0.03449988 | 0.00000730 |
| init | 1 | 0.04638195 | 0.04638195 |
| commit | 1 | 0.08120894 | 0.08120894 |
| list_rules_IPs | 1181 | 0.02705002 | 0.00002290 |
| is_chain | 46965 | 0.37487888 | 0.00000798 |
| delete_rule | 8922 | 0.60892868 | 0.00006825 |
| Sum | 74659 | 1.91651654 | sec |

Total time entire script 23.72 sec

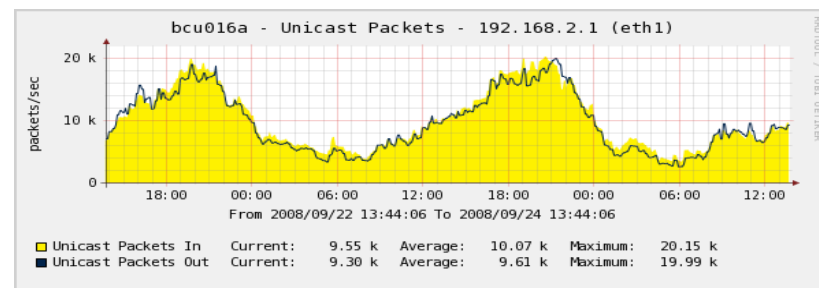
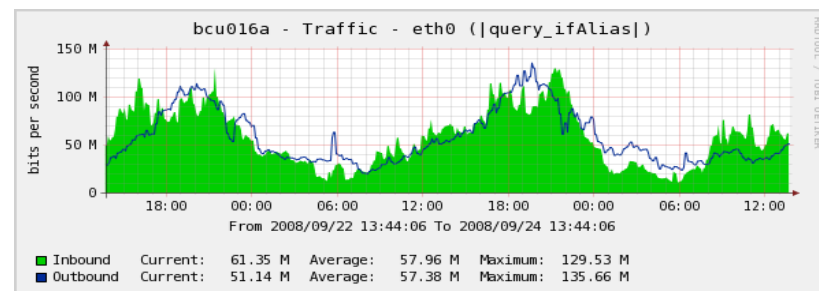
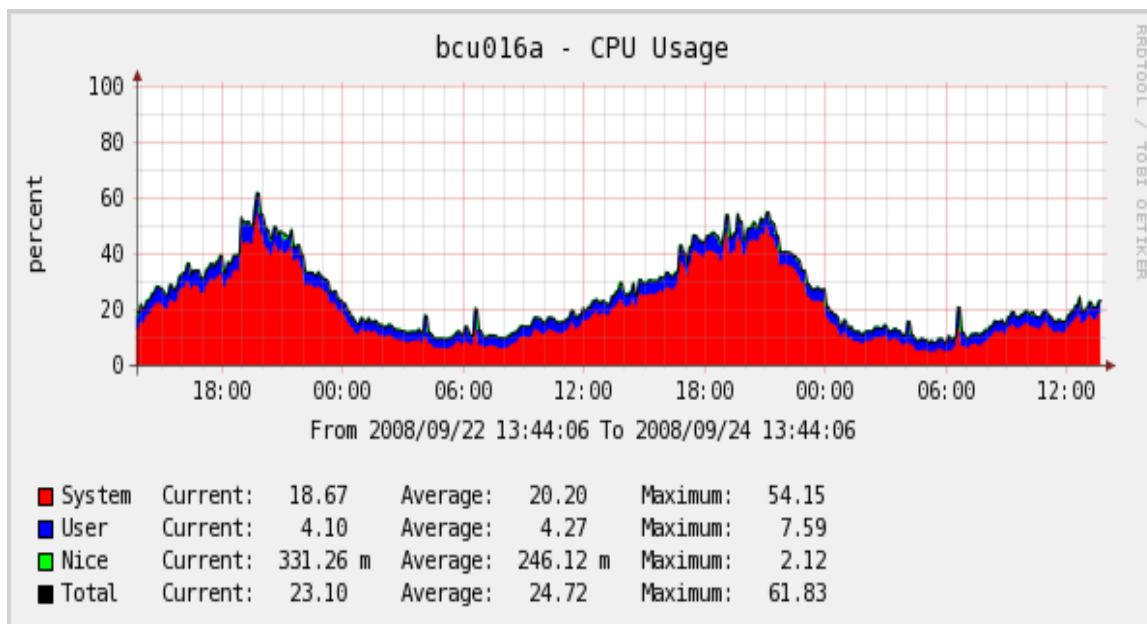
Machine with the most customers, has in filter table
Chains: 9827 Rules:36532

Summary: Open Source

- Open Source Status
 - Chain lookup fix
 - In iptables version 1.4.1
 - 50k chains, listing 5 min -> 0.5 sec
 - Initial ruleset parsing fix
 - In iptables version 1.4.2-rc1
 - Production, reached 10 sec -> 0.046 sec
 - IPTables::libiptc
 - Released on CPAN
 - IPTables::SubnetSkeleton
 - Available via <http://people.netfilter.org/hawk/>

Summary: Goal reached?

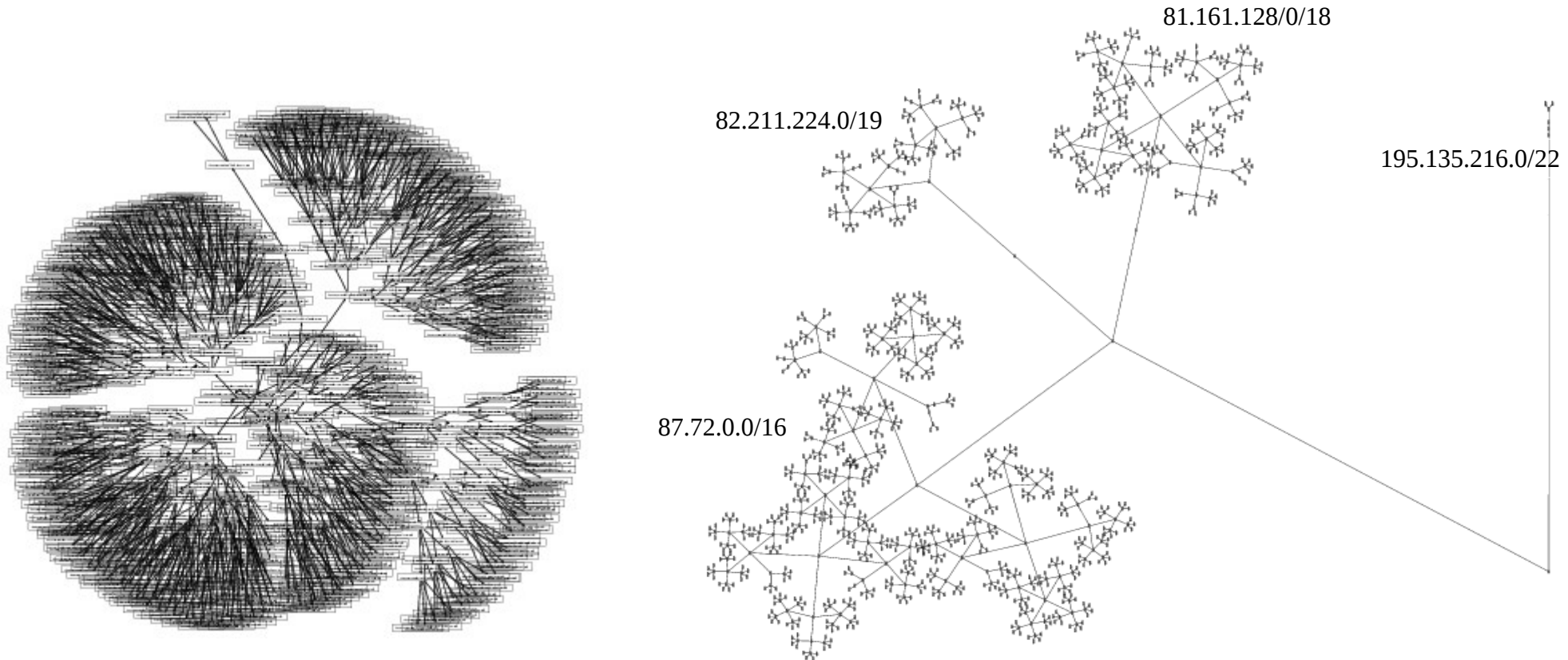
- Goal of 5000 equipment,
 - Production, reached 3400
 - CPU load 30% average, 62% in peek.
 - CPU Xeon (Hyperthread) 3.2 Ghz, 1MB cache
 - In filter table Chains: 9827 Rules: 36532



The End

Goodbye

and thank you for accepting the patches...

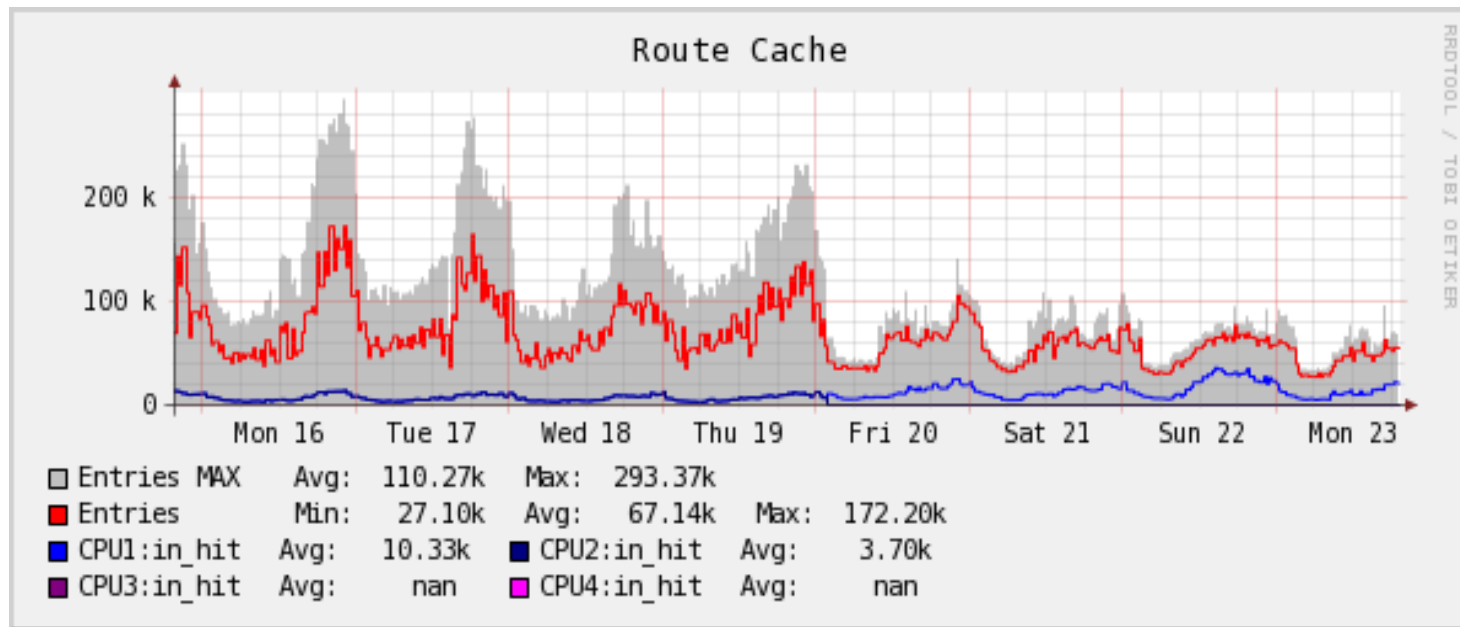


Extra slides

- Bonus slides
 - if time permits
 - or funny questions arise

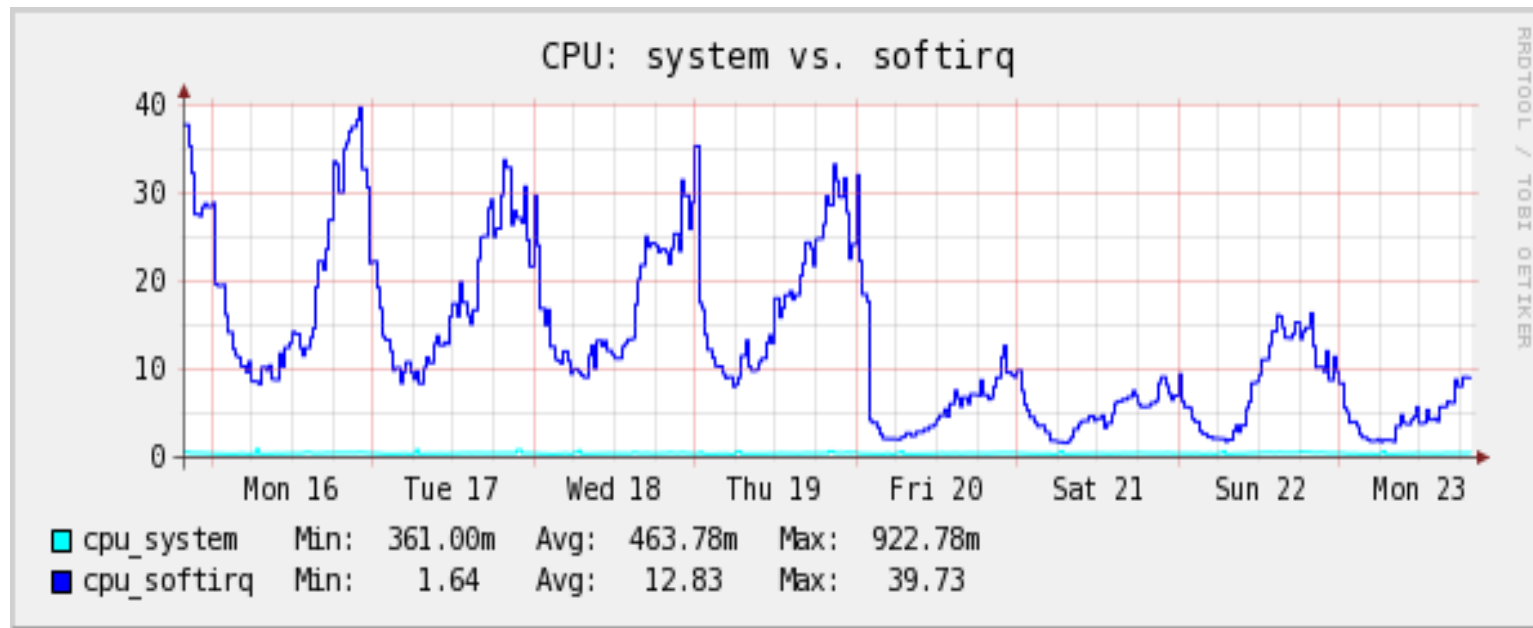
Route cache perf

- Improved route cache
 - Kernel 2.6.15 --> 2.6.25



CPU util softirq

- Softirq CPU usage dropped
 - Kernel 2.6.15 --> 2.6.25
 - Patrick McHardy, improved conntrack locking



More libiptc stats

- Machine with the most customers,
 - Customers:2105 Equipment: 3477
 - In filter table Chains: 9827 Rules: 36532
 - In mangle table Chains: 2770 Rules:14275
 - “Init” time: 0.10719919s
 - “is_chain” time: 0.00001473s

BSD pf firewalling

- My *limited* knowledge of
 - Open/FreeBSD's firewall facility: pf (packet filter)
 - Don't have chains with rules like iptables: Uses one list/chain
 - To compensate, they have an “ipset” like facility called “tables”
 - Quite smart using a radix tree.
 - Has a basic ruleset-optimizer, performs four tasks:
 - 1.remove duplicate rules
 - 2.remove rules that are a subset of another rule
 - 3.combine multiple rules into a table when advantageous
 - 4.re-order the rules to improve evaluation performance
- Don't think pf would solve my categorization needs
 - I could not use “ipset”, for the same reasons cannot use pf “tables”