XDP Virtual Server

An eBPF Load Balancer library

David Coles, February 2026

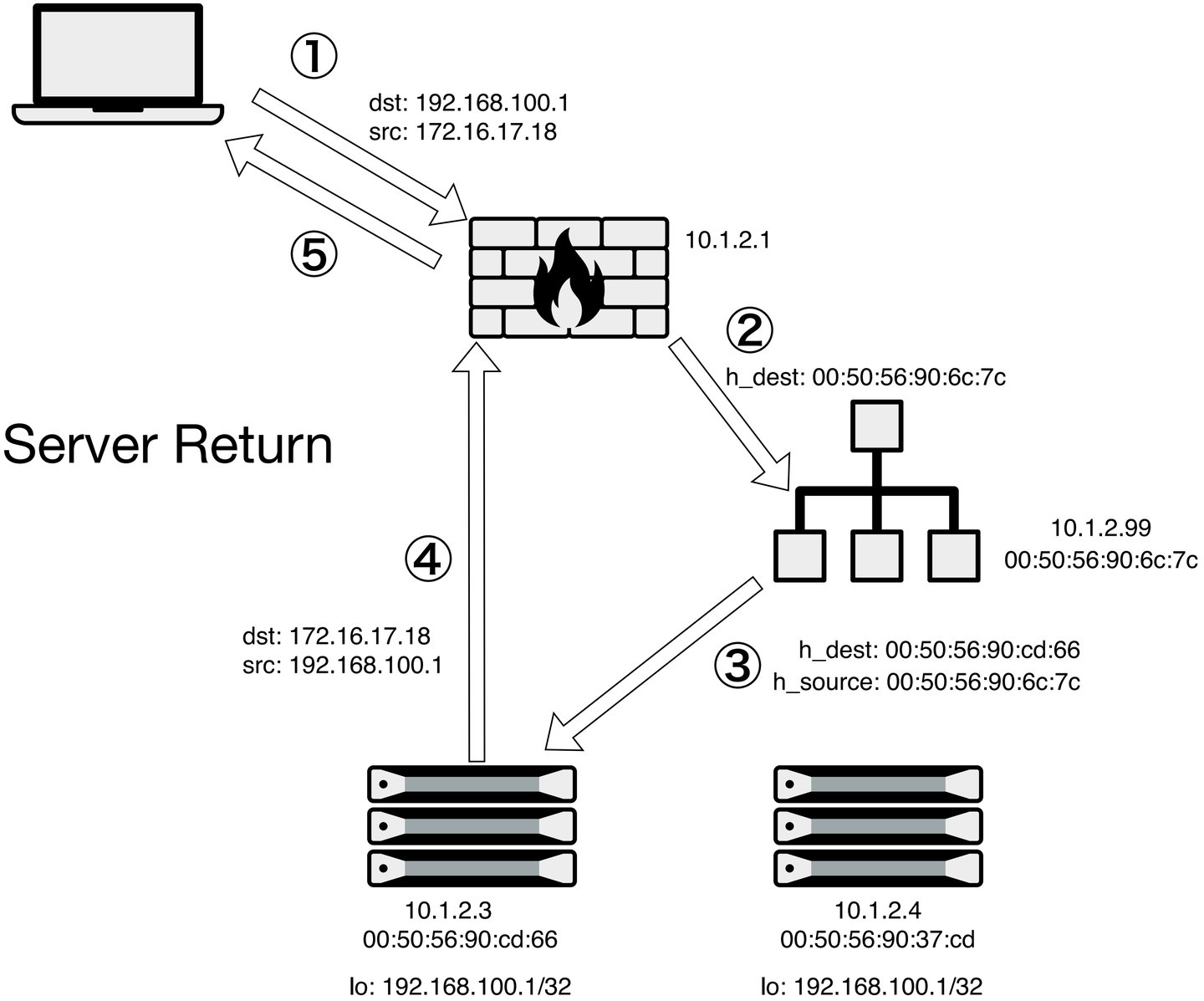
Background

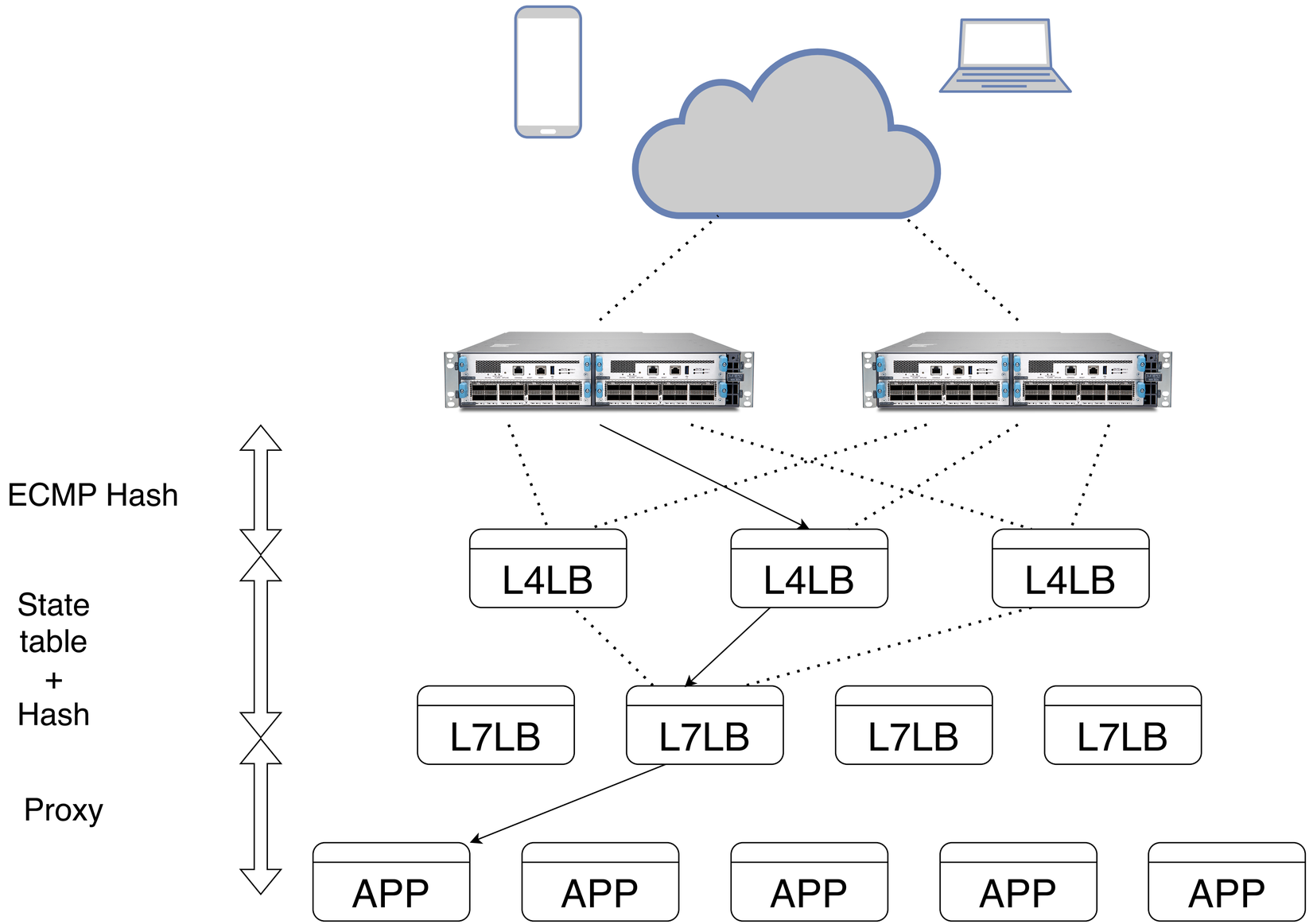
- UK’s largest commercial broadcast radio company
- Legacy hardware LB appliances handling all traffic
- Replace with one solution for high bandwidth applications
- Separate solution for low bandwidth applications
- 95% long lived TCP connections (Icecast)
- IPv4 only, layer 4 LB, Direct Server Return mode at layer 2

Layer 2 Direct Server Return

Direct Routing

nPath (f5)




ECMP Hash
State table + Hash
Proxy
L4LB
L4LB
L4LB
L7LB
L7LB
L7LB
L7LB
APP
APP
APP
APP
APP

Solutions considered

- Google Maglev
- Cloudflare Unimog
- Github GLB
- Facebook Katran
- IPVS

Try building something with XDP/eBPF?

- Direct Server Return relatively simple to implement
- Need the VIP on loopback interface on backend servers
- Update MAC addresses on incoming packets and TX
- How hard could it be?

```
SEC("xdp") int xdp_forward_func(struct xdp_md *ctx)
{
    const __u32 vip = 192<<24 | 168<<16 | 100<<8 | 1;
    const __u8 h_dest[][6] = {{ 0x00, 0x50, 0x56, 0x90, 0xcd, 0x66 },
                              { 0x00, 0x50, 0x56, 0x90, 0x37, 0xcd }};

    void *data_end = (void *)(long)ctx->data_end;
    struct ethhdr *eth = (void *)(long)ctx->data;
    struct iphdr *ip4 = (void *)(eth + 1);

    if ((eth + 1 > data_end) || (eth->h_proto != bpf_htons(ETH_P_IP)) ||
        (ip4 + 1 > data_end) || (ip4->daddr != bpf_htonl(vip)))
        return XDP_PASS;

    memcpy(eth->h_source, eth->h_dest, 6);
    memcpy(eth->h_dest, h_dest[bpf_ntohl(ip4->saddr) % (sizeof(h_dest)/6)], 6);

    return XDP_TX;
}
```

eBPF maps can make this dynamic

- Services and destinations
- VLANs and interfaces
- Backend selection and flow tracking
- Statistics
- Queues

Required library userland functionality

- Control plane functions to manage services
- Backend MAC address discovery via ICMP/ARP
- IP address/VLAN interface discovery
- Other NIC settings, eg. 802.3ad
- Ability to check backend server health *by VIP*

type Client

```
type Client interface {
    Info() (Info, error)

    Config() (Config, error)
    SetConfig(Config) error

    Services() ([]ServiceExtended, error)
    Service(Service) (ServiceExtended, error)
    CreateService(Service) error
    UpdateService(Service) error
    RemoveService(Service) error

    Destinations(Service) ([]DestinationExtended, error)
    CreateDestination(Service, Destination) error
    UpdateDestination(Service, Destination) error
    RemoveDestination(Service, Destination) error
}
```

```go
package main

import (
    "log"
    "net/netip"
    "os"

    "github.com/davidcoles/xvs"
)

// go run balancer.go eth0 10.1.2.0/24 192.168.100.1 10.1.2.3 10.1.2.4
func main() {
    eth := os.Args[1]
    pfx := os.Args[2]
    vip := os.Args[3]
    rip := os.Args[4:]

    options := xvs.Options{IPv4VLANs: map[uint16]netip.Prefix{1: netip.MustParsePrefix(pfx)}}
    client, err := xvs.NewWithOptions(options, eth)
    if err != nil {
        log.Fatal(err)
    }

    service := xvs.Service{Address: netip.MustParseAddr(vip), Port: 80, Protocol: xvs.TCP}
    if err := client.CreateService(service); err != nil {
        log.Fatal(service, err)
    }

    for _, r := range rip {
        destination := xvs.Destination{Address: netip.MustParseAddr(r)}
        if err := client.CreateDestination(service, destination); err != nil {
            log.Fatal(service, destination, err)
        }
    }
}
```

192.168.100.1	10.1.2.3	255.0.0.1
192.168.100.1	10.1.2.4	255.0.0.2
192.168.100.2	10.1.2.4	255.0.0.3
192.168.100.2	10.1.2.5	255.0.0.4
192.168.100.2	10.1.2.6	255.0.0.5

Each unique virtual and real server address pair is mapped to a NAT address

Health checks via NAT addresses

- `xvs` **interface is created with a route to NAT addresses**
- **To obtain NAT addr, use** `nat:=client.NAT(vip, rip)`
- **Use standard TCP/IP stack to build checks**
- `http.Get("http://" + nat.String())`
- **Build a custom** `http.Client` **which always uses NAT IP**

Minimum Viable Product load balancer:

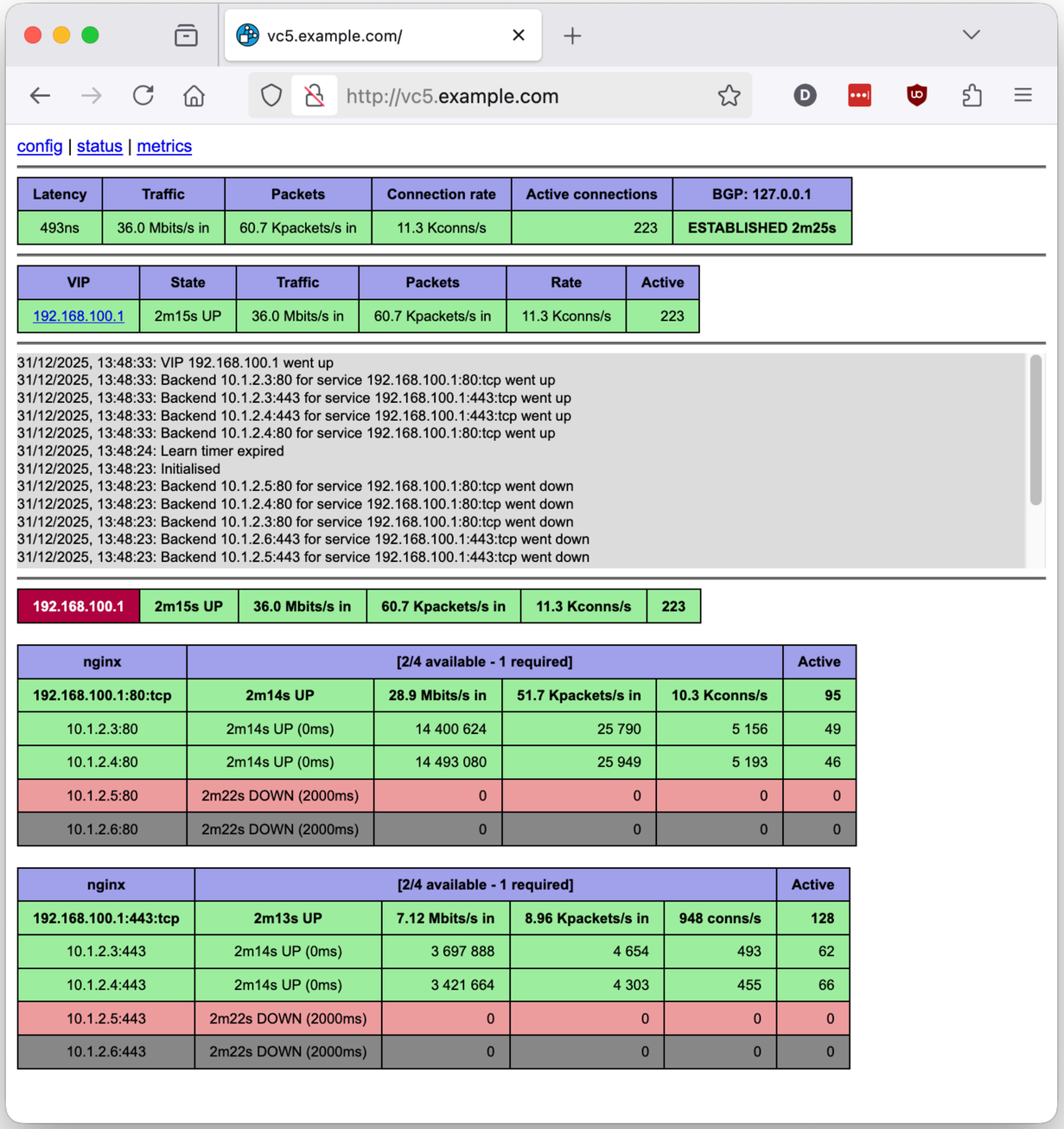
- Health checks
- Advertise VIPs via BGP - “Route Health Injection”
- Metrics
- Logging
- Flow state sharing
- Configuration, eg. YAML/JSON

Minimum Viable Product load balancer:

- Health checks
- Advertise VIPs via BGP - “Route Health Injection”
- Metrics
- Logging
- Flow state sharing
- Configuration, eg. YAML/JSON

There’s an app for that

vc5



config | status | metrics

Latency	Traffic	Packets	Connection rate	Active connections	BGP: 127.0.0.1
493ns	36.0 Mbits/s in	60.7 Kpackets/s in	11.3 Kconns/s	223	ESTABLISHED 2m25s

VIP	State	Traffic	Packets	Rate	Active
192.168.100.1	2m15s UP	36.0 Mbits/s in	60.7 Kpackets/s in	11.3 Kconns/s	223

```
31/12/2025, 13:48:33: VIP 192.168.100.1 went up
31/12/2025, 13:48:33: Backend 10.1.2.3:80 for service 192.168.100.1:80:tcp went up
31/12/2025, 13:48:33: Backend 10.1.2.3:443 for service 192.168.100.1:443:tcp went up
31/12/2025, 13:48:33: Backend 10.1.2.4:443 for service 192.168.100.1:443:tcp went up
31/12/2025, 13:48:33: Backend 10.1.2.4:80 for service 192.168.100.1:80:tcp went up
31/12/2025, 13:48:24: Learn timer expired
31/12/2025, 13:48:23: Initialised
31/12/2025, 13:48:23: Backend 10.1.2.5:80 for service 192.168.100.1:80:tcp went down
31/12/2025, 13:48:23: Backend 10.1.2.4:80 for service 192.168.100.1:80:tcp went down
31/12/2025, 13:48:23: Backend 10.1.2.3:80 for service 192.168.100.1:80:tcp went down
31/12/2025, 13:48:23: Backend 10.1.2.6:443 for service 192.168.100.1:443:tcp went down
31/12/2025, 13:48:23: Backend 10.1.2.5:443 for service 192.168.100.1:443:tcp went down
```

192.168.100.1	2m15s UP	36.0 Mbits/s in	60.7 Kpackets/s in	11.3 Kconns/s	223

nginx	[2/4 available - 1 required]				Active
192.168.100.1:80:tcp	2m14s UP	28.9 Mbits/s in	51.7 Kpackets/s in	10.3 Kconns/s	95
10.1.2.3:80	2m14s UP (0ms)	14 400 624	25 790	5 156	49
10.1.2.4:80	2m14s UP (0ms)	14 493 080	25 949	5 193	46
10.1.2.5:80	2m22s DOWN (2000ms)	0	0	0	0
10.1.2.6:80	2m22s DOWN (2000ms)	0	0	0	0

nginx	[2/4 available - 1 required]				Active
192.168.100.1:443:tcp	2m13s UP	7.12 Mbits/s in	8.96 Kpackets/s in	948 conns/s	128
10.1.2.3:443	2m14s UP (0ms)	3 697 888	4 654	493	62
10.1.2.4:443	2m14s UP (0ms)	3 421 664	4 303	455	66
10.1.2.5:443	2m22s DOWN (2000ms)	0	0	0	0
10.1.2.6:443	2m22s DOWN (2000ms)	0	0	0	0

```
---
bgp:
  as_number: 65001
  peers:
    - 10.1.2.254

services:
  - name: icecast
    virtual:
      - 192.168.100.1
    servers:
      - 10.1.2.3
      - 10.1.2.4
    host: foo.example.com
    path: /health
    policy:
      http:
      https:

vlans:
  1: 10.1.2.0/24
```

config.pl →

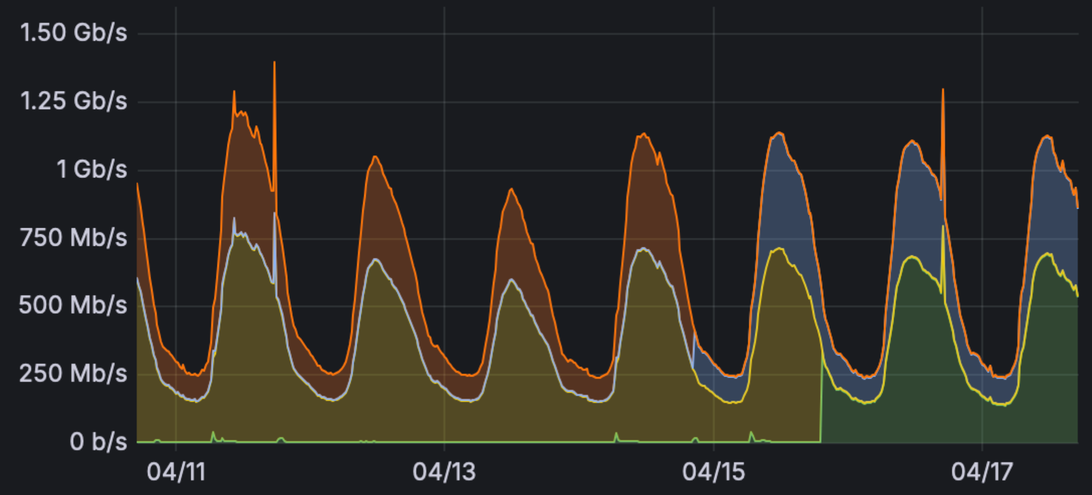
```
{
   "bgp" : {
      "10.1.2.254" : {
         "as_number" : 65001
      }
   },
   "services" : {
      "192.168.100.1:443:tcp" : {
         "name" : "icecast",
         "need" : 1,
         "reals" : {
            "10.1.2.3:443" : {
               "checks" : [
                  {
                     "host" : "foo.example.com",
                     "path" : "/health",
                     "type" : "https"
                  }
               ]
            },
            "10.1.2.4:443" : {
               "checks" : [
                  {
                     "host" : "foo.example.com",
                     "path" : "/health",
                     "type" : "https"
                  }
               ]
            }
         },
         "sticky" : false
      },
      "192.168.100.1:80:tcp" : {
          …
```

Deployment

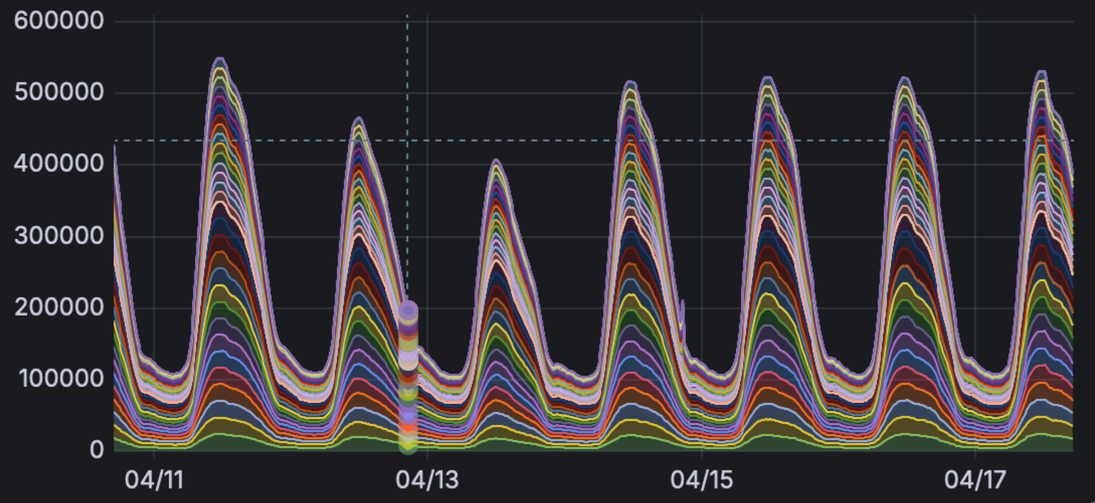
2024 -

- Ran for one year on VMs, small services
- Running our tier 1 services since the start of 2025
- Relatively modest servers - 2.30GHz Xeon, 32core, 32GB
- 4x10G Intel X710 NICs, running VLAN trunks with MLAG
- Migrated services via new VIPs and DNS

Traffic by load balancer instance



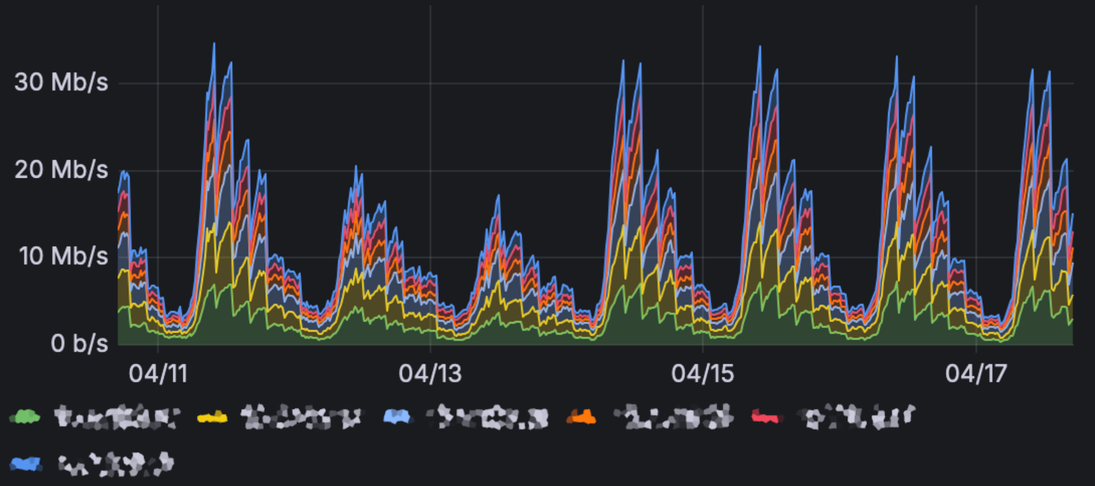
DAX streaming sessions by backend



DAX streaming backend sessions @

DAX streaming backend sessions @

DAX HLS ingress traffic by backend



MAP OF THE INTERNET

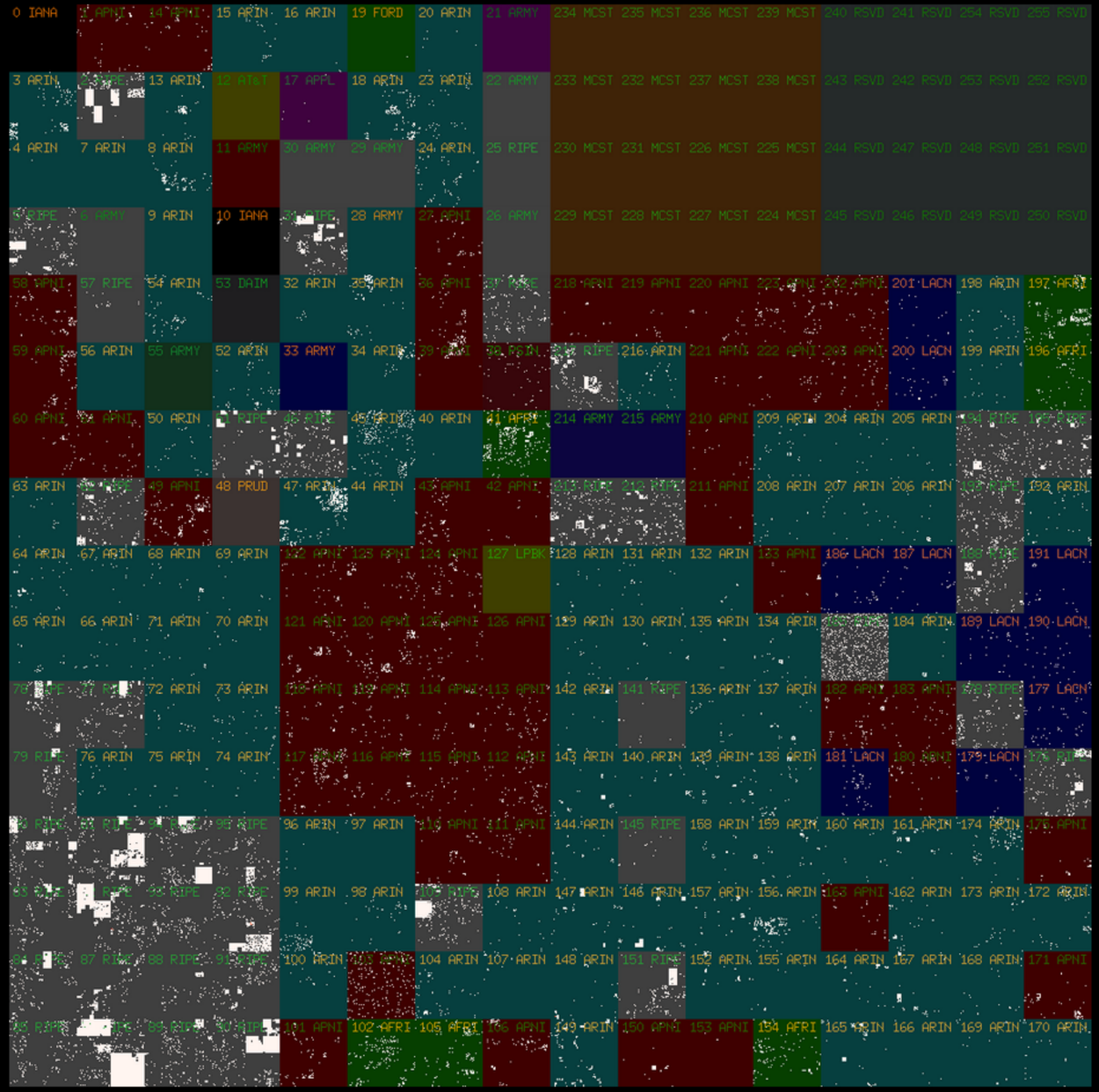
THE IPV4 SPACE, 2006

THIS CHART SHOWS THE IP ADDRESS SPACE ON A PLANE USING A FRACTAL MAPPING WHICH PRESERVES GROUPING -- ANY CONSECUTIVE STRING OF IPs WILL TRANSLATE TO A SINGLE COMPACT, CONTIGUOUS REGION ON THE MAP. EACH OF THE 256 NUMBERED BLOCKS REPRESENTS ONE /8 SUBNET (CONTAINING ALL IPs THAT START WITH THAT NUMBER). THE UPPER LEFT SECTION SHOWS THE BLOCKS SOLD DIRECTLY TO CORPORATIONS AND GOVERNMENTS IN THE 1990's BEFORE THE RIRs TOOK OVER ALLOCATION.

0 1 14 15 16 19 →
3 2 13 12 17 18
4 7 8 11
5 6 9 10

= UNALLOCATED BLOCK

https://xkcd.com/195/


0 IANA
1 APNI
14 APNI
15 ARIN
16 ARIN
19 FORD
20 ARIN
21 ARMY
234 MCST
235 MCST
236 MCST
239 MCST
240 RSVD
241 RSVD
254 RSVD
255 RSVD
3 ARIN
12 AT&T
17 APPL
18 ARIN
23 ARIN
22 ARMY
233 MCST
232 MCST
237 MCST
238 MCST
243 RSVD
242 RSVD
253 RSVD
252 RSVD
4 ARIN
7 ARIN
8 ARIN
11 ARMY
30 ARMY
29 ARMY
24 ARIN
25 RIPE
230 MCST
231 MCST
226 MCST
225 MCST
244 RSVD
247 RSVD
248 RSVD
251 RSVD
6 RIPE
6 ARMY
9 ARIN
10 IANA
31 RIPE
28 ARMY
27 APNI
26 ARMY
229 MCST
228 MCST
227 MCST
224 MCST
245 RSVD
246 RSVD
249 RSVD
250 RSVD
58 APNI
57 RIPE
54 ARIN
53 DAIM
32 ARIN
35 ARIN
36 APNI
37 RIPE
218 APNI
219 APNI
220 APNI
223 APNI
202 APNI
201 LACN
198 ARIN
197 APNI
59 APNI
56 ARIN
55 ARMY
52 ARIN
33 ARMY
34 ARIN
39 APNI
38 RIPE
216 ARIN
221 APNI
222 APNI
203 APNI
200 LACN
199 ARIN
196 AFRI
60 APNI
61 APNI
50 ARIN
51 RIPE
46 RIPE
45 ARIN
40 ARIN
41 APNI
214 ARMY
215 ARMY
210 APNI
209 ARIN
204 ARIN
205 ARIN
194 RIPE
195 RIPE
63 ARIN
62 RIPE
49 APNI
48 PRUD
47 ARIN
44 ARIN
43 APNI
42 APNI
213 RIPE
212 RIPE
211 APNI
208 ARIN
207 ARIN
206 ARIN
192 APNI
193 ARIN
64 ARIN
67 ARIN
68 ARIN
69 ARIN
122 APNI
123 APNI
124 APNI
127 LPBK
128 ARIN
131 ARIN
132 ARIN
133 APNI
186 LACN
187 LACN
188 RIPE
191 LACN
65 ARIN
66 ARIN
71 ARIN
70 ARIN
121 APNI
120 APNI
125 APNI
126 APNI
129 ARIN
130 ARIN
135 ARIN
134 ARIN
184 ARIN
189 LACN
190 LACN
78 RIPE
77 RIPE
72 ARIN
73 ARIN
115 APNI
119 APNI
114 APNI
113 APNI
142 ARIN
141 RIPE
136 ARIN
137 ARIN
182 APNI
183 APNI
178 RIPE
177 LACN
79 RIPE
76 ARIN
75 ARIN
74 ARIN
117 APNI
116 APNI
115 APNI
112 APNI
143 ARIN
140 ARIN
139 ARIN
138 ARIN
181 LACN
180 APNI
179 LACN
176 RIPE
93 RIPE
92 RIPE
91 RIPE
95 RIPE
96 ARIN
97 ARIN
110 APNI
111 APNI
144 ARIN
145 RIPE
158 ARIN
159 ARIN
160 ARIN
161 ARIN
174 ARIN
175 APNI
83 RIPE
82 RIPE
85 RIPE
84 RIPE
99 ARIN
98 ARIN
109 APNI
108 ARIN
147 ARIN
146 ARIN
157 ARIN
156 ARIN
163 APNI
162 ARIN
173 ARIN
172 ARIN
81 RIPE
87 RIPE
88 RIPE
91 RIPE
100 ARIN
103 APNI
104 ARIN
107 ARIN
148 ARIN
151 RIPE
152 ARIN
155 ARIN
164 ARIN
167 ARIN
168 ARIN
171 APNI
80 RIPE
86 RIPE
89 RIPE
90 RIPE
101 APNI
102 AFRI
105 AFRI
106 APNI
149 ARIN
150 APNI
153 APNI
154 AFRI
165 ARIN
166 ARIN
169 ARIN
170 ARIN

Added layer 3 and IPv6 support

- Tunnel traffic over IP: IPIP, GRE, FOU and GUE supported
- IPv4 or IPv6 may be used for VIP or RIP (any combination)
- Need to tell the library about which gateway to use
- Layer 3-only far simpler than layer 2 with multiple VLANs

Issues encountered during rollout

- Intel X710 media negotiation failures
- VMware vmxnet3 buffer sizing bug in driver mode
- Program size increased with layer 3 support - use tail calls
- FOU/GUE termination on IPv6 unsupported?

Future development

- Config could be centralised … or
- Self service - teams could run their own instances … or
- Service discovery - etcd/Consul/ZooKeeper
- Tunnel termination eBPF utility
- We have the flexibility to proceed at our own pace


ssh
root@load-balancer:~/demo#
root@client: ~ (ssh)
root@client:~#
Default (ssh)
root@backend2:~#

This slide intentionally left blank in case of video playback screwups

Enjoy FOSDEM!

github.com/davidcoles/xvs

github.com/davidcoles/vc5

github.com/davidcoles/fosdem2026