Networking

Steven Barth

The Good ol' Days...

Image: "legacy-caution" by Phil Benchoff; CC BY 2.0

Static Configuration

- Fixed private addresses
- DHCP to clients
- Leases & hostnames stored
- NAT hides dynamic changes

Bootstrapping

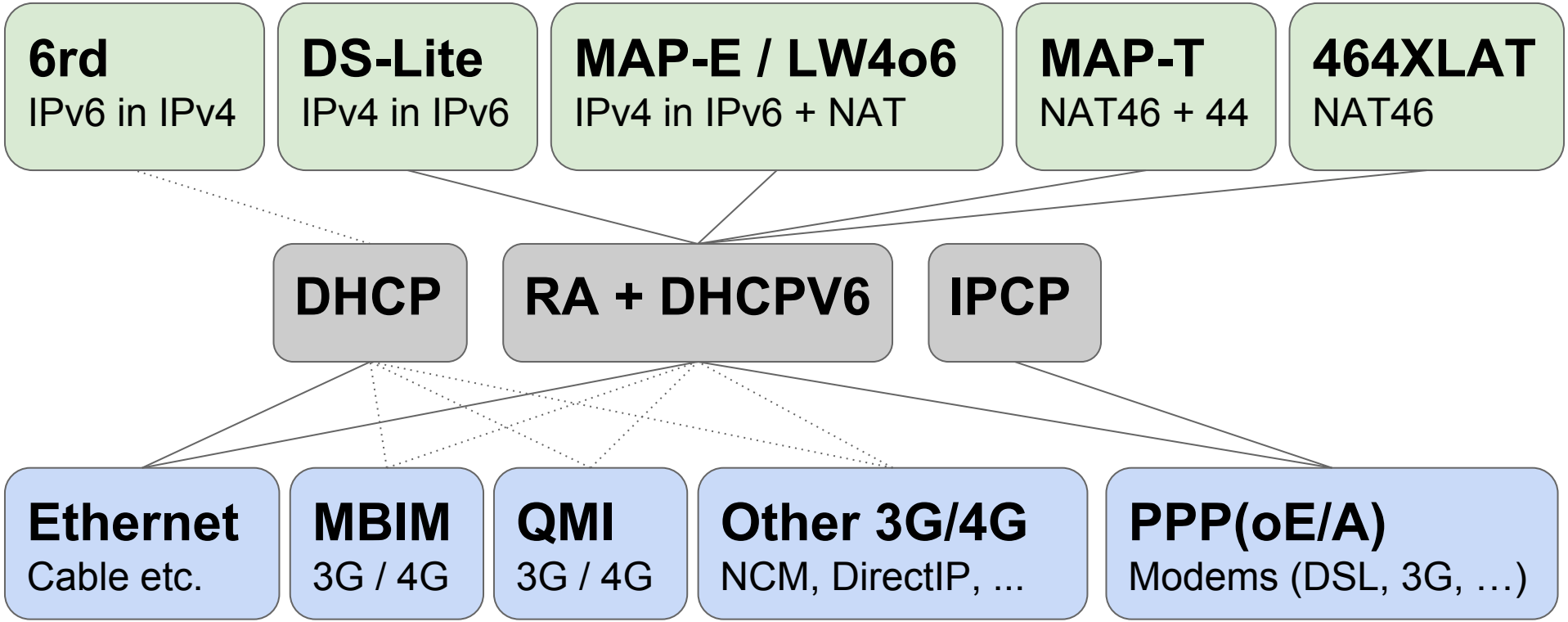
DHCP

IPCP

Ethernet
Cable etc.

PPP(oE/A)
Modems (DSL, 3G, …)

… and bootstrapping now



Expose it to the user?

Configure only what is necessary!

Ethernet / WiFi

```
config interface wan
        option ifname eth1
        option proto dhcp

config interface wan6
        option ifname eth1
        option proto dhcpv6
```

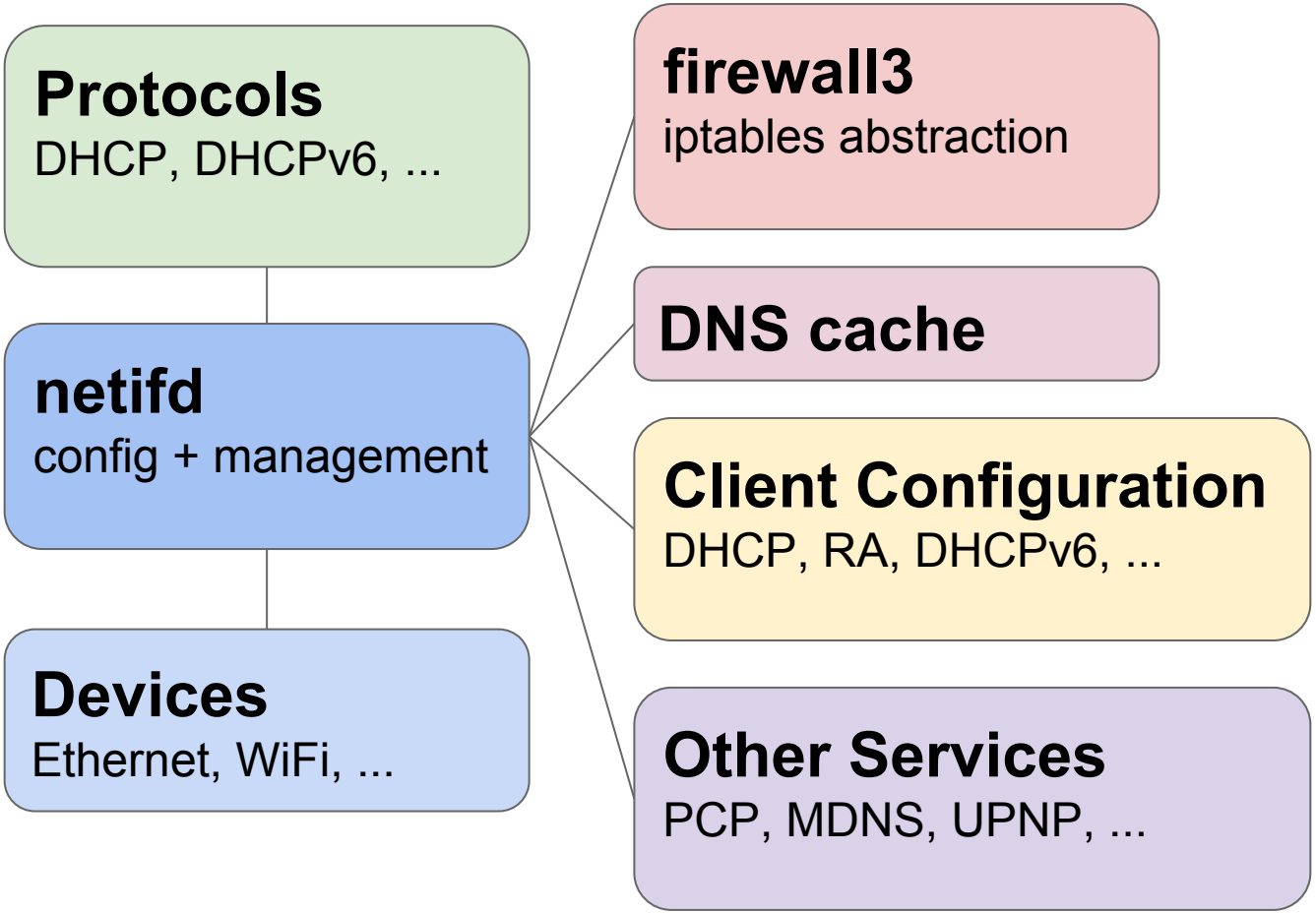
DSL (PPPoE/A)

```
config interface wan
        option ifname   eth1
        option proto    pppoe
        option username #user#
        option password #pass#
```

3G / 4G

```
config interface wan
        option proto    wwan
        option pincode #code#
        option apn #apn#
```

Network Subsystem Overview



netifd: heart of the network subsystem

Configuration & Events

store and manage configuration
calculate minimal changes
react to events: kernel, ubus

Addresses & Route

manage and distribute
send ubus + script events
handle auxiliary data (DNS, ...)

Device Setup

bring up / teardown
hotplug & carrier-detection
bridges, 802.1q/ad, macvlan
settings: L2-address, MTU, …

Layer 3 Protocols

ubus + shell based API
stackable protocols
dependency support

Example Flow

config interface wan
option ifname eth1
option proto dhcp
option hostname foobar

Configuration Event
new interface using device eth1
→ listen for eth1 device events

Kernel device event
eth1 available and cable plugged in

Run dhcp protocol handler
pass option hostname

Protocol Event
success: IP=1.2.3.4, GW=2.3.4.5
on wan (eth1)

Assign addresses and routes
Assign 1.2.3.4 to eth1
Setup default route to 2.3.4.5

Declaring protocols

```
#!/bin/sh

. /lib/functions.sh
. ../netifd-proto.sh
init_proto "$@"

proto_dhcpv6_init_config() {
        proto_config_add_string clientid
}

proto_dhcpv6_setup() {
        local config="$1"
        local iface="$2"

        local reqaddress clientid
        json_get_vars clientid
        [ -n "$clientid" ] && append opts "-c$clientid"

        proto_export "INTERFACE=$config"
        proto_run_command "$config" odhcp6c \
                -s /lib/netifd/dhcpv6.script \
                $opts $iface
}

proto_dhcpv6_teardown() {
        local interface="$1"
        proto_kill_command "$interface"
}

add_protocol dhcpv6
```

file: /lib/netifd/proto/dhcpv6.sh (simpl.)

1. Some Preamble
2. Declare Configuration
3. Protocol setup function
   a: retrieve configuration

   b: spawn protocol daemon
4. Declare teardown function
5. Register protocol

Handling protocol replies

```
#!/bin/sh
. /lib/functions.sh
. /lib/netifd/netifd-proto.sh

setup_interface () {
        proto_init_update "*" 1
        for entry in $ADDRESSES; do
                local addr="${entry%%/*}"
                entry="${entry#*/}"
                local mask="${entry%%,*}"

                proto_add_ipv6_address "$addr" "$mask"
        done
        proto_send_update "$INTERFACE"
}

teardown_interface() {
        proto_init_update "*" 0
        proto_send_update "$INTERFACE"
}

case "$2" in
        bound|informed|updated|rebound)
                setup_interface "$1"
        ;;
        started|stopped|unbound)
                teardown_interface "$1"
        ;;
esac
```

file: /lib/netifd/dhcpv6.script (simpl.)

1. Some Preamble

2. Declare a setup function
   Parse status data and turn it into netifd configuration information

3. Declare a teardown function

4. Depending on daemon state, run setup or teardown function

Stackable Protocols and Dependencies

```
config interface wan
        option proto    wwan
        option pincode #code#
        option apn #apn#
```

```
ubus call network add_dynamic "$(json_dump)"

json_init
json_add_string name "${interface}_6"
json_add_string ifname "@$interface"
json_add_string proto "dhcpv6"
json_add_string extendprefix 1
ubus call network add_dynamic "$(json_dump)"


mbim_setup() {
```

Run interface wan (wwan)
detect and bring up modem “link”
(IPv6 modem “link” was brought up)

Create interface wan_6 (dhcpv6)
configure IPv6 layer 3
detect if 464XLAT / DNS64 is used

Create interface wan_6_4 (464xlat)
configure virtual IPv4 layer 3

WiFi Features

netifd: configuration

change management
plugins to abstract drivers
wpad (hostapd) integration

multicast to unicast

generic conversion layer
workaround WiFi shortcomings

iwinfo: monitoring

status abstraction layer
query settings & capabilities
query associated stations
scan for nearby networks

client “pseudo-bridge”

in the absence of WDS
relayd: proxy ARP, DHCP
odhcpd: proxy NDP, RA, DHCPv6

Custom IPv6 Stack

Uplink Configuration

auto-detected bootstrap
many painful ISP work-arounds
prefix distribution support

Softwire support

lots of transitional technologies
encapsulation and natting
IPv4 address sharing support

Client Configuration

designed for compatibility
optimized power saving defaults
support for downstream routers

Firewall

Generic

event-triggered iptables rule generator
hooks and daemon integration (PCP, UPNP IGD, …)

Zones

aggregation of interfaces
source and destination of rules
assigned via config or protocol

Rules

filtering and NAT
static rules through UCI config
dynamic rules from protocols

Other OpenWrt network projects in core

odhcp6c	IPv6 Router Advertisement & DHCPv6+PD client
uhttpd2	http(s) daemon + json-rpc - ubus bridge
mdns	DNS-SD querier & announcer
map + 464xlat	MAP-E, MAP-T, LW4over6 & 464xlat implementation
umbim	MBIM 3g/4g modem client
uqmi	QMI 3g/4g modem client
omcproxy	IGMPv3 / MLDv2 multicast proxy

QoS, SQM and Bufferbloat.net

Bufferbloat?

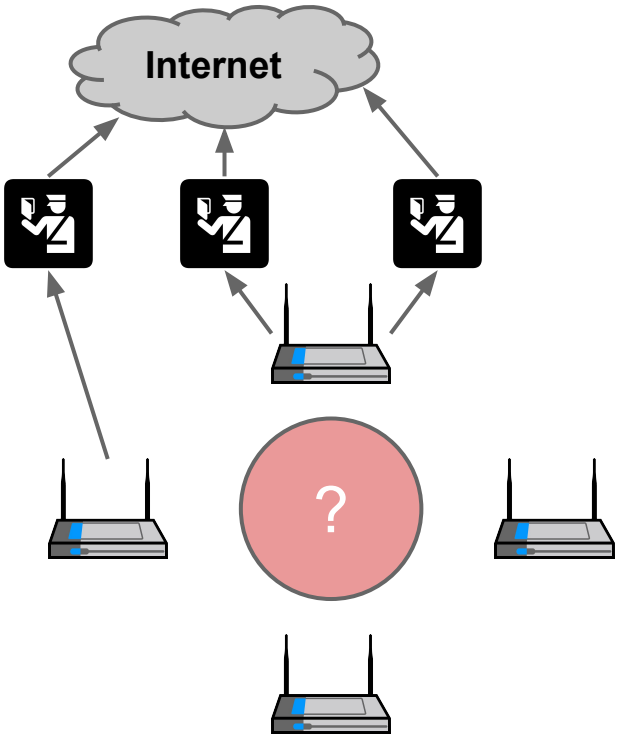
High latency & congestion through lots
of large dumb buffers all over the data paths.

(TCP) packets clog queues → latency rises!

Solution?

controlled delay (fq_codel) by default
optionally full smart queue management scripts
for full control over the bottleneck

Multi-router and multi-ISP networks



What if I want to utilize multiple ISPs at once (ala multipath TCP or Google's QUIC)?

→ IPv6 + source-address aware routing

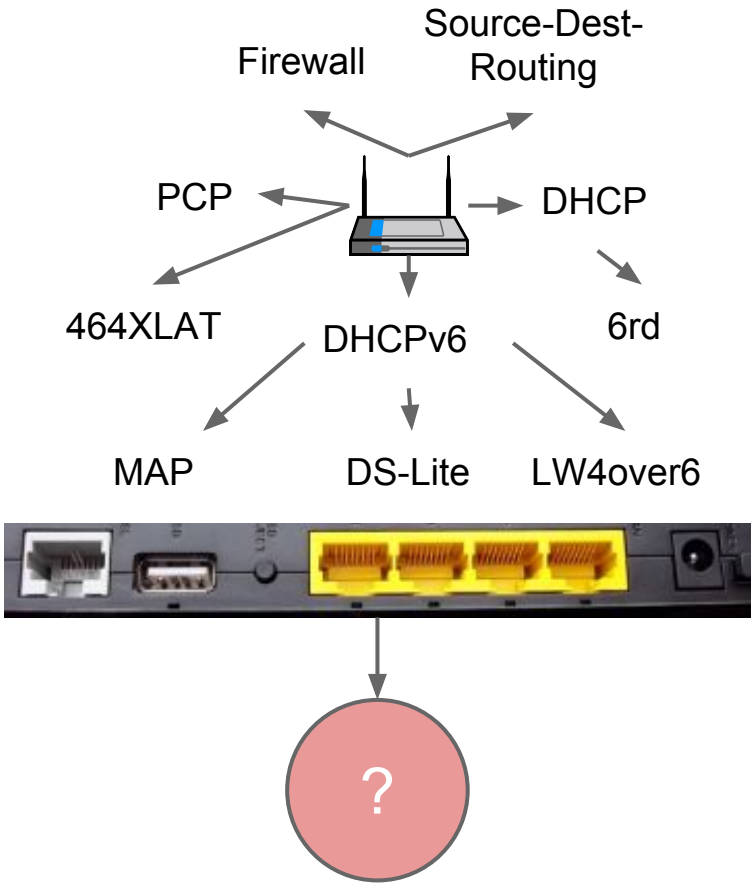
What if I want to directly access my IOT devices without going through the cloud?

→ Layer 2 bridging? Take it to Layer 3!

→ Multi Router SOHO networks

This is unmanageable… for a user!

Research in the IETF homenet WG...



We can build relatively universal more or less self-configuring IPv4 + IPv6 SOHO routers!

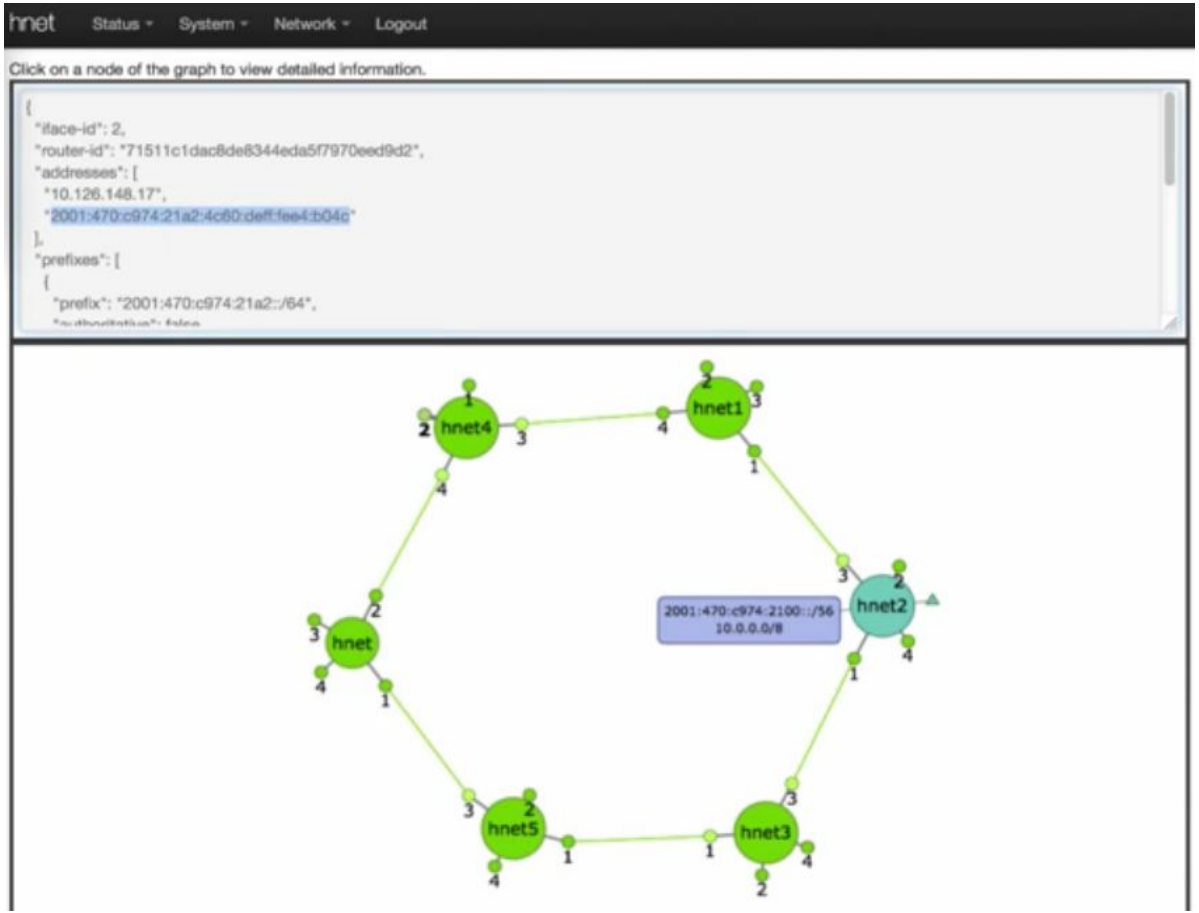
Can we take this one step further?
- Getting rid of WAN-port and LAN-bridge?

Can we scale this up to arbitrary networks?
- "Plug & Play" routers?
- But who "owns" the network(s)?

→ Find a consensus among equal routers
- → DNCP: a distributed consensus protocol

→ Specify requirements for interoperability
- → HNCP: autonomous networks using DNCP

… towards autonomous networks!



→ Topology Detection

→ Border Discovery & Setup

→ Routing Setup

→ Naming & Service Discovery

→ Status Distribution

→ Security Bootstrap

Read more
→ www.homewrt.org

Thank you for your attention! Questions?

Steven Barth
<cyrus@openwrt.org>