android

Networking In Your Pocket

How the Linux networking stack is made to work on Android devices

netdev1.1, Seville, 2016-02-10

{lorenzo,ek}@google.com

Agenda

Mobile device networking

Android routing architecture

Current state of kernel features

Looking forward

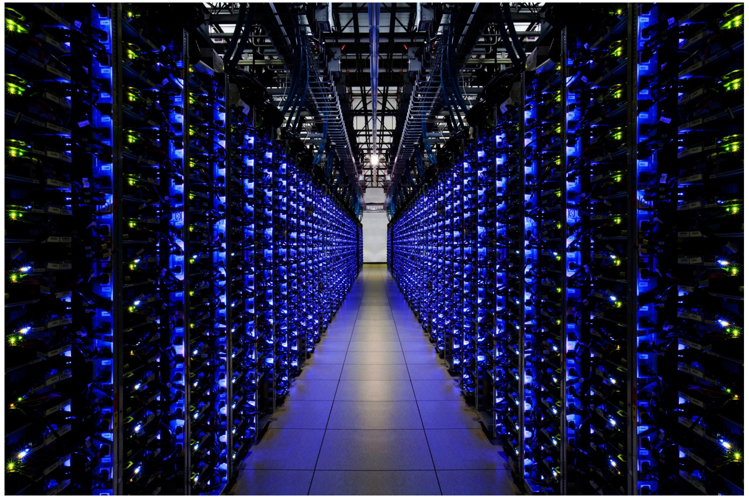
android

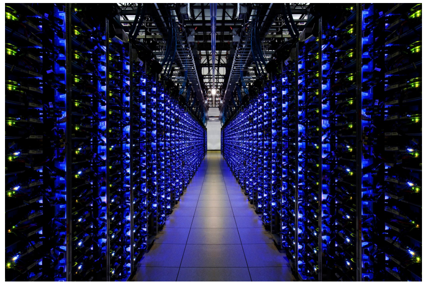
Mobile device networking

android

nexus
Google
nexus
HUAWEI
LG

! =





Moves
Multiple networks
Many link types
Captive portals, disconnected networks, …
Varying network quality
Limited power
Limited / metered bandwidth
Network bounds performance
Single-user
Per-app networking

Does not move
Fixed network attachment points
Mostly Ethernet
Managed connectivity
Stable, high-quality connectivity
Plenty of power
Plenty of bandwidth
High performance
High scalability
Virtualization

Network stack challenges

- Device is connected to multiple networks most of the time
  - VoLTE, Wi-Fi, mobile data/MMS...
- Need fast, seamless network handover
  - Close TCP connections when network disconnects or apps spin
  - Sockets bound on pre-switch network must stay on it until torn down
- IPv6 as a first class citizen
  - >60% of Android devices on US cell networks have IPv6
- Low power usage
- Android 6.0 still supports devices running 3.4 (!) kernels :-(

android

Features users expect

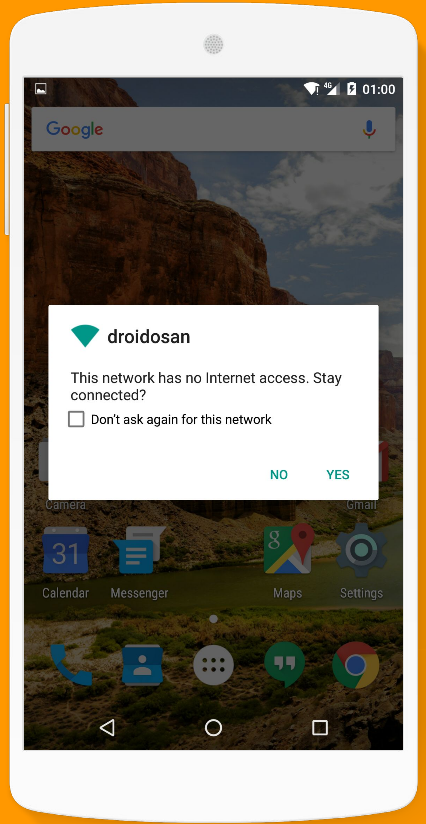
- Captive portal detection/login in background
  - Receive chat messages while you log in to free wifi
- Avoid disconnected networks
  - Use a wifi camera and access the Internet at the same time
  - Don’t use a wifi AP that has no backhaul or has an Internet outage
- Provide network selection APIs to apps
  - MMS app needs to bring up MMS when on wifi
  - Wireless printer /camera app needs to prefer wifi over mobile data
- Data usage tracking
- Per-application, per-user VPN
- Per-network permissions

Simple example

When a user manually selects a network that has no Internet access (e.g., wireless printer), the user is asked whether they want to stay connected to that network.

Internet access continues on mobile data until user decides to connect, at which point default device connectivity switches to wifi.

Requires the ability to send traffic on wifi while the rest of the system is using mobile data.


droidosan

This network has no Internet access. Stay connected?

☐ Don't ask again for this network

NO YES

Android routing architecture

android

Implementing the strong host model

This is a hard requirement: networks don't route each other's traffic. But:

- Can't `bind()` to IP address and use source routing
  - On a dual-stack network, at least 3-4 addresses on each interface
  - With IPv6 autoconf, addresses can appear at any time
  - Appropriate source address depends on DNS lookup and RFC6724
- Can't `SO_BINDTODEVICE` to interface
  - Network can have more than one interface (e.g., 464xlat)
- Can't use iptables to change routing after the fact
  - Local NAT breaks `getsockname()`, MTU/MSS selection, IPv6…
- Can't use namespaces
  - Would need to copy IP addresses between namespaces
  - At the time, interfaces could only be in one namespace

Android connectivity architecture

- Based on available Linux primitives and weak host model
- Multiple routing tables using routing rules
- `Network` API concept loosely equivalent to IETF “Provisioning domain”
- Connections bound to networks via socket marks
  - Implicit on connect() or incoming SYN packet
  - Explicit due to application API usage
- Per-application routing (e.g., VPN) using in-kernel per-UID routing
  - On Android, each app is its own UID
  - Like `xt_owner`, only works until `sock_orphan` is called

Routing tables

- One routing table per interface
  - Routes from different interfaces don't stomp on each other
  - A "network" has 1 or more Interfaces
  - `main` routing table only used in special cases (**never** by apps)
- ip rules select networks
  - Switch between networks = change lowest-priority ip rule
  - Select network = match rule
  - Criteria:
    - Socket mark
    - Bound / specified interface (for `SO_BINDTODEVICE` / `in6_pktinfo`)
    - Incoming interface for tethering
- Completely independent from IP addresses, which can change all the time

Socket marks

- Used to bind sockets to networks
  - Mark matches rule which selects routing table
- Outgoing traffic: `SO_MARK` requires `CAP_NET_ADMIN`
  - Network selection APIs pass socket fd to `netd` process for marking
  - Shim libc so that `connect()`, etc. pass socket to `netd` as well
    - Per-process default socket mark might help here
- Incoming connections:
  - fwmark applied to incoming SYN packet via iptables rules
  - Mark written back into socket mark by kernel on `accept()`
- Currently contain:
  - Network ID
  - Permissions (SYSTEM / CHANGE_NETWORK_STATE / NONE)
  - VPN protect bit

No IP addresses...

```
0:   from all lookup local
10000:       from all fwmark 0xc0000/0xd0000 lookup legacy_system
11000:       from all iif tun0 lookup local_network
12000:       from all fwmark 0xc00d7/0xcffff lookup tun0
12000:       from all fwmark 0x0/0x20000 uidrange 0-99999 lookup tun0
13000:       from all fwmark 0x10063/0x1ffff lookup local_network
13000:       from all fwmark 0x100d6/0x1ffff lookup rmnet0
13000:       from all fwmark 0x100d5/0x1ffff lookup wlan0
13000:       from all fwmark 0x100d7/0x1ffff uidrange 0-0 lookup tun0
13000:       from all fwmark 0x100d7/0x1ffff uidrange 0-99999 lookup tun0
14000:       from all oif rmnet0 lookup rmnet0
14000:       from all oif wlan0 lookup wlan0
14000:       from all oif tun0 uidrange 0-99999 lookup tun0
15000:       from all fwmark 0x0/0x10000 lookup legacy_system
16000:       from all fwmark 0x0/0x10000 lookup legacy_network
17000:       from all fwmark 0x0/0x10000 lookup local_network
19000:       from all fwmark 0xd6/0x1ffff lookup rmnet0
19000:       from all fwmark 0xd5/0x1ffff lookup wlan0
21000:       from all fwmark 0xd7/0x1ffff lookup wlan0
22000:       from all fwmark 0x0/0xffff lookup wlan0
23000:       from all fwmark 0x0/0xffff uidrange 0-0 lookup main
32000:       from all unreachable
```

Meaning of bits

`0x0000ffff`	- Network ID
`0x00010000`	- Explicit mark bit
`0x00020000`	- VPN protect bit
`0x000c0000`	- Permission bits

... and no `main` for non-root

Limitations

- Substantial complexity required to emulate strong host model
  - Routing rules, socket calls shimmed through `netd`, etc.
  - Marks are decided before routing lookup, so not always correct
    - Can’t look at mark and know what network a socket is on
    - Can’t provide seamless handover on bypassable VPNs

- Cannot reasonably support multiple networks on same interface
  - Would need to tag IP addresses, routes etc. with network ID
  - RFC7556 defines framework provisioning domain architecture
    - Working with IETF MIF working group to define API design
    - Once that’s done, implement / upstream?

Current state of kernel features

What’s upstream

What’s not upstream, and can we upstream it?

Socket marks

- Kernel-originated packets have zero mark
    - Replies (TCP RST, ICMP, ...) reflect socket mark of original packet
        - `fwmark_reflect` sysctl

- Incoming connection bound to network based on interface that got SYN
    - iptables INPUT rule marks incoming SYN
    - `accept()` writes `skb->mark` into `sk->sk_mark`
        - `fwmark_reflect` sysctl

- Upstream since 3.15

IPv6

- Uses kernel autoconf
    - Uses `accept_ra_rt_table` sysctl to put autoconf routes in right table
        - Not yet sent upstream

- Weak host model will happily use cell IPv6 address with wifi default route
    - During DAD
        - Upstreamed `use_optimistic` sysctl
    - If wifi provides default route and no address
        - Upstreamed `use_oif_addrs_only` sysctl

VPN

- Two modes
  - Secure: unprivileged traffic forced onto VPN
  - Bypassable: traffic on VPN by default, apps can choose otherwise
- VPN apps can provide / force VPN service to specific apps only
  - Each app is its own UID
- No local NAT, which enables IPv6 support, correct MSS / `getsockname`, ...
  - Means that VPN must be evaluated before source address is chosen
- Relies on per-UID routing rules - `FRA_UID_{START,END}`
  - Sent for review in 2012, not accepted
    - Attempt again?
    - Possible alternative: 64-bit socket mark?

Forced socket closes

- Currently use `SIOCKILLADDR` to close TCP connections on network switch
  - Has caused merge pain and kernel crashes when TCP code changes
  - Not flexible enough (VPN, mobile data always on…)

- `SOCK_DESTROY` recently upstreamed to replace it
  - Allows userspace to close individual sockets via `NETLINK_SOCK_DIAG`

- Future uses:
  - Better connection closing when a VPN comes up
  - Mobile data always on?

Data usage accounting / limiting

- Accounting
  - Tracks data usage for each app (UID) on each network interface
  - Uses out-of-tree `xt_qtaguid` module to collect stats for all combinations and publish result in `/proc`
    - `iptables` might be used to do this, but would require one rule per (UID, interface) pair
  - Upstreaming discussion a few years ago did not reach a conclusion

- Limiting
  - Relies on out-of-tree `xt_quota2` module to drop packets after limit
  - Still uses deprecated/removed netfilter nflog socket for notifications

Looking forward

Ongoing challenges

- Lots of backporting
  - Kernel versions mostly determined by SoC vendors
  - Substantial time window between upstream and kernel versions used by currently-supported devices
    - Nexus 5X/6P use 3.10
- iptables rules
  - iptables libraries are GPL-licensed, Android is Apache-licensed
  - fork / exec iptables takes >30ms… once for v4 and once for v6
  - Similar issue with any feature that has no stable/usable kernel API and GPL client library

android

Hardware offload

- Rely on lower-power chips to do networking instead of main CPU

- Today, IPsec keepalives
- Tomorrow:
  - TCP keepalives
  - Hardware packet filtering via BPF-like interpreter on wifi chipset
    - Useful to avoid repeated packets waking up the CPU
      - Some IPv6 networks send RAs every 3 seconds
  - …

- Unfortunately, not feasible to use emerging kernel APIs for this yet because device kernels very far from upstream

Questions?

android

Native Routing API

- Provide C/C++ level access to some multi-network functionality
- Still requires querying the framework to create/manage networks and learn associated network identifiers ("netids")
- E.g.

```
int android_setprocnetid(net_handle_t netid);
int android_setsocknetid(int fd, net_handle_t netid);
int android_getaddrinfofornetwork(net_handle_t netid,
    const char *name, ..., struct addrinfo **results);
```

Kernel unit testing

- Lots of network functionality since 5.0 relies on kernel changes

- Features not available in kernel releases used by SoC vendors
  - Some features are upstream in “recent” kernels like 3.15
  - Some features are not upstream at all

- When kernel features are missing, breakage is not always obvious
  - Might fail CTS (Compatibility Test Suite) test
  - Might behave incorrectly in some network environments
    - e.g., switching from Verizon Wireless to Comcast

net_test

- Runs Android kernel as a process on a Linux host
- Uses tap interfaces to test multi-network behaviour, examine packets
- Can be connected to Internet if desired
- Fast cycle time, easy to use, easy to develop
  - Unit tests are just Python using scapy
- https://source.android.com/devices/tech/config/kernel_network_tests.html

464xlat

- Provides IPv4 connectivity on IPv6-only networks
  - Full support for all apps on IPv6-only networks
  - Translates IPv6 <-> IPv4 in userspace
- Used by a number of mobile carriers
  - T-Mobile, SK Telecom, Orange Poland, …
- Works on wifi as well
  - Uses `IPV6_JOIN_ANYCAST` addresses for IPv6 neighbour discovery
- Packets sent through userspace via `PACKET_RX_RING` and tun interface
  - Performance could be better: ~200 Mbps on wifi
  - In-kernel implementation would be better
    - Cross-family translation not easy with current mangle/NAT