libuinet

The FreeBSD TCP/IP Stack as a Userland Library, Plus Extras

Patrick Kelsey <kelsey@ieee.org>

/whoami

The developer of libuinet

More at https://www.bsdcan.org/2014/schedule/speakers/236.en.html

Even more at www.linkedin.com/in/patrickjkelsey/

/agenda

Background

Goals and structure of the port

Promiscuous sockets

Passive receive

Performance

Future work

Q & A

/what is this

libuinet is a userland port of the FreeBSD TCP/IP stack, with interesting extensions

The project is young (but useful!)
- One developer, part time, since about a year ago
- No official release yet, not even a preliminary version number

This is not a research project (nothing wrong with research projects!)
- It was created to serve a specific need, although the results are more generally useful
- Currently more than one commercial product is being built with it

The aims of this talk are
- To explain what is going on
- To raise the profile of the project
- To attract commentary, users, further development

/origin story

Late in 2012, I was asked by Juli Mallett <jmallet@freebsd.org> if I'd be interested in adding a userland TCP/IP stack to WANProxy (http://wanproxy.org), and I insisted I was too busy

- Early in 2013, I was asked by Juli Mallett if I'd be interested in adding a userland TCP/IP stack to WANProxy, but again, I waffled
    - About a month later, I admitted I was reading code and having opinions, plus nobody else bit on freebsd-jobs@, plus there was funding. I was out of options. Work began.

The goal was to be able to build transparent TCP proxies with WANProxy that could support lots of connections (for values of lots > 64k) and lots of VLANs (including nested)

- And for the result to be portable (FreeBSD, Linux, OS X, …)

The decision was made to begin by porting the FreeBSD stack to userland to provide a platform to build upon

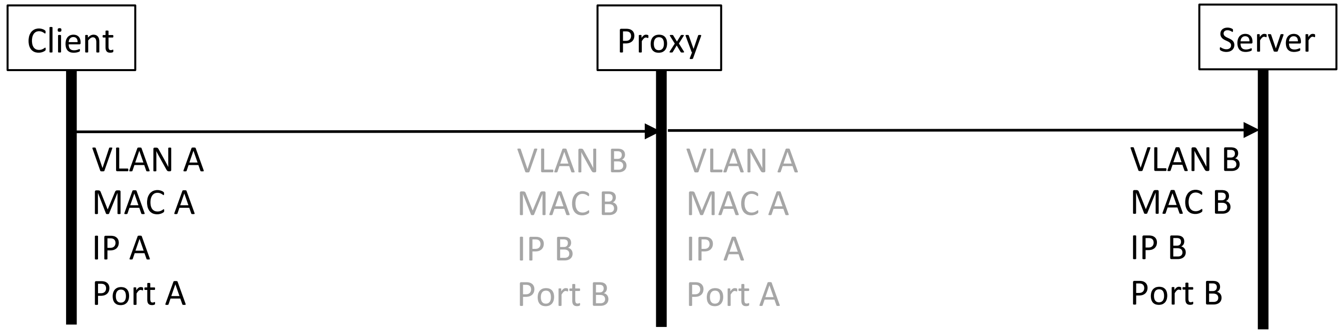
- Because it's stable, widely-used, being actively improved, and has a good license

/transparent TCP proxy

A transparent TCP proxy is one that can

- establish connections with local clients using VLAN tags, MAC addr, IP addr, and port number identical to that of a remote server, and can
- initiate connections to that remote server using VLAN tags, MAC addr, IP addr, and port number identical to that used by the local client

A scalable, transparent TCP proxy is one that can do this for a large number of arbitrarily addressed connections.



/port/goals

Scalability
- Non-blocking, event-based API
- Multi-thread and multi-instance

Tight coupling to the application
- Linked in as a library, everything in-process
- Callback-based API, other opportunities for enhanced functionality or performance

Portability
- POSIX environments, initially

Maintainability
- Track evolution of FreeBSD stack in a straightforward way, avoid becoming hopelessly stale

Non-goals
- Providing an unimposing, drop-in-replacement sockets library (either compile-time or run-time)
- Providing an access-controlled, multi-process facility

/port/alternatives

Name a ‘lightweight’ and/or ‘independent’ TCP/IP implementation

- They are all lightweight (feature-poor)
- And/or independent (small user base/not mature)

libplebnet

- Userland port of an 8-series FreeBSD TCP/IP stack, with different goals
- Apparently unmaintained/abandoned, parts used to seed the libuinet port

Rump kernel

- Framework for running NetBSD kernel components in userland, including the TCP/IP stack
- At the outset, it appeared an overly heavy framework for the intent here, with different API goals
- It was going to need non-trivial adaptation, opted to apply adaptation to the FreeBSD stack

/port/structure

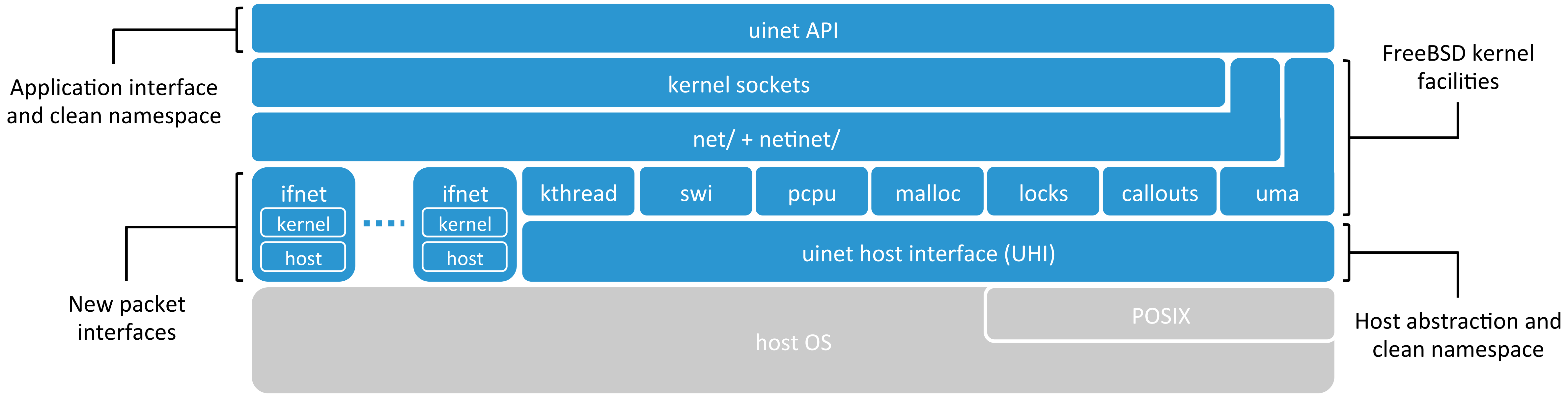
Approach

- Reimplement kernel facilities required by network stack, outside of kernel source tree
- Leave network code untouched, except where necessary for new features
- New features can completely disappear with #ifdef
- Try to implement new features so they could be used inside the kernel, not just in libuinet
- Make it possible to include libuinet in FreeBSD base, so that it is possible

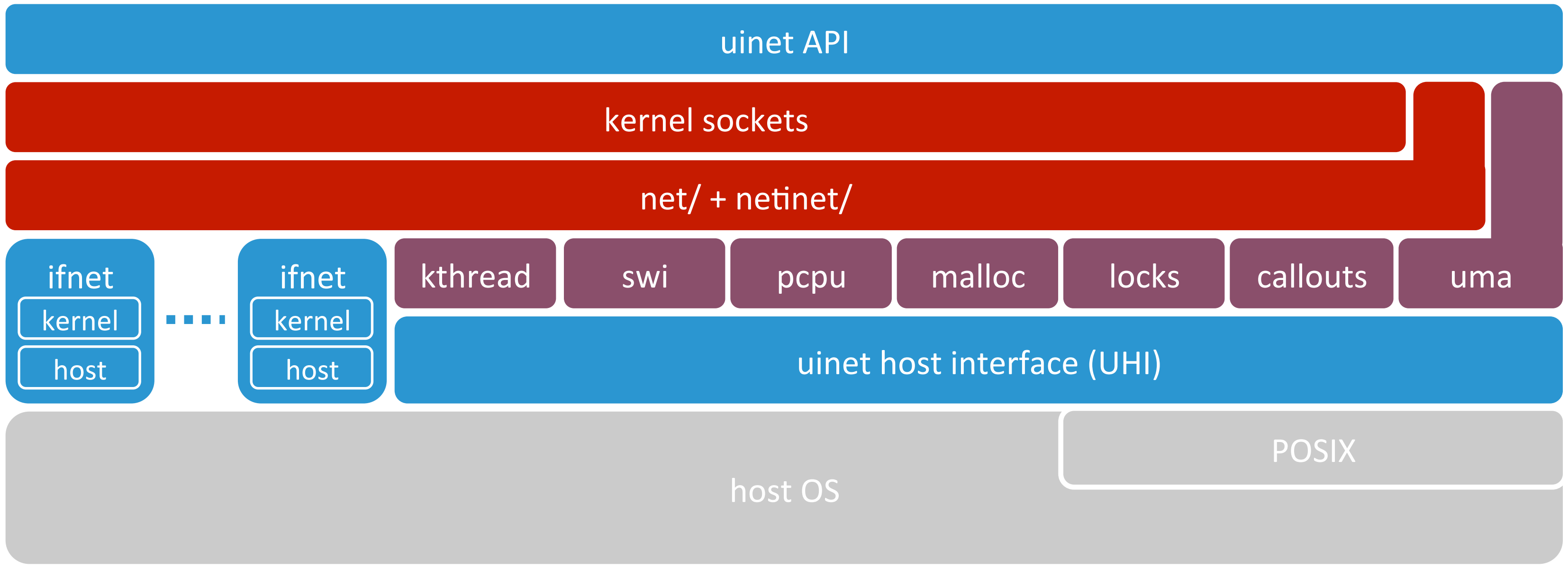
Source structure

- sys/ - kernel source subtrees needed by libuinet. Not all files are needed, whole subtrees are used to ease merging later releases
- lib/libuinet - userland reimplementations of needed kernel facilities and uinet API
- lib/libev - fork of libev with uinet watcher type
- lib/[other] - support libraries for sample programs
- bin/* - sample programs

/port/structure/layers

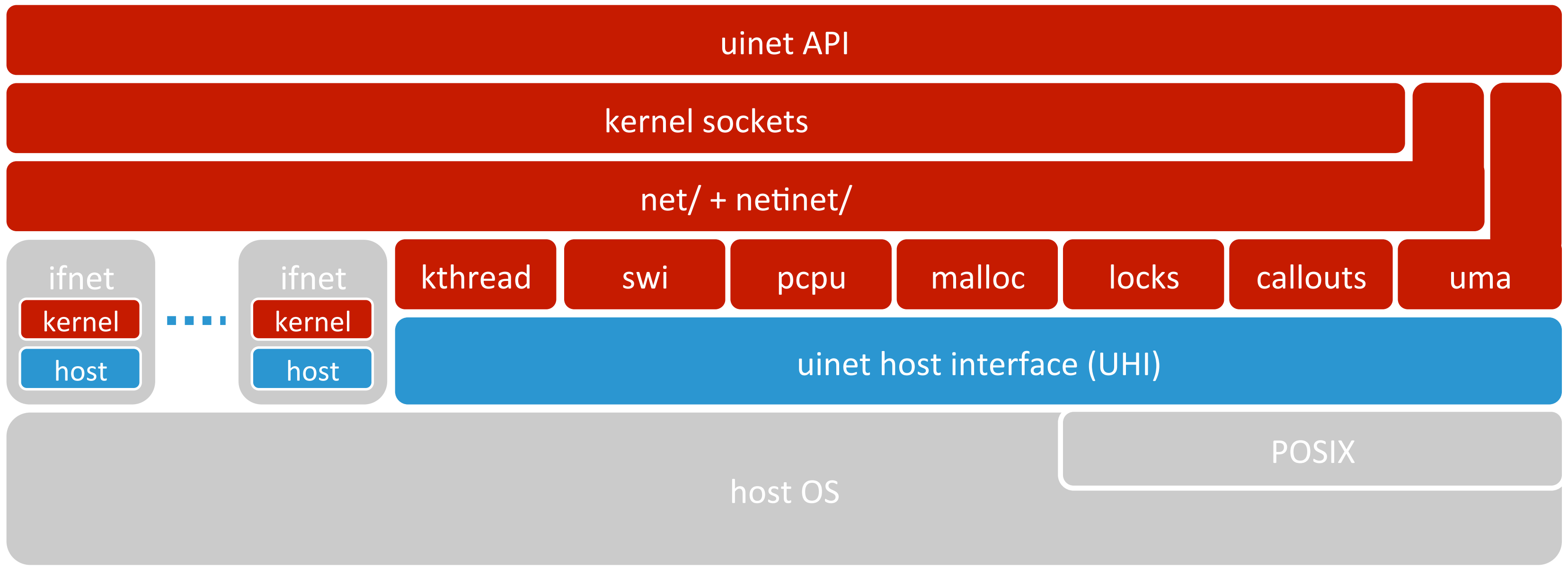


/port/structure/component sources



🟥 FreeBSD 🟦 libuinet 🟪 Partially reimplemented FreeBSD facilities

/port/structure/namespaces



🟥 Implementation can use FreeBSD kernel and UHI namespaces (kernel preferred)

🟦 Implementation can use host OS and UHI namespaces

/port/structure/uinet API

The uinet API is intended for non-blocking, event driven use

- (Actually, blocking operation is mostly supported, but currently no way to wait on groups of sockets)

This is supported natively by the FreeBSD kernel sockets API (nonblocking + upcalls)

The uinet API supports integration with existing event systems, because most people would be happier knowing/learning that event system than grokking bare upcalls* (lock ordering…)

First event system integration was WANProxy

- WANProxy facilities can now use host OS socket and/or libuinet sockets

Second event system integration was libev

- libev-based programs can now handle libuinet sockets in their event loops, along with host OS sockets, timers, signals, etc

*I made this up, but it feels defensible

/port/structure/packet interfaces

The packet interfaces are ifnet implementations, same as kernel interface drivers

Currently there are two included: netmap and pcap

netmap interface

- Does zero-copy on receive up to some fraction of ring buffers outstanding to the stack
- Builds with current netmap, but is not yet taking advantage of expanded rx buffers and tx zero-copy features

pcap interface

- Mainly used for testing using pcap files
- Also useful for portability work, as it is widely supported

Could be anything

- DPDK, unix domain sockets, clay tablets…

/port/open issues

Locking

- Currently, all kernel lock types are mapped to pthread_mutex
- Should at least arrange for a real rwlock (pthread_rwlock does not have matching recursion behavior, something must be built)

pcpu

- Can't fully/directly reimplement pcpu – there is no userland preemption disable
- Want to have pcpu optimizations, certainly don't want to turn them into pessimizations
- Rely on thread pinning?
  - Should deliver cache and inter-cpu lock contention avoidance benefits
  - Only pinned threads will benefit – all unpinned threads will thrash in a common context across cpus
- Make per-thread instead of per-cpu?
  - Think jemalloc arenas
  - Should deliver some lock contention avoidance benefit
  - Dynamic thread set an issue
- …

/promiscuous sockets

'Promiscuous sockets' is the term I'm using for

- Listen sockets that can control which VLAN tag stack, IP, and port they capture connections for, including wildcarding any of them (and in the case of VLANs, insisting there are no tags)
- Active sockets that can control which VLAN tag stack, src and dst MAC addrs, src IP and src port they use
- Supporting infrastructure for the above

Supporting infrastructure includes

- Bypassing routing via 'connection domains' (cdoms)
- New interface mode that provides additional handling of L2 info (VLANs and MAC addrs)
- SYN filters

/promiscuous sockets/cdoms

All interfaces belong to a connection domain

All established connection contexts (inpcbs) belong to a connection domain

All received packets are assigned to a connection domain based on arrival interface

A given received packet can only match a connection context in its connection domain

When sending, the outbound interface is chosen based on the connection context's connection domain

/promiscuous sockets/cdoms

cdom 0 is the default, and is special

- Outbound packets in cdom 0 are routed
- Multiple interfaces can be assigned to cdom 0
- Special among cdoms, actually normal in that this is how the stack usually works

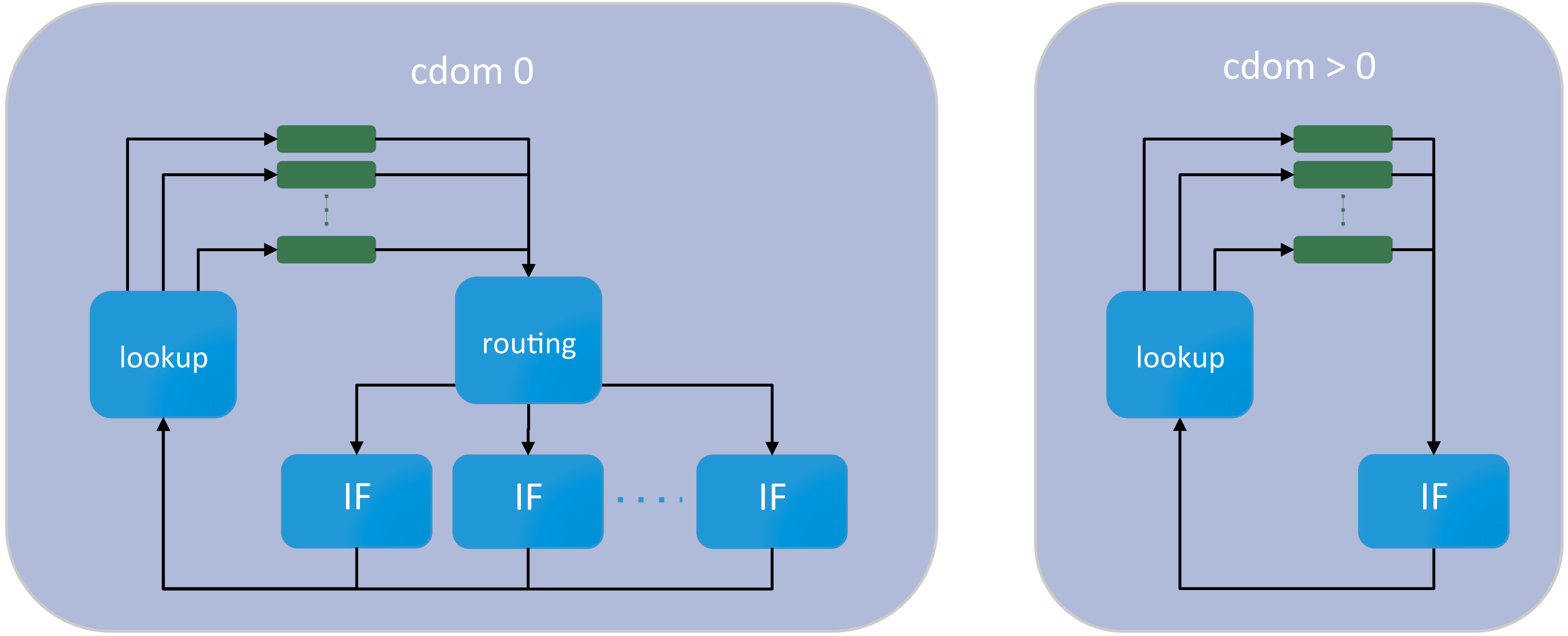
cdom > 0 used for promsicuous sockets

- Outbound packets are **not** routed
- Only one interface can be assigned

In the implementation, the existing FIB plumbing is used to implement the cdom concept, as

- Interfaces have a FIB property
- Sockets and connection contexts (inpcbs) have a FIB property
- mbufs have a FIB property

/promiscuous sockets/cdoms



🟩 Connection

/promiscuous sockets/interface mode

Promiscuous sockets are supported by a new interface mode, 'promiscuous inet' mode

When IFF_PROMISCINET is set on an interface, for each inbound packet

- VLAN tag stacks are removed
- The VLAN tag stack (if any) and MAC addrs are attached to the packet using an mbuf tag

When IFF_PROMISCINET is set on an interface, for each outbound packet

- VLAN tag stack and MAC addrs are sourced from attached mbuf tag

/promiscuous sockets/SYN filter

You can install a SYN filter on a promiscuous listen socket

A SYN filter will be called for every arriving SYN that matches to that listen socket, and can examine the full SYN contents to make a decision on what to do with it

Normally, an arriving SYN is entered into the syncache, a SYN|ACK response is sent, and if/when the ACK to the SYN|ACK arrives, a new socket is created and queued on the listen socket

The SYN filter runs before the arriving SYN is entered into the syncache, and it can

- Silently reject the SYN
- Reject the SYN with a RST
- Accept the SYN, in which case normal syncache processing immediately proceeds
- Defer the decision by taking the SYN away and resubmitting it to the syncache in the future with one of the above dispositions

/promiscuous sockets/SYN filter

SYN filters allow more complex matching for listen sockets than specific/wildcard selectors for VLAN tag stack, IP, and port

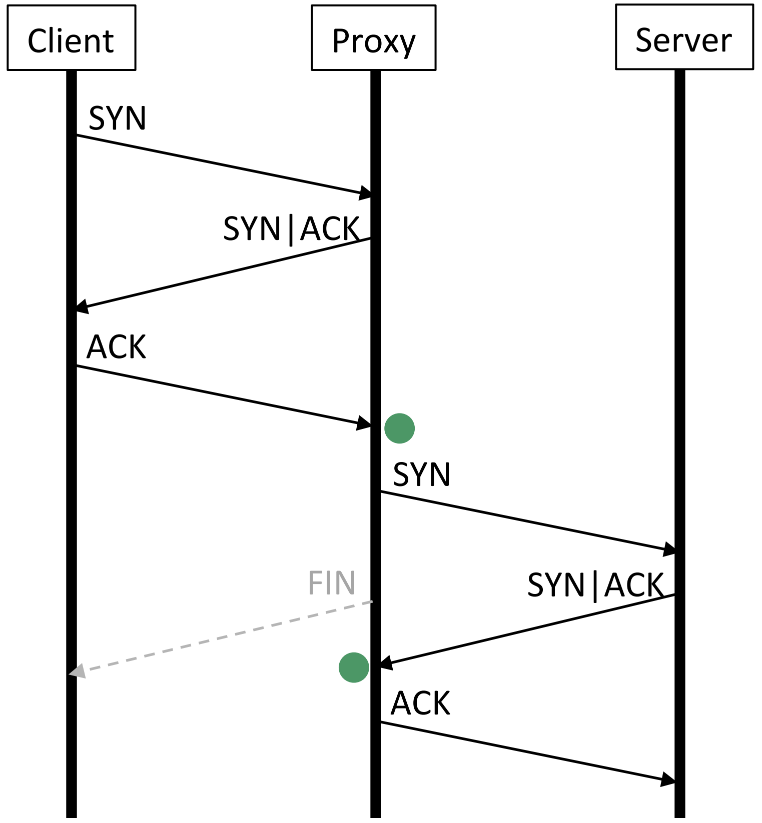
SYN filters also allow more desirable proxy behavior, as a SYN filter can be used to defer responding to the client until a connection can be established with the server (see next slide)

Things to keep in mind when implementing SYN filters:

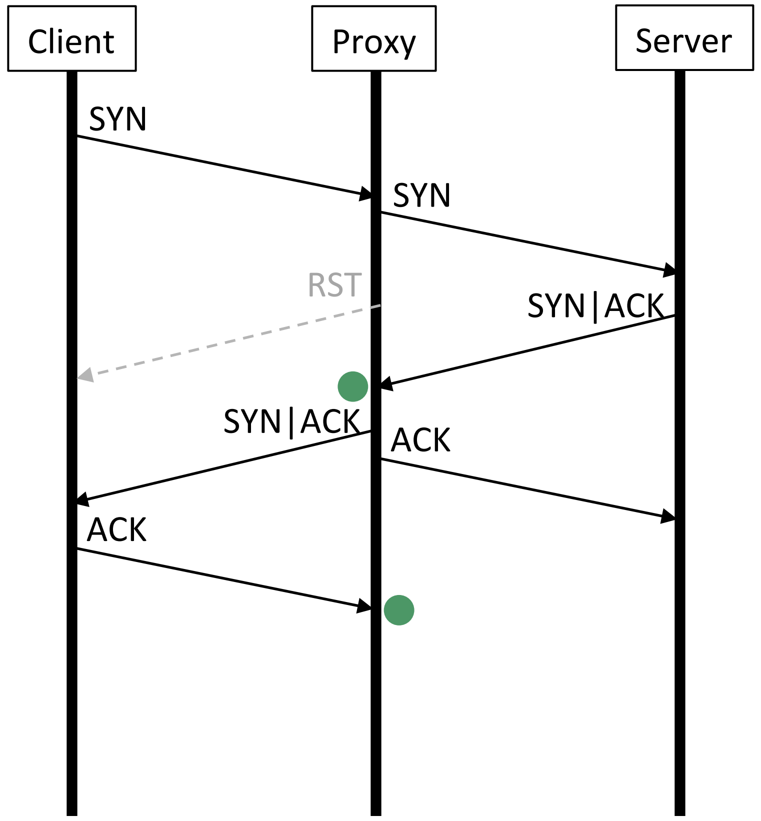
- The SYN filter has to be prepared for multiple invocations due to retransmitted SYNs
- Even though the syncache is being used (and even syncookies still work), a SYN filter implementation can defeat the effectiveness of the foregoing if it allocates significant resources on every SYN

/promiscuous sockets/SYN filter

Proxy without SYN filter



Proxy with SYN filter



/promiscuous sockets/API

To create a promiscuous socket, create a socket as usual, then set socket options

- setsockopt(SO_PROMISC, 1)
- setsockopt(SO_SETFIB, cdom)
- setsockopt(SO_REUSEPORT, 1)
- setsockopt(IP_BINDANY, 1)
- setsockopt(SO_L2INFO, [VLAN_TAG_NONE, VLAN_TAG_ANY, {vlan tag stack}]), if listen socket
- setsockopt(SO_L2INFO, { [VLAN_TAG_NONE, {vlan tag stack}], src_mac, dst_mac}), if active socket

Promiscuous socket binding

- Listen sockets can bind({IP, port}), where 0 indicates a wildcard for either
- Active sockets can bind({IP, port}), where neither is 0

Following accept(), getsockopt(SO_L2INFO) can be used to retrieve the VLAN tag stack and MAC addrs for the new connection, which currently are those present on the final ACK in the handshake

SYN filters are modeled after the existing accept filters and can be installed on a listen socket via setsockopt(IP_SYNFILTER, {syn filter spec}), which is wrapped in the uinet API with a routine that takes a callback pointer from the application

/promiscuous sockets/scalability

Decision was made to keep all connections in one pcb table
- Don’t know how connections will be distributed across tuple space, so other approaches present resource management issues
- Using a more expensive hash for promiscuous mode connections that has good distribution across hash buckets under a variety of scenarios (non-promiscuous still uses regular hash approach)

Tested up to 1 million listen sockets plus 1 million active, working connections.
- Various distributions across tuple space, million IPs, million VLANS (3 levels), mixture
- This works mostly because of the quality implementation of the FreeBSD stack, all I have done is arrange for suitable hashing for an absurd numbers of things
- Around this point, the number of timers in the callwheel starts to get awkward, which is to say, callwheel processing starts to consume significant CPU time due to large numbers of callouts in individual callwheel slots

/passive receive

Passive receive is another new socket mode that allows for passive reconstruction of both data streams in a TCP connection using a copy of the packet stream.

One of the main differences between regular TCP operation and passive TCP operation is that you cannot count on retransmits happening if a packet is missing

- The packet may be missing only from the local point of view, due to a drop in the copy data path between the actual packet stream and the observer
- The packet may have gone missing in the actual packet stream, but the actual retransmit may have been dropped in the copy data path

Another difference is that the result of an accept() is two sockets instead of one - each socket will provide the reconstructed stream received by one of the connection endpoints

And finally, one has to consider that the receive window of the passive reconstructor is subject to overrun if it is not sized by the application to be large enough to handle the received stream processing latency

/passive receive/missing packets

Passive receive handles the missing packet issue in several ways

The mbuf infrastructure has been modified (sufficiently, not completely) to allow an mbuf to represent a hole (a place marker for a given number of zeroes)

soreceive() has been modified to fill the output buffer with zeroes from mbuf holes, and to be able to break at hole boundaries and report whether the returned data is hole fill or not

As the missing packet(s) may contain FINs for the connection, timers must be used to prevent connections from hanging around indefinitely

- The existing idle and keepalive timers can be used to trigger connection death
- Alternatively, the application has to handle this

Missing packets can stall the TCP reassembly queue indefinitely, so a reassembly deadline timer is used to generate a hole and continue progressing when it expires. Note this also takes care of missing IP fragments.

/passive receive/general approach

The general approach is to perform the reconstruction for each stream almost exclusively using the packet flowing in that streams direction, due to potential ordering issues between the two streams

The exception is at connection establishment time, when both the SYN and corresponding SYN|ACK are looked for in sequence in order to establish which TCP options are in use

Otherwise, everything proceeds through the syncache as usual, until the handshake is complete, at which point two sockets are created instead of one

The primary socket (representing the endpoint that received the initial SYN) is obtained via accept(), and the secondary socket (representing the endpoint that sent the initial SYN) is obtained from the primary socket

Note that the secondary socket TCP state machine is kicked off by synthesizing the SYN|ACK it should have seen from the syncache entry and submitting it to tcp_do_segment()

/passive receive/API

To create a passive receive socket, create a listen socket as usual, then

- setsockopt(SO_PASSIVE, 1)

After accept(&newso), the secondary socket can be obtained via get_passive_peer(newso)

Sockets created from a passive receive listen can be read from as usual using soreceive(). If there are holes in the data due to missing packets, zeroes will be filled in by soreceive().

To detect holes in the received data:

```
flags=MSG_HOLE_BREAK;
soreceive(…, &flags);
if (flags & MSG_HOLE_BREAK)
    all returned data is hole fill
else
    all returned data is real
```

Passive receive and promiscuous sockets functionality can be used together. To do so, create a promiscuous listen socket as desired then set SO_PASSIVE.

/performance

The focus on libuinet development to date has been on correctness of the port, new feature implementation, correctness of those new features, and scaling out promiscuous mode connections

Just beginning to enter the performance analysis phase of development

There are clear indications that there is work to be done

- For example, throughput on a single-connection netcat type test using libuinet was recently measured to be only about 70% of throughput using the kernel stack

There are obvious places to start

- As mentioned earlier, pcpu and locking implementations
- Improved use of netmap new zero-copy support

/future work

Focus on performance

Plumb promiscuous sockets support through ipv6, now that it is proven out under ipv4

Efficiency features for transparent proxies

- Introduce bridging options that could allow transition from bridging to transparent proxying

Expose more stack functionality through top level API

Upgrade to 10-series

Make system (most likely autoconf, given portability goals)

Integrate contributed sysctl tool

/delayed ACKs

To Juli Mallett <jmallet@freebsd.org>, for being persistent in connecting me to this project, for being a sounding board throughout development, for suffering through the first libuinet integration…

To the sponsors of this work, all of whom that wish to be named are listed below, along with their logos, which link to their websites:

listen(room, -1)