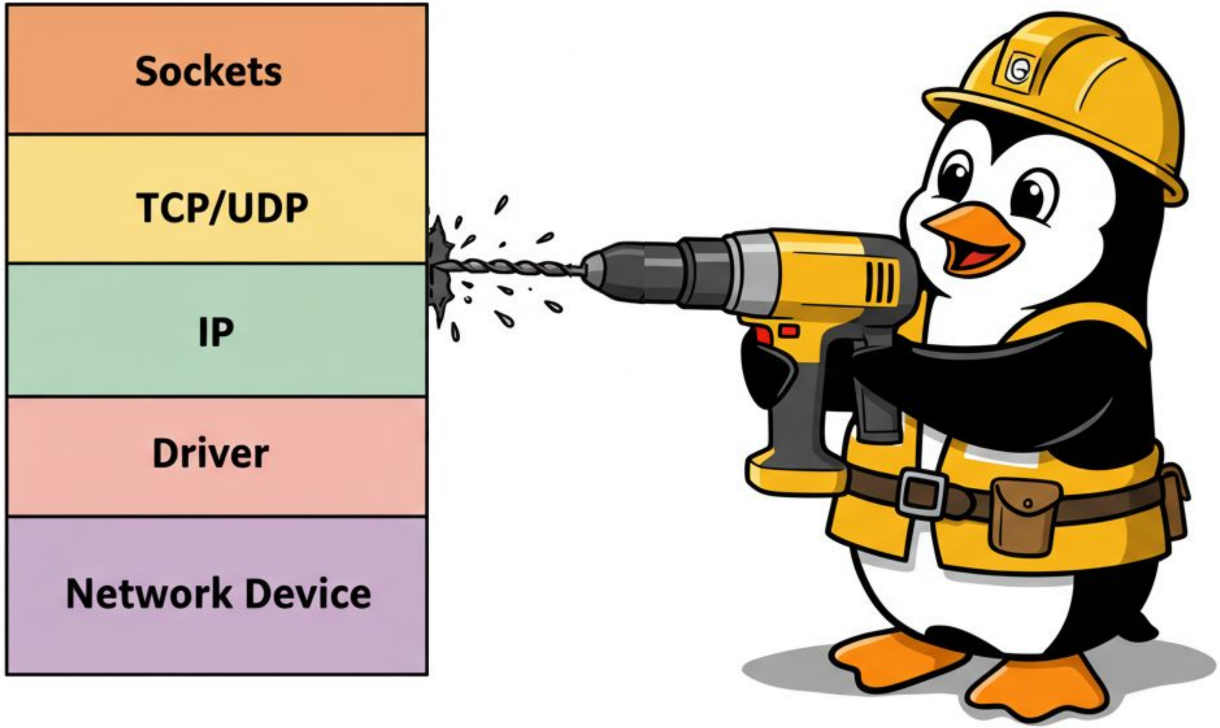
Using Packetdrill: a Power Tool for Automated Testing of the Linux Networking Stack

Neal Cardwell
@ Google

Netdev 0x19
Mar 2025



ACKs

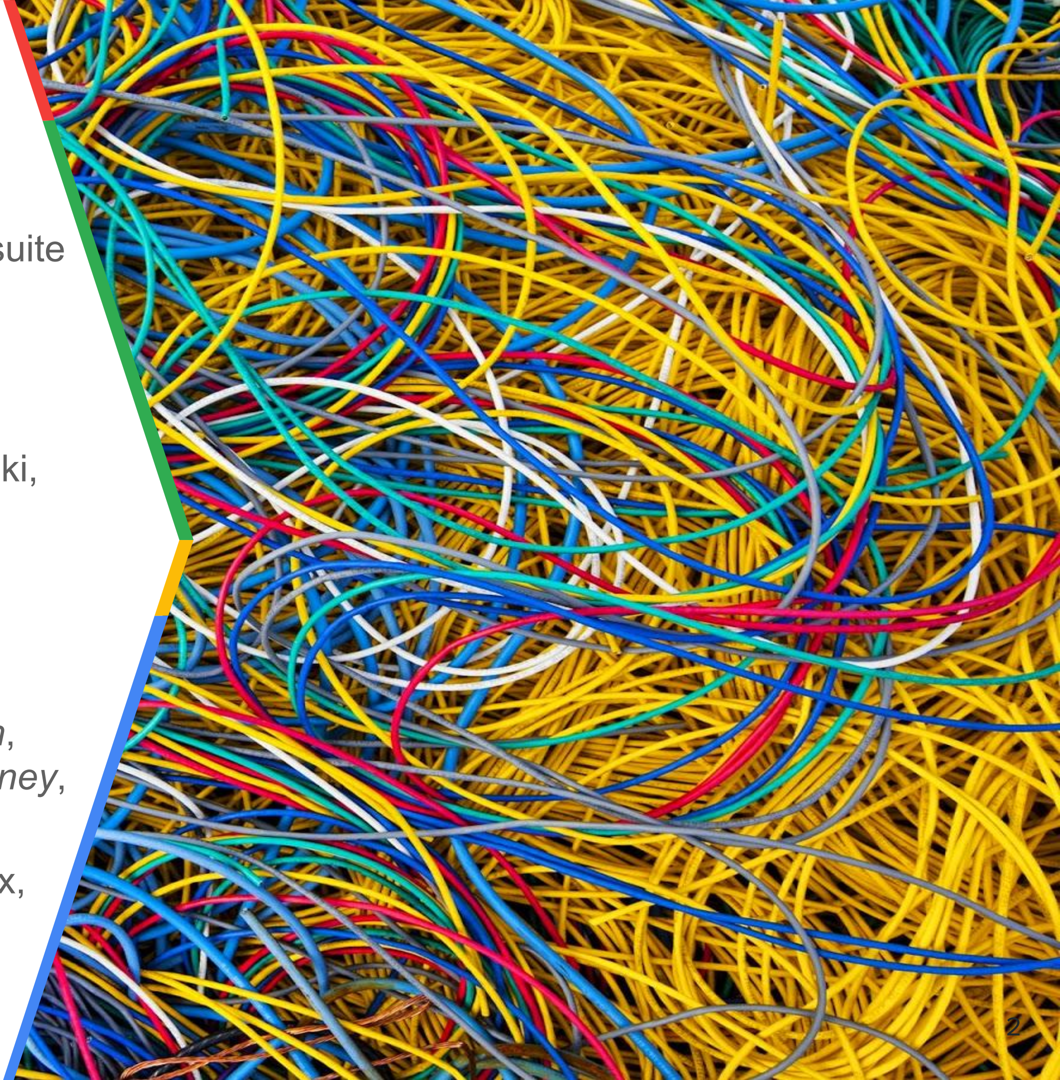
The packetdrill tool and the Linux packetdrill test suite are joint open source work by many members of various Networking software teams at Google:

Neal Cardwell, Eric Dumazet, Yuchung Cheng,
Willem de Bruijn, Shuo Chen, Maciej Żenczykowski,
Wei Wang, Kevin Yang, Soheil Hassas Yeganeh,
Yousuk Seung, Priyaranjan Jha,
Haitao Wu, David Morley, Yaogong Wang,
Nandita Dukkipati, Arjun Roy, *Luke Hsiao*,
Mubashir Adnan Qureshi, *Abdul Kabbani*,
Lawrence Brakmo, *Matt Mathis*, *Barath Raghavan*,
Hsiao-keng Jerry Chu, Andreas Terzis, *Mike Maloney*,
and *Tom Herbert,* ...   [ex-Googlers in italics]

... and many other generous members of the Linux,

*BSD, and IETF networking community...

Thank you!



Outline

- Introduction to packetdrill
  - What it is
  - Goals
  - Its scripting language
  - How to write and run tests
  - Design and implementation of local and remote mode
- Using packetdrill: basics
  - Varieties of packetdrill
  - How to download and build packetdrill
  - Troubleshooting
  - Best practices
  - Integrating packetdrill into your development workflow
  - Submitting patches
- Using packetdrill: advanced techniques
  - Testing protocol details
  - Tips and tricks

Part 1: Introduction to packetdrill

What is packetdrill?

- Packetdrill is a scripting tool for unit-testing network stacks
- For entire TCP/UDP/IPv4/IPv6 network stacks
    - From the system call layer down to the NIC hardware
- Works on Linux, FreeBSD, OpenBSD, and NetBSD; ports exist for MacOS
- Has two execution models for testing network stack behavior:
    - **Local**: Within a single machine using a tun virtual network device (for speed and ease)
    - **Remote**: Over a physical NIC on a physical LAN (for testing real drivers and NICs)
- Enables quick, precise (packet-header-level), reproducible, automated tests
- Open source since 2013
    - License: GPLv2, same as Linux kernel
- Interpreted, for fast edit/test cycles
    - Even on production machines w/o build tools

When is packetdrill useful?

- In what phases of software development is it useful?
  - Testing new changes, fixes, or features during development
    - Using black box unit tests
  - Automated regression testing
    - More precise and reproducible than netperf, load tests, or production testing
  - Troubleshooting
    - To replay traces, reproduce issues, test what-if theories
- What aspects of network stacks can it test?
  - Correctness - does the protocol implement the spec?
  - Reliability - does the state machine handle challenging corner cases well?
  - Interoperability - does it handle the kind of packets other stacks send?
  - Performance - are congestion control, loss recovery algorithms correct? (in tricky cases?)
  - Security - how does handle malicious messages?

When not to use packetdrill...

- What kinds of network testing is packetdrill **not** suited for?
  - High-speed or long-duration performance testing (e.g., CPU usage, throughput, latency)
    - Instead, use neper or netperf or iperf/iperf2/iperf3
  - Testing protocols above layer 4 (e.g., HTTP, etc.)
    - Instead, use load testing tools
  - Fuzzing
    - Instead, use: syzkaller

packetdrill scripting language: design goals

- Interpreted
  - For fast edit/test cycles (no make/compile/link/scp required)
  - Even on production machines w/o build tools
- Easy to write/ read for kernel network stack developers accustomed to...
  - Writing/reading **C code**
  - Looking at **strace** dumps to understand application system call interactions w/ the kernel
  - Looking at **tcpdump** traces to understand network stack behavior on the wire
- Feasible to turn strace + tcpdump traces from production into test cases
- Encourage simple, succinct tests
  - Encourages each script to be simple and easy to write and read
  - A description of one simple scenario
  - No conditionals, loops, or variables

packetdrill scripting language: design elements

- Comments
- System calls
- Packets
- Shell commands
- Python scripts

packetdrill: comments

- Comments
  - Document the intent for readers/maintainers
- Syntax:
  - C or C++ syntax

Examples:

```
/* C-style comments work */
// C++-style comments work
```

packetdrill: system calls

- System calls: strace-like syntax
  - system calls to invoke
  - output/return value to expect
  - blocking or non-blocking (only one blocking system call at a time)
- Syntax:
  - strace-like syntax

Example:

```
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
```

packetdrill: packets

- Packets: tcpdump-like syntax
  - inbound packets to construct and inject into the network stack under test
  - outbound packets to expect and sniff and verify wrt timing and contents
  - TCP, UDP, ICMP
- Syntax:
  - tcpdump-like packet syntax prefixed by a pipe-inspired < (inbound) or > (outbound) specifier

Examples:

Inbound (inject):

```
< S 0:0(0) win 32792 <mss 1000,nop,nop,sackOK,nop,wscale 6>
```

Outbound (expect/sniff/verify):

```
> S. 0:0(0) ack 1 <mss 1460,nop,nop,sackOK,nop,wscale 7>
```

packetdrill: shell commands

- shell commands
  - To configure or inspect the machine under test
  - May use: ip, sysctl, tc, set device/module parameters ...
- Syntax:
  - Regular bash commands enclosed in single-tick quotes: ` `

Example:

```
`sysctl -q net.ipv4.tcp_congestion_control=cubic`
```

packetdrill: Python scripts

- Python scripts
  - Used to check/print internal socket state: TCP_INFO, TCP_CC_INFO, SO_MEMINFO
- Syntax:
  - Regular Python code enclosed in `%{ }%` braces
  - Can be multi-line scripts that assert, define and call Python functions, print output, etc.

Example:

```
%{ assert tcpi_snd_cwnd == 7 }%
```

packetdrill: specifying event times

- Every event in a packetdrill script starts with a time specifier: "when should this happen?"
- packetdrill allows flexibility in timing assertions
- Supported timing models:
  - Absolute: `0.100` `// event should happen 100ms since test start`
  - Relative: `+0.100` `// should happen 100ms since previous event`
  - Range: `0.100~0.200` `// should happen 100ms to 200ms since test start`
  - Relative range: `+0.100~+0.200` `// should happen 100ms to 200ms since previous event`
  - Wildcard: `*` `// test doesn't care; event can happen any time`
- Checking time:
  - Timing is critical for reliability and performance (loss recovery, congestion control, etc)
  - When a network stack event happens at an unexpected time, packetdrill raises a test failure
  - To avoid flakes, the default tolerance for timing variation: 4ms
  - Command line option to change the tolerance for all events (useful for debug or KASAN kernels):
    - `--tolerance_usecs=8000`
- blocking system calls
  - Provide a specified start and an expected end time separated by `...`
  - Syntax: `0.100...0.200`

packetdrill: setup and cleanup

- A test can use two special (optional) commands for **Setup** and **Cleanup**
  - If specified, they are always run, even if a test fails
  - They do not have a time specifier (their time is implicitly the start and end of the test, respectively)
  - Their execution is not timed, so can be as slow as it needs to be
- **Setup**:
  - Intended to set initial host / namespace configuration via: ip, tc, sysctl, etc...
- **Cleanup**:
  - Intended for checking behavior and cleaning up any changes made in setup
  - Always runs at test conclusion, even if test fails in the middle

Example:

```
// Setup: this test will specifically test reno congestion ctrl:
`sysctl -q net.ipv4.tcp_congestion_control=reno`

// ...timed system calls and packets to test reno...

// Cleanup: Now let's restore our state; always runs, even if test fails
`sysctl -q net.ipv4.tcp_congestion_control=cubic`
```

The packetdrill ellipsis construct: `...`

- `...` means "I don't care about this detail; just make it work"
  - Tells packetdrill to fill in boring boilerplate in the expected way to make things work
- Handy for several reasons:
  - 1: Eases writing of tests: you don't have to choose details to use (addresses, buffer contents)
  - 2: Eases reading of tests: you don't have to read unimportant details
  - 3: Allows scripts to be reused by being agnostic about details that can vary
    - Address family (AF_INET/AF_INET6) or addresses can vary

IP versions: ipv4, ipv6, ipv4-mapped-ipv6

- packetdrill supports 3 IP address family modes:
    - ipv4 AF_INET sockets, IPv4 packets and addresses
    - ipv6 AF_INET6 sockets, IPv6 packets and addresses
    - ipv4-mapped-ipv6 AF_INET6 sockets, IPv4 packets and addresses
- Specify the mode on the command line via --ip_version flag:
    - --ip_version=[ipv4,ipv4-mapped-ipv6,ipv6]
- Best practice: test scripts are written to be run in all 3 modes using `...`
    - All address-family-specific system call inputs/outputs are elided with `...`

```
0   socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
+0  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
+0  bind(3, ..., ...) = 0
+0  listen(3, 1) = 0
// SYN/SYNACK/ACK packets go here
+0  accept(3, ..., ...) = 4
```

Putting it all together...

- The following slide has a complete example packetdrill script
  - Testing loss recovery (fast recovery) and congestion control (CUBIC)
  - A typical packetdrill test for Linux TCP

```
Key:      BLACK: system call                              (strace syntax)
          BLUE:  input: incoming injected packet          (tcpdump-style syntax)
          RED:   output: outgoing sniffed packet          (tcpdump-style syntax)
          GREEN: Python script                            (Python code)
```

```
// Test Fast Recovery and CUBIC cwnd response to Fast Recovery.
`sysctl -q net.ipv4.tcp_congestion_control=cubic`  // shell command configures host

0   socket(..., SOCK_STREAM, IPPROTO_TCP) = 3             /* C-style comments work */
+0  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0 // C++-style comments work
+0  bind(3, ..., ...) = 0
+0  listen(3, 1) = 0
+0  < S 0:0(0) win 32792 <mss 1000,nop,nop,sackOK,nop,wscale 6>
+0  > S. 0:0(0) ack 1 <...>          // <...> means: don't care about outgoing options
+.1 < . 1:1(0) ack 1 win 257
+0  accept(3, ..., ...) = 4

+0  %{ assert tcpi_snd_cwnd == 10 }%  // check socket state from TCP_INFO
+0  write(4, ..., 4000) = 4000        // ask kernel under test to send 4 packets
+0  > P.  1:4001(4000) ack 1          // expect to sniff data pkts w/ these seqs/flags

+.1 < . 1:1(0) ack 1 win 257 <sack 1001:2001,nop,nop> // inject dupack #1
+0  < . 1:1(0) ack 1 win 257 <sack 1001:3001,nop,nop> // inject dupack #2
+0  < . 1:1(0) ack 1 win 257 <sack 1001:4001,nop,nop> // inject dupack #3
+0  > . 1:1001(1000) ack 1 // immediately after 3 dupacks we expect a fast retransmit!

+.1 < . 1:1(0) ack 4001 win 257                       // retransmit repaired loss
+0  %{ assert tcpi_snd_cwnd == 7 }%  // check CUBIC cwnd was cut by expected 30%
```

packetdrill local and remote modes

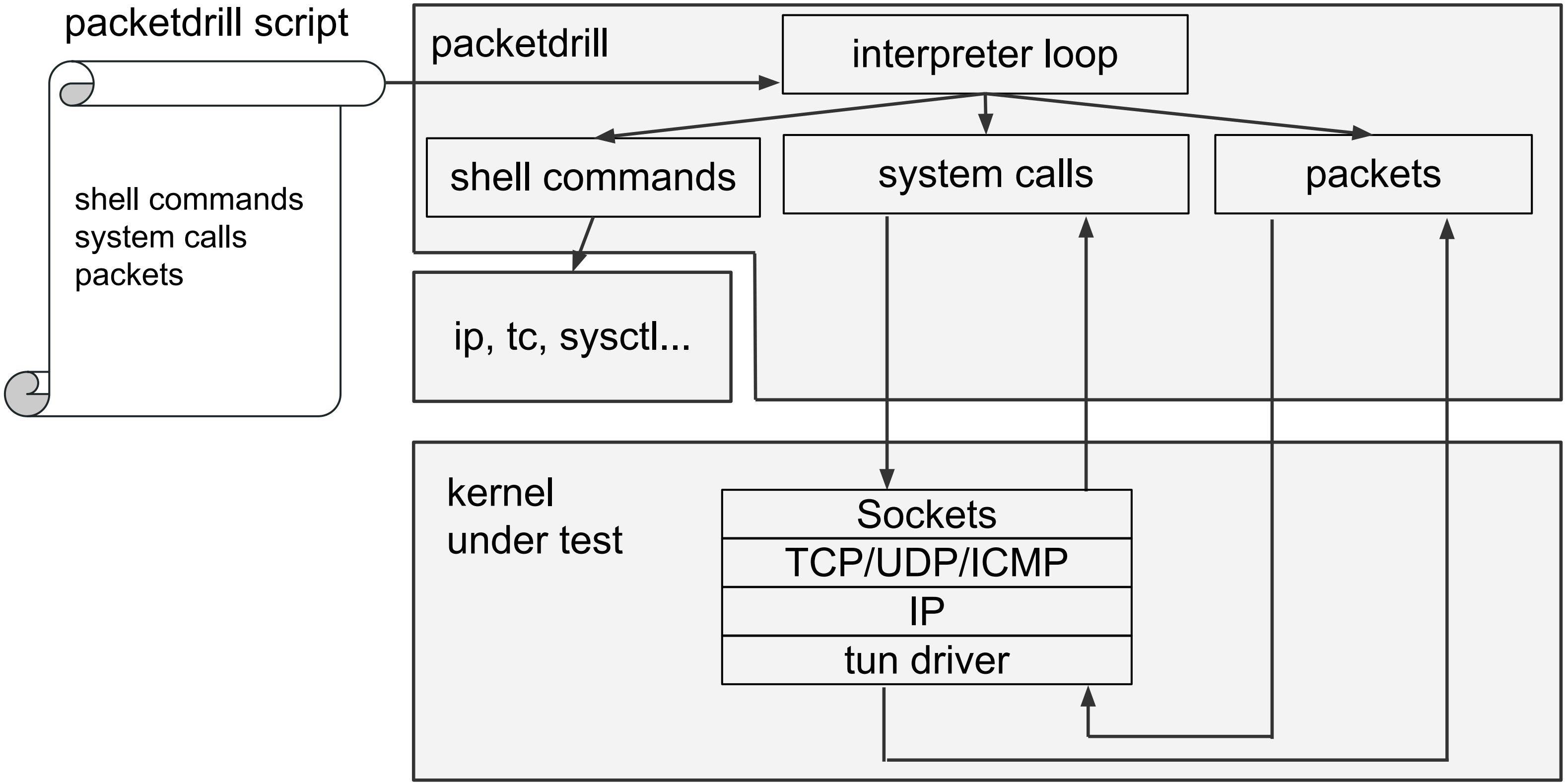
- Two execution models for testing network stack behavior:
  - **Local**: Within a single machine using a tun virtual network device (for speed and ease)
    - Single-machine testing using TUN virtual network device
      - Tests sockets, L4 (TCP/UDP/ICMP), L3 (IP)
    - A single packetdrill process
      - Runs system calls and shell commands
      - Injects packets via tun virtual network device, sniffs and verifies packets via tun
  - **Remote**: Over a physical NIC on a physical LAN (for testing real drivers and NICs)
    - Two-machine testing of real NICs over a LAN
      - Tests L4, L3, L2, L1, including driver, offload mechanisms, NIC, LAN
    - Machine 1: a client packetdrill process running on the kernel under test
      - Runs system calls and shell commands
    - Machine 2: a server packetdrill process running on a remote machine
      - Injects packets over real LAN, sniffs and verifies packets over real LAN

Running a packetdrill test in local mode

- Local mode is the default mode for test execution
- Run a packetdrill process as root on a single machine
- Need to provide:
  - The list of path names of script to execute
- Example:

```
test_machine# ./packetdrill foo.pkt
```

packetdrill local mode: design and implementation



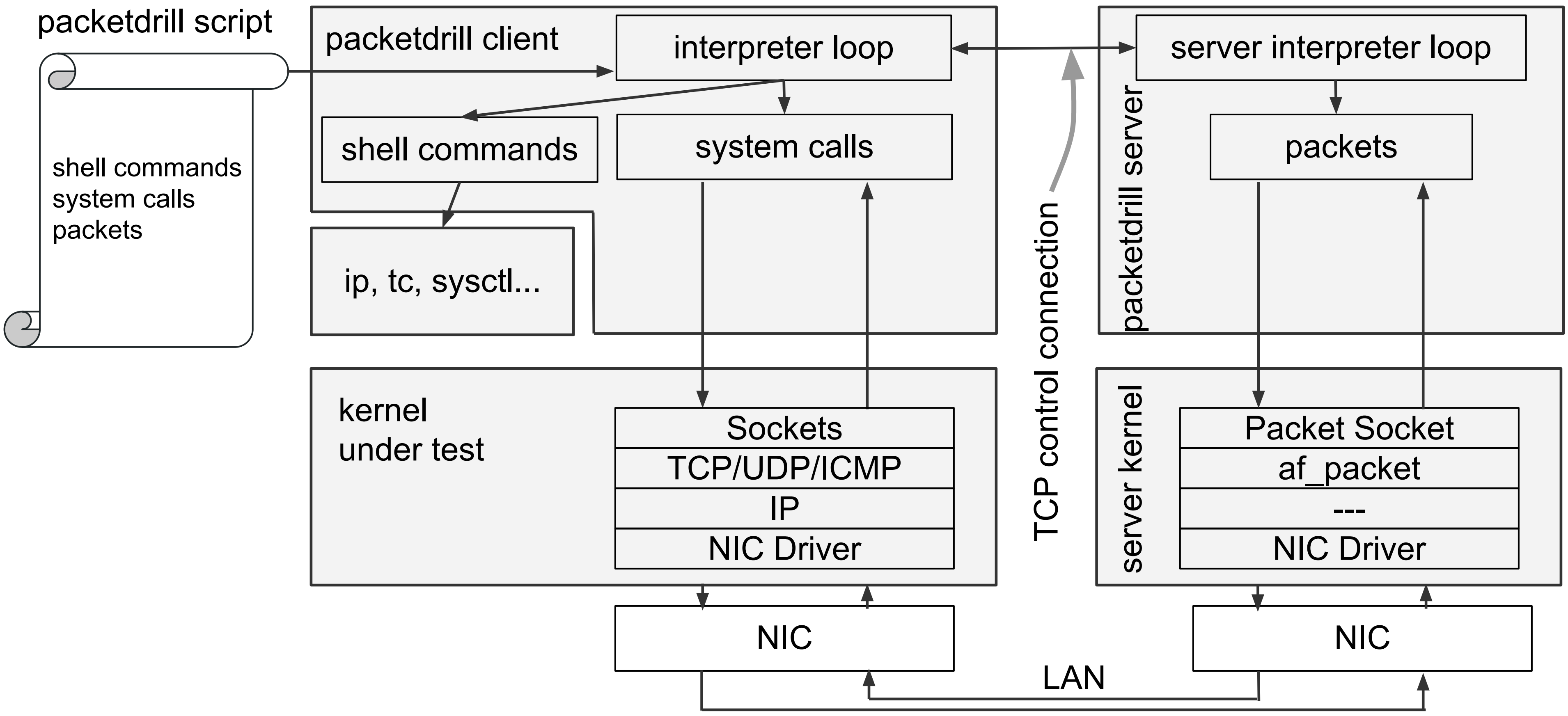
Running a packetdrill test in remote mode

- Remote mode requires an extra parameter for test execution
- Run a packetdrill process as root on two machines: a client and a server
- Need to provide:
  - The list of path names of script to execute
  - The DNS name or IP address of the server machine
- Example:

```
server_machine# ./packetdrill --wire_server

client_machine# ./packetdrill --wire_server_at=1.2.3.4 foo.pkt
```

packetdrill remote mode: design and implementation



Reusing scripts between local and remote mode

- The same test can be executed in both local mode and remote mode
- This works as long as network stack behavior asserted by test is the same
- Note: Sometimes MTU/MSS are different on different devices
  - Tests can often simply work around this with <...> to skip checking options
  - The earlier example script can run in both local and remote mode using this one weird trick

Part 2:
The basics of using packetdrill

Varieties of packetdrill

- There are several varieties/forks of packetdrill:
  - Original packetdrill by our team at Google:
    - https://github.com/google/packetdrill
  - Version with support for MPTCP, concurrent connections:
    - https://github.com/aschils/packetdrill_mptcp
  - The nplab packetdrill: supports UDPLite, SCTP, FreeBSD, MacOS:
    - https://github.com/nplab/packetdrill
  - A packetdrill for QUIC, at Apple:
    - 'Testing QUIC with packetdrill'
    - Unreleased (so far)

packetdrill test suites for Linux

- Current test suite status
  - In 2024 (Linux v6.12), 72 tests added to Linux tree (e.g. [tools/testing/selftests/net/packetdrill]())
  - 167 in Github at https://github.com/google/packetdrill
  - Over 1100 internally used by our team
    - For 13 years the Google Linux kernel networking team has used packetdrill scripts as a core part of continuous testing of the production kernel used on Google's machines
    - Gradually open-sourcing (Google => mainline) and porting (github => mainline)
- Example areas of coverage for Linux TCP ([gtests/net/tcp/]() subdirectories):

```
blocking          epoll              limited_transmit  sendfile          ts_recent
close             fast_recovery      md5               shutdown          user_timeout
common            fast_retransmit    mss               slow_start        validate
cubic             fastopen           mtu_probe         splice            zerocopy
cwnd_moderation   gro                nagle             syscall_bad_arg
ecn               inq                notsent_lowat     tcp_info
eor               ioctl              sack              timestamping
```

Downloading and building Google packetdrill

- First install the dependencies

```
sudo apt install git gcc make bison flex python net-tools  # for Debian/Ubuntu
```

- Then download the source code from https://github.com/google/packetdrill

```
git clone https://github.com/google/packetdrill.git
```

- Then build the packetdrill binary

```
cd ~/packetdrill/gtests/net/packetdrill

./configure

make
```

Running the github packetdrill test suite for Linux

- To run all tests in the packetdrill Linux test suite at https://github.com/google/packetdrill ...
- The easiest way is to use the run_all.py script by Willem de Bruijn:

```
cd ~/packetdrill/gtests/net/

sudo ./packetdrill/run_all.py -S -v -l tcp/
```

- This runs all test scripts with all 3 supported address families
  - ipv4, ipv6, ipv4-mapped-ipv6 (IPv6 sockets w/ IPv4 addresses)
- Explaining the command line options (see: ./packetdrill/run_all.py --help):
  - -S --serialized (run one test at a time; slower than parallel; avoids timing flakes)
  - -l --log_on_error (print stderr and stdout for failed tests)
  - -v --verbose (required for --log_on_error)

Main types of packetdrill errors

- Syntax errors
- Semantic errors (e.g., packet size doesn't match sequence number range)
- System call errors
    - Return value or completion time of blocking system call did not match expectations
- Packet errors
    - Contents or timing of outbound packet did not match expectations
    - Timed out waiting for outbound packet
- Shell command errors
    - Non-zero exit status (e.g., grep nstat output to check SNMP counter values)
- Python errors, including assertion failures
    - Show Python output and Python stack trace

How to interpret packetdrill test failures

- Packetdrill error message format:
  - <script_file_name>:<line_number>: <error_details>
- Example:

```
Script line:
+0  > . 3001:4001(1000) ack 1 // immediately after 3 dupacks we expect a fast retransmit!

Error:
test.pkt:21: error handling packet: live packet field tcp_seq: expected: 3001 (0xbb9) vs actual: 1 (0x1)
script packet:  0.200812 . 3001:4001(1000) ack 1
actual packet:  0.200808 . 1:1001(1000) ack 1 win 502
```

- How to interpret this error:
  - Script named test.pkt had an error on line 21
  - The script expected to sniff an outbound packet at time `0.200812` that looked like:
    - `. 3001:4001(1000) ack 1`
  - The network stack under test instead actually sent a packet at time `0.200808` that looked like:
    - `. 1:1001(1000) ack 1 win 502`
  - So basically the network stack sent a packet with an unexpected TCP sequence number range

Troubleshooting packetdrill test failures

- If a test fails, it can be useful to re-run and acquire more data
- 3 levels of detail can often be useful:
  - Run packetdrill with --verbose
    - Shows which system calls and packets happen, and when
    - Shows all variable values from TCP_INFO, etc, that are available in Python
  - Run packetdrill with --debug
    - Packetdrill shows function-level gory details
    - Useful for debugging packetdrill itself
  - Use strace on packetdrill and tcpdump in parallel
    - sudo tcpdump -n -i any port 8080 &
    - sudo strace -ttt --follow-fork packetdrill foo.pkt
      - -ttt for microsecond-level timestamps
      - --follow-fork to trace all threads

High-level advice for writing packetdrill tests

- Standard advice for writing unit tests applies...
- Add comments for future readers/maintainers (they will be debugging failures)
  - At top of file, explain what behavior is being tested/verified
  - Explain the key moments of stimulus/input: // Here we inject a special X packet ....
  - Explain the expected result: // Here we expect the kernel to send a Y packet because Z:
- Keep tests small and focused
  - And make test names clearly convey the functionality/scenario being tested
  - Eases review, maintenance, interpreting and root-causing failures
- Don't assert/check outside that focus (i.e., behavior test isn't focusing on)
  - e.g., only tests specifically for TCP receive window code should check receive window values
  - Minimizes/focuses toil/churn when the network stack behavior changes
- Make dependencies minimal and explicit
  - Avoid assuming specific config (receive buffer size, congestion control,...) because these vary
  - But if a test depends on something, set or check that explicitly

Best practices for configurations and network namespaces

- (1) Have a script to set config values you expect (sysctl / module parameters)
- (2) Run packetdrill tests inside a network namespace
  - Avoids tests accidentally changing global config parameters
  - Changing global config parameters can cause mysterious/hard-to-debug test failures
    - e.g., congestion control algorithm, receive buffer sizes, etc
- (3) Have explicit tests for the sysctl parameters you care about
  - There can be bugs (e.g., conflict resolution bugs) in simple code to initialize sysctl parameters
  - These can be hard to find if you are using (1)
  - So have tests that check:
    - Default global sysctl parameter values
    - Default values for per-netns sysctl values

Ways to integrate packetdrill into your workflow

- Can be useful:
  - Reproducing bugs or replaying traces
  - Testing what-if theories
  - git commit message documentation for bug fixes or feature additions
- Highly recommended:
  - Unit tests during network stack development (bug fixes, new features)
  - Running full regression suite before sending changes for review/merge
  - Automated continuous regression testing in a CI/CD pipeline (smp, debug, KASAN)

Contributing patches for packetdrill

- We are happy to incorporate fixes into packetdrill
  - And small-to-medium-sized features, as time permits!
- To contribute patches, please follow the recipe here; mainly:
  - Join the packetdrill mailing list on Google groups
  - Verify that you can certify the origin of your code with a Signed-off-by footer, according to the standards of the Linux open source project
  - Use scripts/checkpatch.pl from the Linux source tree to check the style of the C code
  - Please use a commit first/summary line like:
    - net-test: packetdrill: add new packetdrill support for foo
    - net-test: add new test cases for foo feature
  - Two ways to submit patches:
    - Github pull request
    - Use git commit/format-patch/send-email to the packetdrill mailing list on Google groups

Part 3:
Advanced packetdrill techniques

Tips on MSS and MTU

- Making tests work with all address families
  - Use --mtu=1520 with ipv6, since IPv6 headers are 20 bytes bigger
  - run_all.py handles this detail when running tests w/ all 3 address family modes
- For easy writing/reading of seq/ack, it's easiest to make MSS 1000

Tips on reducing timing flakes

- Use relative timestamps when possible
  - They avoid failures due to accumulating timing errors due to CPU / timer slowness
- Be careful with shell commands
  - They can take tens of milliseconds due to disk seeks, machine-wide synchronization, etc
  - So try to only use them in setup/cleanup blocks, which are untimed
- Be careful with exponential backoff in RTOs
  - Exponential backoff magnifies noise
  - Linux jiffy-granularity timers are up to 12.5% slower than you expect due to timer wheel impl
  - You may need to allow a relative range:
    - `+0.100~+0.200 // should happen 100ms to 200ms since previous event`

Defining symbols from command line with -D name=val

- Sometimes you want to reuse a test script with slight twists
  - e.g., some aspects of IPv4 and IPv6 differ beyond IP addresses
- packetdrill scripts can use generic symbols
- Invocations define macro-style symbols from command line with -D name=val
  - Much like: gcc -D name=val ...
- run_all.py includes some standard mappings; for example:

```
A script can use something like:
  cmsg_type=CMSG_TYPE_RECVERR,
  cmsg_data={ee_errno=ENOMSG,

For IPv4 run_all.py invokes packetdrill with:
  -D CMSG_LEVEL_IP=SOL_IP -D CMSG_TYPE_RECVERR=IP_RECVERR

For IPv6 run_all.py invokes packetdrill with:
  -D CMSG_LEVEL_IP=SOL_IPV6 -D CMSG_TYPE_RECVERR=IPV6_RECVERR
```

Testing ICMP with packetdrill

- Useful for testing how TCP and UDP deal with incoming ICMP errors
- For example, testing how TCP handles packets that are too big:

```
// TCP/IPv4 PMTU discovery:
   +0 write(4, ..., 1460) = 1460
   +0 > P. 1:1461(1460) ack 1
// ICMP says that segment was too big:
+.005 < icmp unreachable frag_needed mtu 1200 [1:1461(1460)]
// TCP picks a packet size using the MTU from the message, and retransmits ASAP:
   +0 > P. 1:1461(1460) ack 1
```

```
// TCP/IPv6 PMTU discovery:
   +0 write(4, ..., 1460) = 1460
   +0 > P. 1:1461(1460) ack 1
// ICMP says that segment was too big:
   +0 < icmp packet_too_big mtu 1280 [1:1461(1460)]
// TCP picks a packet size using the MTU from the message, and retransmits ASAP:
   +0 > P. 1:1461(1460) ack 1
```

Testing UDP with packetdrill

- Useful for testing how UDP deals with MTU, incoming ICMP errors, TOS, etc.
- For example:

```
// Create and connect a UDP socket:
    0 socket(..., SOCK_DGRAM, IPPROTO_UDP) = 3
 +.01 connect(3, ..., ...) = 0
   +0 getsockopt(3, IPPROTO_IP, IP_MTU, [1500], [4]) = 0

// Send the biggest possible UDP/IPv4 packet (without fragmentation).
   +0 write(3, ..., 1472) = 1472
   +0 > udp (1472)

// Check behavior with an MTU of net.ipv4.route.min_pmtu = 552 (512 + 20 + 20)
 +.01 < icmp unreachable frag_needed mtu 552 [udp (1472)]

// Verify we get an EMSGSIZE upon read() and can also read incoming packets:
 +.01 < udp (1472)
   +0 read(3, ..., 2000) = -1 EMSGSIZE (Message too long)
   +0 read(3, ..., 2000) = 1472
```

Encapsulation

- packetdrill can inject/sniff any combo of several common encap formats:
  - IPv4, IPv6, GRE, MPLS
- Syntax: separate encap header specs with : (colon) characters
- Examples:

```
// IPIP encap:
+0 > ipv4 1.1.1.1 > 2.2.2.2 : . 1:1001(1000) ack 1
// Double IPv6 encap:
+0 > ipv6 1::1111 > 2::2222: ipv6 3::3333 > 4::4444: . 1:1001(1000) ack 1
// Simple IPv4/GRE encap:
+0 > ipv4 1.1.1.1 > 2.2.2.2: gre: . 1:1001(1000) ack 1
// GRE encap with all GRE header fields specified:
+0 < ipv4 2.2.2.2 > 1.1.1.1 :
      gre flags 0xb000, sum 511, off 1023, key 0x80001234, seq 512 :
      . 1:1001(1000) ack 1 win 123
// GRE plus 2-entry MPLS encap:
+0 < ipv4 2.2.2.2 > 1.1.1.1 : gre :
     mpls (label 0, tc 0, ttl 0) (label 1048575, tc 7, [S], ttl 255) :
     . 1:1001(1000) ack 1 win 123
```

ECN: Explicit Congestion Notification

- packetdrill allows specifying the 2-bit ECN field in packets
- Useful for testing various flavors of ECN: Classic, DCTCP, L4S
- Syntax:

`[noecn]`	IP ECN field is 00; Not-ECT	sender transport (e.g., TCP) does not support ECN
`[ect0]`	IP ECN field is 10, ECT(0)	sender indicates “ECN-Capable Transport” (Classic)
`[ect1]`	IP ECN field is 01, ECT(1)	sender indicating “ECN-Capable Transport” (L4S)
`[ce]`	IP ECN field is 11, CE	set by network element to say “Congestion Experienced”

Example: injecting a packet with CE

```
+0 < [ce] P. 4001:4501(500) ack 3 win 257
```

Other IP header fields: TOS, flowlabel, TTL

- Packetdrill can set and check other interesting IP header fields...
- `tos`: ToS / QoS / DSCP / Traffic Class (packetdrill uses `tos` for both IPv4 and IPv6)
  - Useful for testing rx or tx processing of ToS, e.g. ToS reflection

```
// Inject SYN with tos 0x80 and verify outgoing SYNACK reflects incoming TOS:
+.1 < (tos 0x80) S 0:0(0) win 32792 <mss 1000,sackOK,nop,nop,nop,wscale 2>
 +0 > (tos 0x80) S. 0:0(0) ack 1 <...>
```

- `flowlabel`: IPv6 flowlabel
  - Useful for testing outgoing flowlabel for PLB load balancing [SIGCOMM '22], Protective ReRoute [SIGCOMM '23]

```
// Verify outgoing flowlabel matches (or is different from) previous flowlabel:
+0 > (flowlabel 0x1) P. 1001:2001(1000) ack 1
```

- `ttl`: TTL / Hop Limit (packetdrill uses `ttl` for both IPv4 and IPv6)
  - Useful for testing rx or tx processing of TTL

```
// Inject packet with TTL 100
+0   < (ttl 100) . 1:1(0) ack 201 win 1000
```

Part 4:
Wrapping up

For more information about packetdrill

- The Google packetdrill README at: https://github.com/google/packetdrill
- packetdrill: Scriptable Network Stack Testing, from Sockets to Packets
    - 2013 USENIX ATC
- Drilling Network Stacks with packetdrill
    - Usenix ;login: October 2013
- packetdrill mailing list on Google groups

The future...

Some features/directions that seem useful (contributions are welcome!)...

- A "black box recorder" option that will, upon test failure, dump a timestamped log of all test activity
  - To speed troubleshooting, especially for failed tests in automated test runs
- Optional Integration with UML and "Time Travel Mode" (NetDev 0x14)
  - To speed testing by running test suite in user mode on simulated time instead of booting a kernel and running on wall clock time

Conclusions

- Our team finds packetdrill useful for automated unit tests for Linux networking
  - We hope you'll find it useful too!
- Some packetdrill tests are now in mainline Linux: tools/testing/selftests/
- Please join us to make things even better in this open source ecosystem:
  - Contributing features/fixes for packetdrill
  - Contributing to the suite of packetdrill tests for Linux networking
  - Helping port packetdrill tests from: github => mainline or Google => mainline
- Thanks!