LibOS as a regression test framework for Linux networking

Hajime Tazaki

2016/02/12

netdev 1.2

outline

- libOS introduction
- testing framework introduction
- case studies
- QA

what is LibOS ?

- Library version of Linux kernel
- presented at netdev0.1, proposed to LKML (2015)

http://www.slideshare.net/hajimetazaki/library-operating-system-for-linux-netdev01

media

- LWN
  - https://lwn.net/Articles/637658/
- Phoronix
  - http://www.phoronix.com/scan.php?page=news_item&px=Linux-Library-LibOS
- Linux Magazine
  - http://www.linux-magazine.com/Issues/2015/176/Kernel-News
- Hacker News
  - https://news.ycombinator.com/item?id=9259292

how to use it ?

- Network Stack in Userspace (NUSE)
  - **LD_PRELOADed** application
  - Network stack personality
- Direct Code Execution (DCE, ns-3 network simulator)
  - Network simulation integration (running Linux network stack on ns-3)

what is NOT LibOS?

- *not only* a userspace operating system
- *not only* a debuging tool

- but LibOS *is*
  - a library which can link with **any programs**
  - a library to form any purpose of program

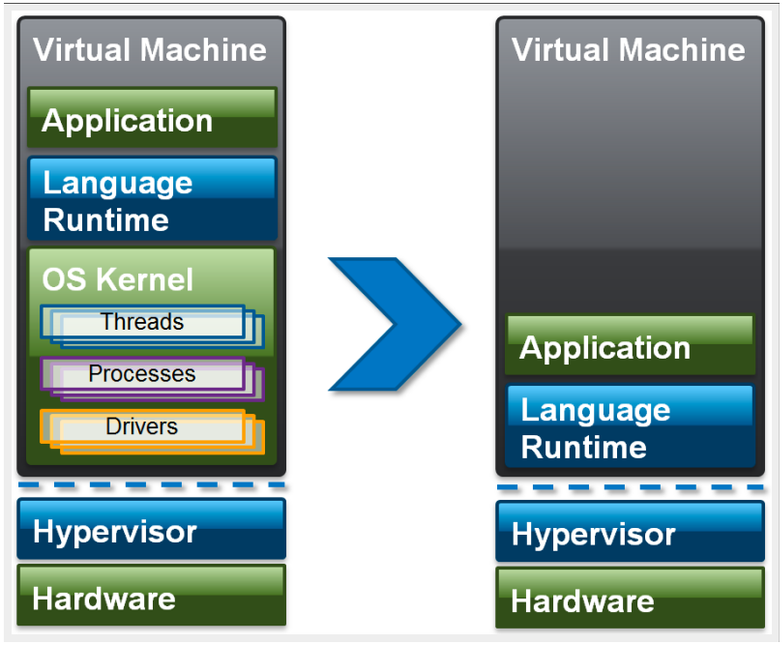
anykernel

- introduced by a NetBSD hacker (rump kernel)
- Definition:

> *We define an anykernel to be an organization of kernel code which allows the kernel's **unmodified** drivers to be **run in various configurations** such as application libraries and microkernel style servers, and also as part of a monolithic kernel. -- Kantee 2012.*

- can form various kernel for various platforms
- userspace (POSIXy), bare-metal, qemu/kvm, Xen
  - **Unikernel ?**

single purpose operating system

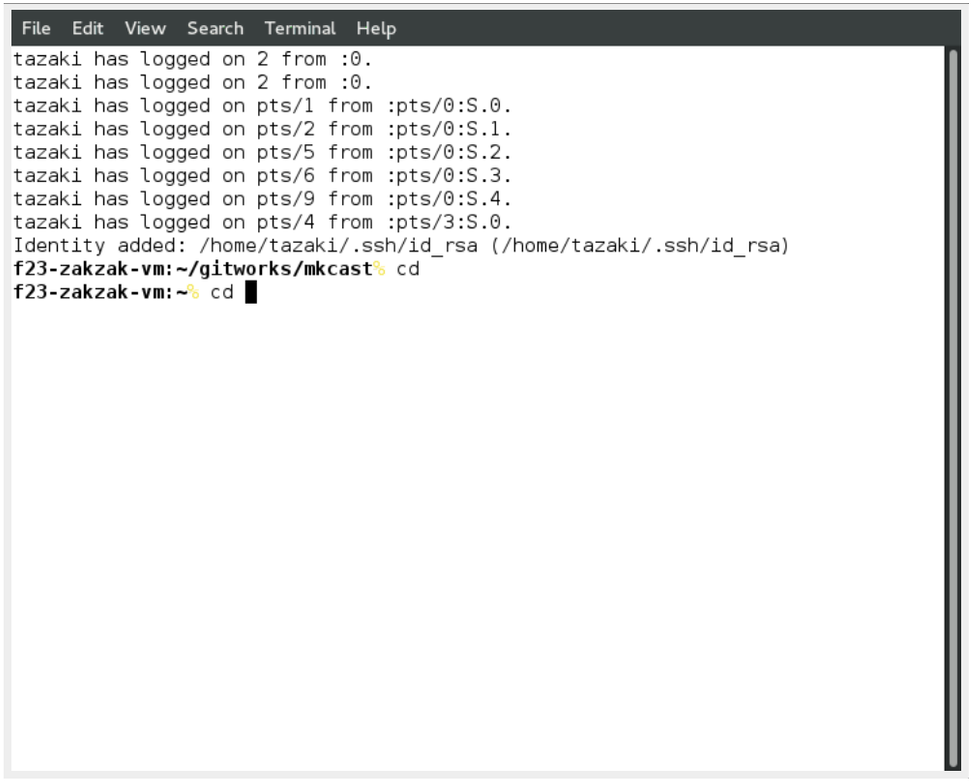


- http://www.linux.com/news/enterprise/cloud-computing/751156-are-cloud-operating-systems-the-next-big-thing-

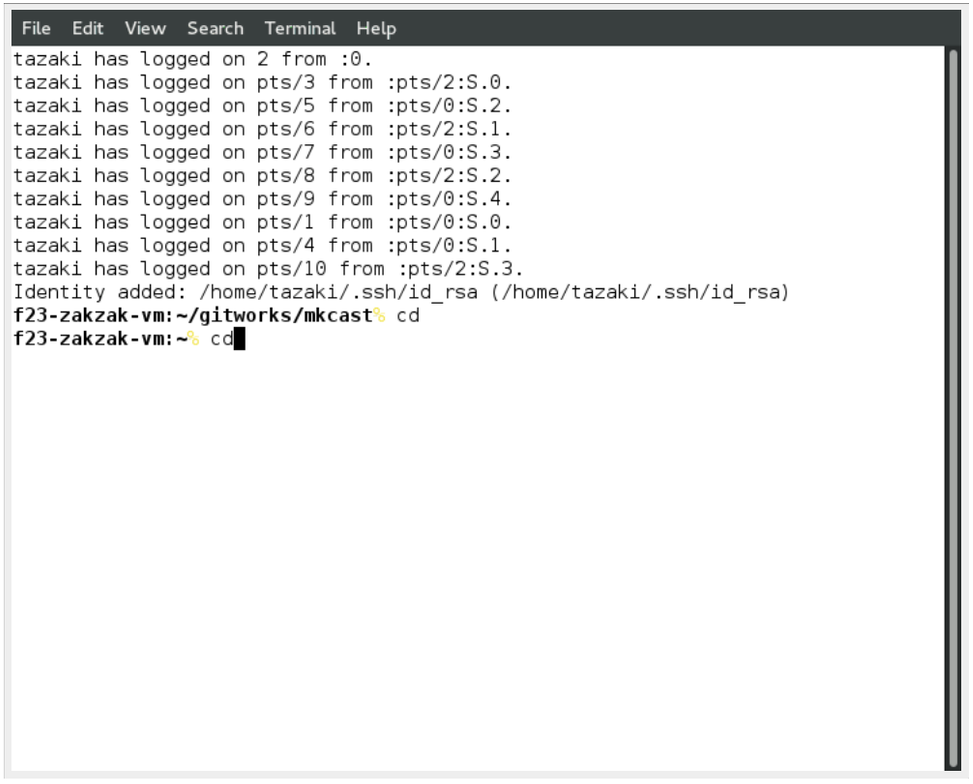
- Strip downed software stack
  - single purpose
  - resource efficient with speed
- boot within TCP 3-way handshake [1]

[1]: Madhavapeddy et al., Jitsu: Just-In-Time Summoning of Unikernels, USENIX NSDI 2015

demos with linux kernel library



Unikernel on Linux (ping6 command embedded kernel library)



Unikernel on qemu-arm (hello world)

what's different ?

- User Mode Linux
  - generate executable of Linux kernel in userspace
  - no shared library
- Containers
  - no foreign OS (shared kernel with host)
- nfsim
  - broader coverage of kernel code

recent news

- Linux kernel library (LKL) is coming
  - by Octavian Purdila (Intel)
  - since 2007, reborn in late 2015

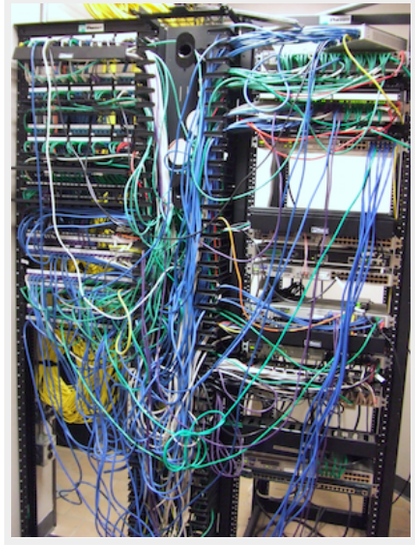
LibOS project is going to migrate to LKL project

- port NUSE code to LKL already
- DCE (ns-3 integration) not yet
- unikernel in progress

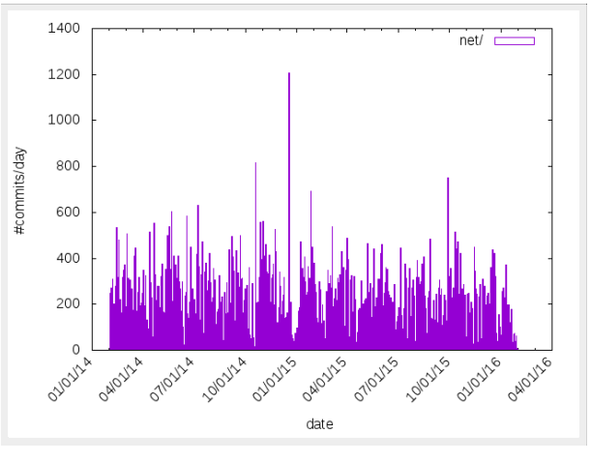
testing network stack

motivation

- testing networking code is hard
  - complex cabling
  - inefficiency with massive VM instances
- You may do
  - in your own large testbed
  - with your test programs



are we enough ?



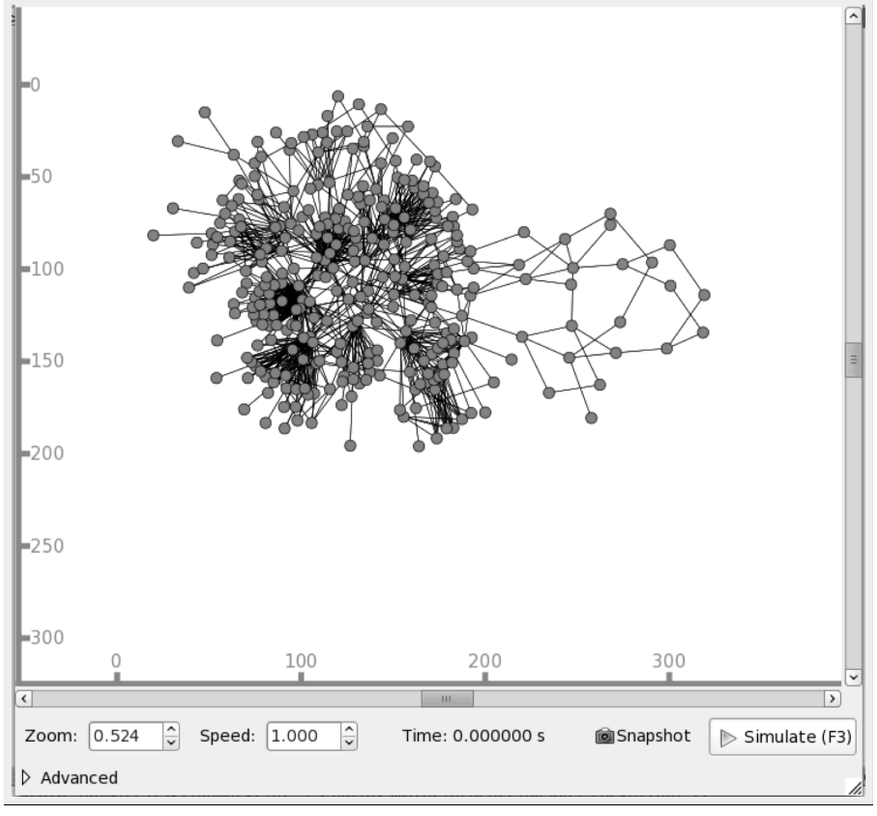
- the number of commit per day

- frequently changing codebase
  - many commits (30~40 commits/day)
  - out of 982K LoC (`cloc net/`)
  - may have increased num of regression bugs

your test

- easy to create in your laptop with VM (UML/Docker/Xen/KVM)
- ***only IF*** the test is enough to describe

your test (cont'd)



- huge resources to conduct a test
- not likely to **reproduce**
- tons of **configuration scripts**
- running on different machines/OSes
  - controling is troublesome
  - distributed debugger...

many terminal windows with gdb

other projects

- Test suites/projects
  - LTP (Linux test project, https://linux-test-project.github.io/)
  - kselftest (https://kselftest.wiki.kernel.org/)
  - autotest (http://autotest.github.io/)
  - ktest (in tree, http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/tools/test id=HEAD)
  - kernelci (https://kernelci.org/)
  - NetDEF CI (quagga)
- those are great but **networking is always hard**
  - controlling remote hosts is (sometimes) painful
  - combination of userspace programs are unlimited
  - timing is not deterministic, across distributed networks

why LibOS ?

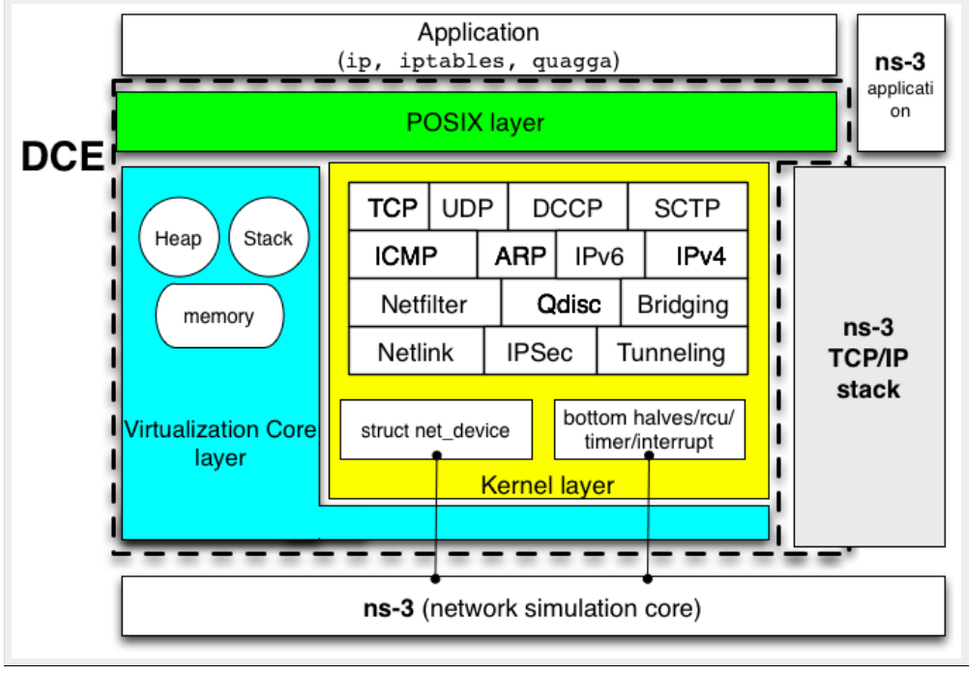
- **single process model** with multiple nodes
  - ease of debug/test/development
- **deterministic behavior** (by ns-3 network simulator)
- rich network configuration by **ns-3 network simulator**
- ease of testing by automation (on public CI server)

public CI server (circleci.com)

- test per commit (push)
- test *before* commit
- easily detect regressions

architecture

1. Virtualization Core Layer
   - deterministic clock of simulator
   - stack/heap management
   - isolation via **dlmopen(3)**
   - **single process model**
2. Kernel layer
   - reimplementation of API
   - glue/stub codes for kernel code
   - use as-is
3. POSIX glue layer
   - reimplementation of POSIX API
   - hijack host system calls



How ?

- a single scenario script (C++, sorry) to describe all
  - application, network stack (kernel as a lib), traffic, link, topology, randomness, timing, etc

1. Recompile your code
   - Userspace as Position Independent Executable (PIE)
   - Kernel space code as shared library (libsim-linux.so)
2. Run with ns-3
   - Load the executables (binary, library) in an isolated environment among nodes
   - synchronize simulation clocks with apps/kernels clock

features

- app supports
  - routing protocols (Quagga)
  - configuration utilities (iproute2)
  - traffic generator (iperf/ping/ping6)
  - others (bind9, unbound, dig)
- protocol supports
  - IPv4/ARP/IPv6/ND
  - TCP/UDP/DCCP/SCTP/(mptcp)
  - L2TP/GRE/IP6IP6/FOU

what's *not* useful

- performance study of the computation
  - deterministic clock assumes **unlimited** computation/storage resources
  - e.g., you can define 100Tbps link without any packet loss

test suite list

- verify results
  - socket (raw{6},tcp{6},udp{6},dccp{6},sctp{6})
  - encapsulation (lt2p,ip6ip6,ip6gre,fou)
  - quagga (rip,ripng,ospfv{2,3},bgp4,radvd)
  - mptcp
  - netlink
  - mip6 (cmip6,nemo)
- simple execution
  - iperf
  - thttpd
  - mptcp+iperf handoff
  - tcp cc algo. comparison
  - ccnd

bugs detected by DCE (so far)

- having nightly tested with the latest net-next (since Apr. 2013~=4yrs)
- [net-next,v2] ipv6: Do not iterate over all interfaces when finding source address on specific interface. (v4.2-rc0, **during VRF**)
  - detected by: http://ns-3-dce.cloud.wide.ad.jp/jenkins/job/daily-net-next-sim/958/testReport/
- [v3] ipv6: Fix protocol resubmission (v4.1-rc7, **expanded from v4 stack**)
  - detected by: http://ns-3-dce.cloud.wide.ad.jp/jenkins/job/umip-net-next/716/
- [net-next] ipv6: Check RTF_LOCAL on rt->rt6i_flags instead of rt->dst.flags (**v4.1-rc1, during v6 improvement**)
  - detected by: http://ns-3-dce.cloud.wide.ad.jp/jenkins/job/daily-net-next-sim/878/
- [net-next] xfrm6: Fix a offset value for network header in _decode_session6 (v3.19-rc7?, **regression only in mip6**)

Use Case

network simulator in a nutshell

- (mainly research purpose)
- flexible parameter configurations
- usually in a single process
  - can be extended distributed/parallel processes for speedup
- usually with abstracted protocol implementation
  - but **no abstraction** this time (thanks to LibOS)
- always produce same results (**deterministic**)
  - can inject pseudo-randomness
  - not realistic sometimes
  - but useful for the test (always reproducible)

workflow

1. (installation of DCE)

```
make testbin -C tools/testing/libos
```

2. ~~develop a model (of interests)~~
   - (you already have: the Linux network stack)
3. write a simulation scenario
   - write a network topology
   - parameters configuration (randomization seed, link, traffic, applications)
4. test it
   - one-shot (locally)
   - nightly, per-commit, per-push, etc

simulation scenario

```
int main(int argc, char **argv)
{
  // create nodes
  NodeContainer nodes;
  nodes.Create (100);

  // configure DCE with Linux network stack
  DceManagerHelper dce;
  dce.SetNetworkStack ("ns3::LinuxSocketFdFactory",
                       "Library", StringValue ("libsim-linux-4.4.0.so"));
  dce.Install (nodes);

  // run an executable at 1.0 second on node 0
  DceApplicationHelper process;
  ApplicationContainer apps;
  process.SetBinary ("your-great-server");
  apps = process.Install (nodes.Get (0));
  apps.Start (Seconds (1.0));

  Simulator.Stop (Seconds(1000.0))
  Simulator.Run ()
}
```

API (of DCE helpers)

- userspace app
  - `ns3::DceApplicationHelper class`
- kernel configuration
  - sysctl with `LinuxStackHelper::SysctlSet()` method
- printk/log
  - generated into `files-X` directory (where X stands for the node number)
  - syslog/stdout/stderr tracked per process (files-X/var/log/{PID}/)
- an instant command (`ip`)
  - `LinuxStackHelper::RunIp()`
- **manual**
  - https://www.nsnam.org/docs/dce/manual/html/index.html

test it !

- use `waf` for a build the script

```
cd tools/testing/libos/buildtop/source/ns-3-dce/
./waf
```

- run the script with `test.py` to generate XUnit test results

```
./test.py -s exapmle -r
```

- run the script with valgrind

```
./test.py -s exapmle -g
```

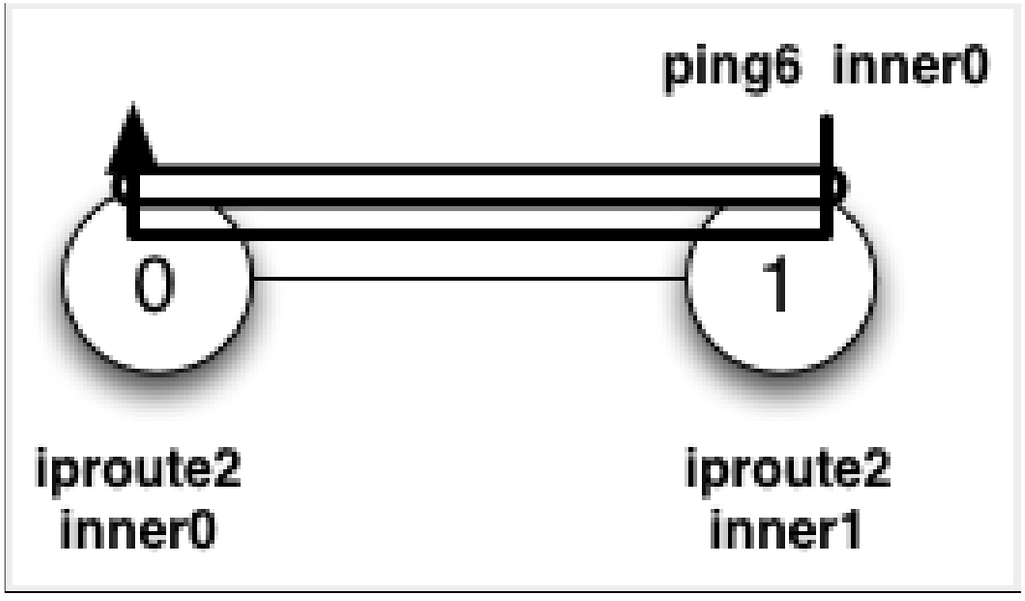
- a wrapper in Makefile

```
make test ARCH=lib ADD_PARAM=" -s example"
```

(*the directories may be changed during upstream (etc)*, **sorry 'bout that**)

case study: encapsulation test

ns-3-dce/test/addons/dce-linux-ip6-test.cc



- unit tests for encapsulation protocols
  - ip6gre, ip6-in-ip6, l2tp, fou
  - with `iproute2`, `ping6`, libsim-linux.so (libos)
- full script
  - https://github.com/direct-code-execution/ns-3-dce/blob/master/test/addons/dce-linux-ip6-test.cc

encap protocols tests

1) tunnel configurations

```
LinuxStackHelper::RunIp (nodes.Get (0), Seconds (0.5),
                         "-6 tunnel add tun1 remote 2001:db8:0:1::2 "
                         "local 2001:db8:0:1::1 dev sim0");
LinuxStackHelper::RunIp (nodes.Get (1), Seconds (0.5),
                         "-6 tunnel add tun1 remote 2001:db8:0:1::1 "
                         "local 2001:db8:0:1::2 dev sim0");
```

2) set up ping6 command to generate probe packet

```
dce.SetBinary ("ping6");
dce.AddArgument ("2001:db8:0:5::1");
apps = dce.Install (nodes.Get (1));
apps.Start (Seconds (10.0));
```

3) verify if the encap/decap work fine or not

```
if (found && icmp6hdr.GetType () == Icmpv6Header::ICMPV6_ECHO_REPLY) {
   m_pingStatus = true;
}
```

That's it. Test Test Test !

XUnit test result generation

- `make test ARCH=lib ADD_PARAM=" -s linux-ip6-test -r"` gives you a test result retained

```
% head testpy-output/2016-02-08-09-49-32-CUT/dce-linux-ip6.xml

<Test>
<Name>dce-linux-ip6</Name>
<Result>PASS</Result>
<Time real="3.050" user="2.030" system="0.770"/>
<Test>

<Name>Check that process &#39;plain&#39; completes correctly.</Name>

<Result>PASS</Result>

<Time real="0.800" user="0.370" system="0.310"/>
</Test>
<Test>

<Name>Check that process &#39;ip6gre&#39; completes correctly.</Name>

<Result>PASS</Result>

<Time real="0.600" user="0.460" system="0.100"/>
</Test>
```

git bisect

you can now *bisect* a bug with a single program !

- prepare a bisect.sh

```
#!/bin/sh

git merge origin/nuse --no-commit
make clean ARCH=lib
make library ARCH=lib OPT=no

make test ARCH=lib ADD_PARAM=" -s dce-umip"

RET=$?
git reset --hard

exit $RET
```

- run it !

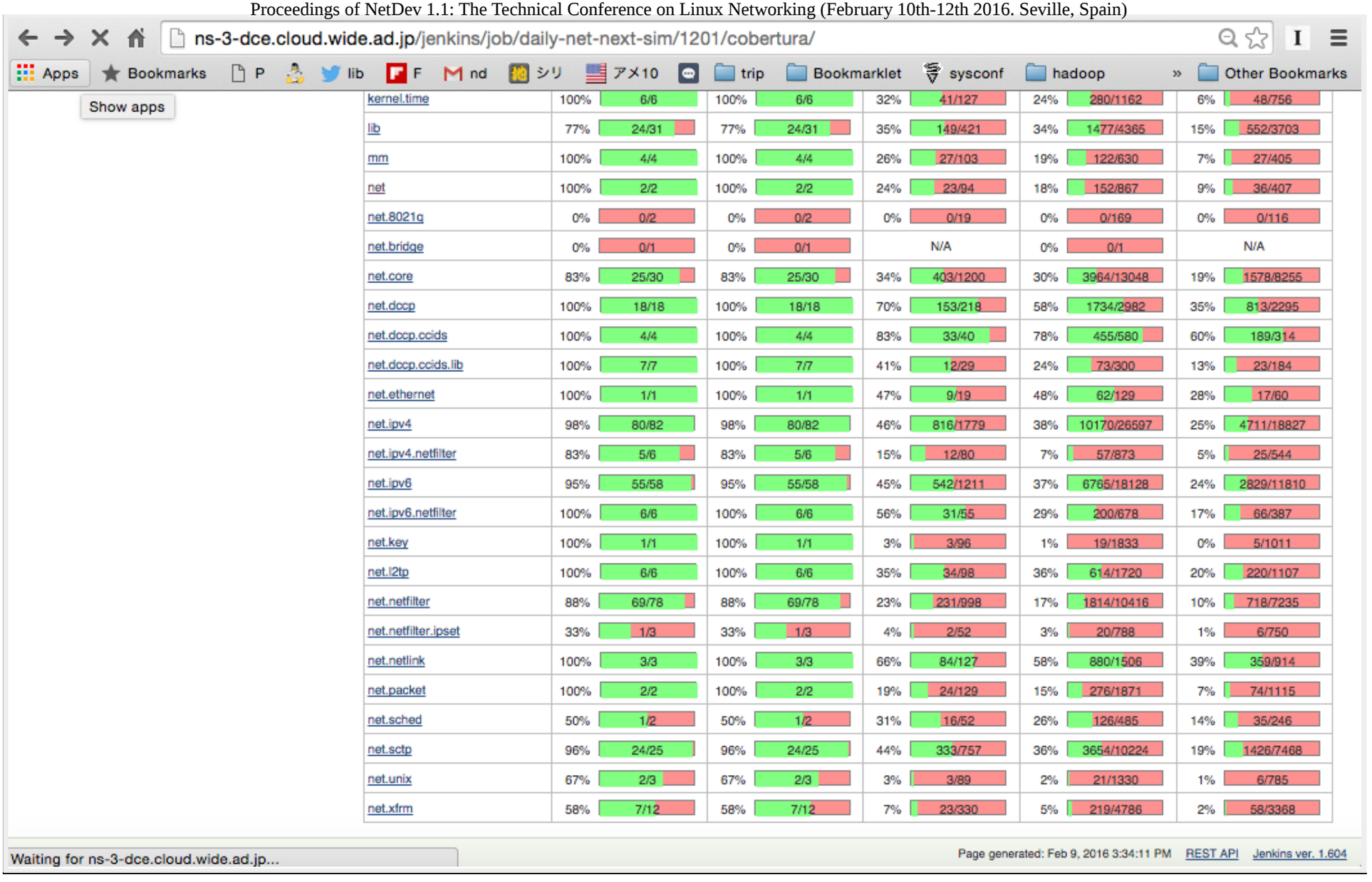
```
git bisect run ./bisect.sh
```

gcov (coverage measurement)

- coverage measurement across multiple nodes

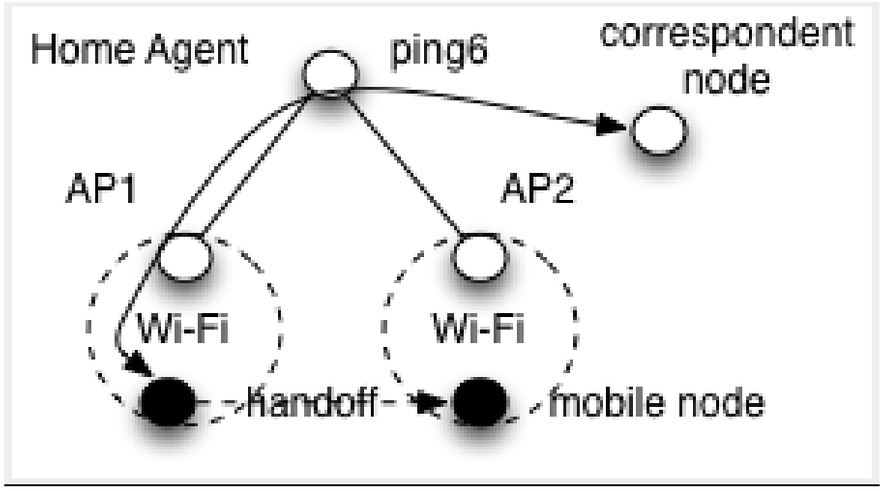
```
make library ARCH=lib COV=yes
make test ARCH=lib
```

(the **COV=yes** option does the job for you)



kernel.time	100%	6/6	100%	6/6	32%	41/127	24%	280/1162	6%	48/756
lib	77%	24/31	77%	24/31	35%	149/421	34%	1477/4365	15%	552/3703
mm	100%	4/4	100%	4/4	26%	27/103	19%	122/630	7%	27/405
net	100%	2/2	100%	2/2	24%	23/94	18%	152/867	9%	36/407
net.8021q	0%	0/2	0%	0/2	0%	0/19	0%	0/169	0%	0/116
net.bridge	0%	0/1	0%	0/1	N/A		0%	0/1	N/A	
net.core	83%	25/30	83%	25/30	34%	403/1200	30%	3964/13048	19%	1578/8255
net.dccp	100%	18/18	100%	18/18	70%	153/218	58%	1734/2982	35%	813/2295
net.dccp.ccids	100%	4/4	100%	4/4	83%	33/40	78%	455/580	60%	189/314
net.dccp.ccids.lib	100%	7/7	100%	7/7	41%	12/29	24%	73/300	13%	23/184
net.ethernet	100%	1/1	100%	1/1	47%	9/19	48%	62/129	28%	17/60
net.ipv4	98%	80/82	98%	80/82	46%	816/1779	38%	10170/26597	25%	4711/18827
net.ipv4.netfilter	83%	5/6	83%	5/6	15%	12/80	7%	57/873	5%	25/544
net.ipv6	95%	55/58	95%	55/58	45%	542/1211	37%	6765/18128	24%	2829/11810
net.ipv6.netfilter	100%	6/6	100%	6/6	56%	31/55	29%	200/678	17%	66/387
net.key	100%	1/1	100%	1/1	3%	3/96	1%	19/1833	0%	5/1011
net.l2tp	100%	6/6	100%	6/6	35%	34/98	36%	614/1720	20%	220/1107
net.netfilter	88%	69/78	88%	69/78	23%	231/998	17%	1814/10416	10%	718/7235
net.netfilter.ipset	33%	1/3	33%	1/3	4%	2/52	3%	20/788	1%	6/750
net.netlink	100%	3/3	100%	3/3	66%	84/127	58%	880/1506	39%	359/914
net.packet	100%	2/2	100%	2/2	19%	24/129	15%	276/1871	7%	74/1115
net.sched	50%	1/2	50%	1/2	31%	16/52	26%	126/485	14%	35/246
net.sctp	96%	24/25	96%	24/25	44%	333/757	36%	3654/10224	19%	1426/7468
net.unix	67%	2/3	67%	2/3	3%	3/89	2%	21/1330	1%	6/785
net.xfrm	58%	7/12	58%	7/12	7%	23/330	5%	219/4786	2%	58/3368

gdb (debugger)




```
(gdb) b mip6_mh_filter if dce_debug_nodeid()==0
Breakpoint 1 at 0x7ffff287c569: file net/ipv6/mip6.c, line 88.
<continue>
(gdb) bt 4
#0  mip6_mh_filter
    (sk=0x7ffff7f69e10, skb=0x7ffff7cde8b0)
    at net/ipv6/mip6.c:109
#1  0x00007ffff2831418 in ipv6_raw_deliver
    (skb=0x7ffff7cde8b0, nexthdr=135)
    at net/ipv6/raw.c:199
#2  0x00007ffff2831697 in raw6_local_deliver
    (skb=0x7ffff7cde8b0, nexthdr=135)
    at net/ipv6/raw.c:232
#3  0x00007ffff27e6068 in ip6_input_finish
    (skb=0x7ffff7cde8b0)
    at net/ipv6/ip6_input.c:197
```

- Inspect codes during experiments
  - among distributed nodes
  - in a single process
- perform a simulation to reproduce a bug
- see how badly handling a packets in Linux kernel

http://yans.pl.sophia.inria.fr/trac/DCE/wiki/GdbDce

valgrind

```
==5864== Memcheck, a memory error detector
==5864== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==5864== Using Valgrind-3.6.0.SVN and LibVEX; rerun with -h for copyright info
==5864== Command: ../build/bin/ns3test-dce-vdl --verbose
==5864==
==5864== Conditional jump or move depends on uninitialised value(s)
==5864==    at 0x7D5AE32: tcp_parse_options (tcp_input.c:3782)
==5864==    by 0x7D65DCB: tcp_check_req (tcp_minisocks.c:532)
==5864==    by 0x7D63B09: tcp_v4_hnd_req (tcp_ipv4.c:1496)
==5864==    by 0x7D63CB4: tcp_v4_do_rcv (tcp_ipv4.c:1576)
==5864==    by 0x7D6439C: tcp_v4_rcv (tcp_ipv4.c:1696)
==5864==    by 0x7D447CC: ip_local_deliver_finish (ip_input.c:226)
==5864==    by 0x7D442E4: ip_rcv_finish (dst.h:318)
==5864==    by 0x7D2313F: process_backlog (dev.c:3368)
==5864==    by 0x7D23455: net_rx_action (dev.c:3526)
==5864==    by 0x7CF2477: do_softirq (softirq.c:65)
==5864==    by 0x7CF2544: softirq_task_function (softirq.c:21)
==5864==    by 0x4FA2BE1: ns3::TaskManager::Trampoline(void*) (task-manager.cc:261)
==5864==  Uninitialised value was created by a stack allocation
==5864==    at 0x7D65B30: tcp_check_req (tcp_minisocks.c:522)
==5864==
```

- Memory error detection
  - among distributed nodes
  - in a single process
- Use **Valgrind**

http://yans.pl.sophia.inria.fr/trac/DCE/wiki/Valgrind

Summary

- walk through review of testing framework with LibOS + DCE
- uniqueness of experiemnt with the library (LibOS)
  - multiple (host) instances in a single process
  - flexible network configurations
  - deterministic scheduler (i.e., bugs are always reproducible)

future directions

- merging to LKL (Linux Kernel Library)
  - part of LibOS has done
- continuous testing to net-next branch
  - I'm watching at you (don't get me wrong.. :))

resources

- Web
  - https://www.nsnam.org/overview/projects/direct-code-execution/ (DCE specific)
  - http://libos-nuse.github.io/ (LibOS in general)
- Github
  - https://github.com/libos-nuse/net-next-nuse
- LKL (Linux Kernel Library)
  - https://github.com/lkl/linux