Probing the Latencies of Software Timestamping

Benjamin Villain\textsuperscript{1}, \textbf{Matthew Davis}\textsuperscript{2}, \textbf{Julien Ridoux}\textsuperscript{2}, Darryl Veitch\textsuperscript{2} Nicolas Normand\textsuperscript{1}

\textsuperscript{1} Université de Nantes, IRCCyN UMR CNRS 6597, Nantes, France
benjamin.villain@gmail.com, nicolas.normand@univ-nantes.fr

\textsuperscript{2} Electrical & Electronic Engineering Dpt, The University of Melbourne, Australia
{matt, julien}@synclab.org, dveitch@unimelb.edu.au

This work was supported in part by a research grant from Symmetricom Inc.
Software clocks have some obvious advantages

- Cheap way for desktops or servers to sync to a master clock
- Easy to deploy (no antenna, no separate cabling)
- Seamless integration with user applications (no need to rewrite code, access a specific device . . . )

... but unavoidable issues (network and in-host)

- Delay asymmetry (hard problem)
  → median clock error as high as 100 μs
- Delay variability (easier problem)
  → clock error standard deviation in 1-10 μs range
One Cause of Delay: In-Host Timestamping

```c
int some_function()
{
    int foo;
    /* Some processing */
    statement_1;
    /* Need to know what time it is */
    clock_gettime(CLOCK_REALTIME);
    /* Some more processing */
    statement_2;
    return (0);
}
```

- System Call Enter
- Kernel routine calls
- Hardwake Counter access
  - TSC, HPET, ACPI
  - 1588 clock device
- System Call Exit
- Scheduler impact?

How much time does this take?
Impact on Software Clocks

Software clocks are userland applications

- Rely on network packet departure / arrival timestamps
- Timestamping latency affects final clock error
- The smaller the network delay, the more important the in-host timestamping latency

Solutions

- Robust algorithm, tight filtering
- Measure / estimate in-host delays
  - Is it possible?
  - Feasible in production environment?
Objectives

- Proof of concept to benchmark latency of a production system
- Characterise in-host delays within the network stack
- Provide results relevant to software clocks and their timestamping strategies
  - ntpd
  - ptpd
  - RADclock
Probing the OS

Software Probing Tools

- Tools have emerged over the past 5-7 years.
- SystemTap: Linux
- DTrace: Solaris, Mac OSX, FreeBSD

DTrace on FreeBSD 9.0

- Ease of use, lightweight, targets production environments
- DTrace probes available in kernel and user context
- Scripts specify which probes fire (e.g. syscall entry)
  - Probes are investigation points to inspect function properties
  - Simple actions: increment counters, record clock time
  - Data collected can be aggregated
DTrace Clock & Probe Effect

DTrace Clock

- Based on fast access TSC (Time Stamp Counter)
- Effectively a scaled TSC counter that does not track drift
- Clock error may be of the order of 50 PPM
  - Very bad absolute clock
  - Ideal difference clock (error below 100 ns over a 1 ms interval)

DTrace Probe Effect

- Probe effect of the order of a couple of µs per probe
Experimental Setup

- UDP client and server exchange small UDP test packets
- DAG card hardware timestamp probes (control data)
- DTrace probes deployed within the network stack
- DTrace clock is read when UDP packet is processed
In-Host Timestamping locations (1/2)

DTrace Probes attached to timestamping functions entry points

- Userland (ntpd)
  - gettimeofday(), clock_gettime()

- Socket timestamping SO_TIMESTAMP (ptpd)
  - Incoming packets timestamped in FreeBSD socket layer
  - Copy of outgoing packet sent on loopback interface in RX path

- Berkeley Packet Filter (RADclock)
  - Driver dispatches a copy of packet to BPF subsystem

- NIC Driver (Hardware timestamps)
  - Timestamping function in Intel i350 driver
In-Host Timestamping locations (2/2)

- All timestamps created with the same clock (DTrace)
- Measure delays using DTrace difference clock
- Driver timestamp used as reference
Experiment: Stress Testing

Spice things up with stress tests

- Continuous measurement over normal and stress periods
- Alternate 2 hour periods

Stress scenarios (using stress2 test suite)

- **NS**: Non-Stress period
- **IO**: writes and reads of files of random size to generate high disk activity, plus memory swapping
- **SC**: System Calls resulting in many user/kernel contexts switches
- **IP**: transmission of UDP packets on loopback interface to stress the network stack
One-Way Delays: Outgoing Path
Experiment Setup: Where to Probe

- Outgoing path has both larger delays, and higher variability.
  - Note no device polling here
- `bpf` performs much better than `so_timestamp` and userland
- Outgoing `so_timestamp`, shows negative OWD in all scenarios
  - under IP minimum delay is \(-356\,\mu s > RTT!!\)
  - `so_timestamp` breaks causality (packets transmitted before their timestamp is made) $\rightarrow$ cannot measure RTTs
- Userland shows larger delays and variability
Evaluation of In-Host Asymmetry

Paths asymmetry → median clock error
- With IEEE 1588, LAN and hardware master clock:
  - RTT gets smaller
  - relative contribution of in-host path to asymmetry increases

Calibrate and compensate for asymmetry
- bpf has small bound and consistent asym. under stress
- so_timestamp breaks causality
- userland has large bound and may suffer from large variations
Conclusion

Timestamping location helps fighting delay variability

- Avoid `so_timestamp` no reliable bounding of RTT possible
- `bpf` is best choice for performance
- couple it with good filtering in sync. algo does a good job

Compensating for in-host asymmetry

- `bpf` has lower bound $\rightarrow$ smaller asymmetry
- `bpf` has lower bound $\rightarrow$ better estimate of asymmetry

Variation of asymmetry under load remains a barrier to achieving $1\mu s$ precision with software clocks.

http://www.synclab.org/radclock