Report: communcation jitter analysis, periodic peaks

Davide Rovelli, Michele Dalle Rive

04/2024

1 Objective

We analyse the latency disitribution in the communication between 2 bare-metal nodes in a datacenter. The objective of our experiments is to make packet processing latency of Linux-based hosts with modern NICs as *deterministic* as possible. This consists in reducing communication jitter to a minimum, where $jitter = \max(latency) - \min(latency)$.

Contrarily to usual low tail-latency experiments/systems which aim to reduce the common-case latency up to some percentile (e.g.$99.5\%, 99.9\%$) we want to minimise the 100% latency, even at the cost of (non-substantially) increasing the common-case.

2 Methodology

2.1 Benchmark

We run ping-pong tests across 2 hosts **A** and **B** by sending packets at a fixed sending rate. Each packet carries a packet ID and 4 timestamps:

- ts_1: timestamp just before sending the PING message in host **A**
- ts_2: timestamp as soon as the process in host **B** receives the PING packet.
- ts_3: timestamp just before sending the PONG message in host **B**
- ts_4: timestamp as soon as the process in host **A** receives the PONG packet.

We then calculate the difference between timestamps of successive packets to get the variations in latency. For example, $ts_1(33) - ts_1(32)$ where 33 and 32 are the packet IDs, represents the time interval between the PING send timestamp of packet 32 and the PING send timestamp of packet 33. If there's no jitter, it should be equal to the sending rate. The difference between $ts_1(33) - ts_1(32)$ and the sending rate represents the relative jitter for that pair.

2.2 Hardware setup

2x CloudLab xl170 nodes. See cloudlab.us for more detailed specs.

- CPU: Intel Xeon E5-2640 v4 at 2.40GHz, 10 cores, 64GB RAM
- OS/kernel: Ubuntu 22.04.4 LTS / 6.6.19-060619-generic x86_64
- NIC/driver: Mellanox Connect-X 4 / mlx5

Nodes are connected over Ethernet, the physical connection is likely to span multiple switches. No concurrent traffic should be present as CloudLab guaranteed experiment reproducibility.

2.3 Software setup

We use different *kernel bypass* methods in order to minimize the well-known overhead introduced by the classical network stack in Linux machines. We test the following:

- RoCE, specifically double-sided SEND/RECV RDMA with the Unreliable Datagram (UD) transport type
- eBPF XDP in two different modalities:
  - XDP-offload: packet is handled by an eBPF XDP hook and executed as early as possible in the network stack
  - AF_XDP: XDP socket type which handles TX and RX buffer management to the application
- Standard network stack over UDP

Packet size: *1024 bytes*

3 Analysis of periodic outliers

Here we show some of the results which include interestingly periodic latency outliers, which we refer to as "peaks". We observe two types of periodic peaks of which we have not yet identified the root cause. We refer to them as ts_1**-offset** and ts_2**-arrow**.

Figure 1 shows a typical benchmark containing a thick baseline around the sending rate ($996\mu s$), which means that most packets are subject to a minimal jitter apart from the the outlier types that we analyze below.

ts_1**-offset** The top-left plot in Figure 1, representing the difference between 2 successive send events of the PING packet on host A, shows ts_1**-offset**: a small number of outliers delayed from baseline by a similar offset in the y-axis ($\approx 150\mu s$). Successive outliers evenly spaced, meaning there is a fixed number of packet (i.e. time interval) between them.

ts_2**-arrow** The top-right plot in Figure 1, representing the difference between 2 successive receive events of the PING packet on host B, shows ts_2**-arrow**: a series of outliers with decreasing amplitude. Every outlier on top of the baseline (delayed packet) is immediately followed by a symmetric point below the baseline which shows that the successive packet was not delayed, hence the difference between its timestamp and the one of the previous delayed packet is less than then sending rate. Successive outliers forming the arrow are evenly spaced, meaning there is a fixed number of packet (i.e. time interval) between them.

Bottom left and bottom right plot show no periodic outliers, meaning that the PONG send and receive events do not seem to be subject to periodic peaks in our experiments. ts_1**-offset** and ts_2**-arrow** have the following characteristics:

- Independent from the software technology used, they manifiest with all bypass techniques listed in subsection 2.3 as well as the normal network stack.
- Throughput dependent: changing the sending rate changes some characteristics as we cover in subsection 3.1

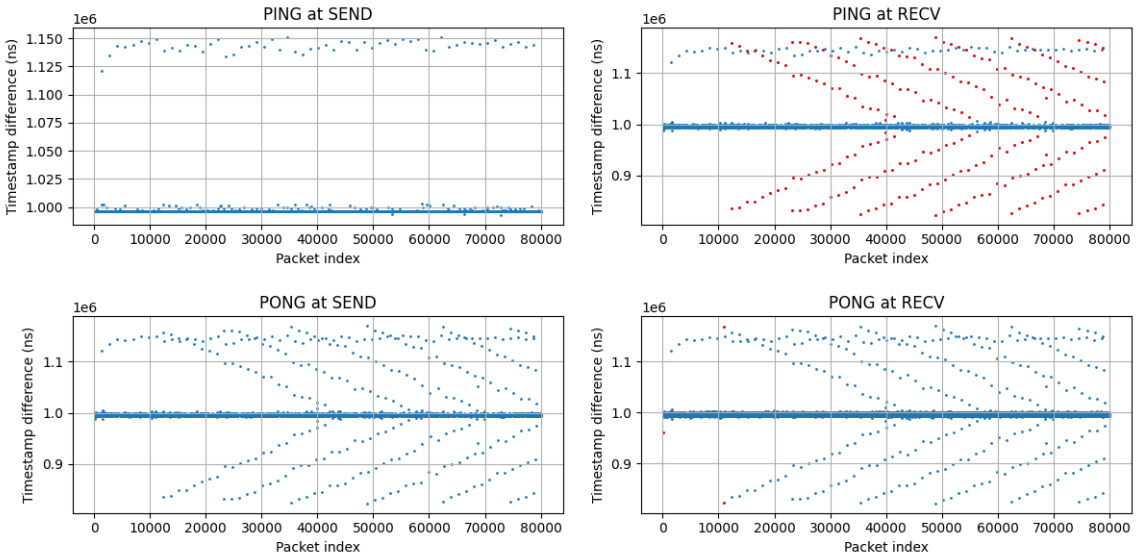


Figure 1: Pingpong benchmark plot - sending interval 996μs. Each dot corresponds to a packet ID (x-axis) and the difference between its timestamp and the timestamp of the packet with the previous ID (y-axis). The sending interval is fixed and packets are sent in order. See subsection 2.1 for a more detailed explanation of the benchmark

- Partially deterministic: 2 runs with the same settings often leads to the same result but not always, especially ts_2**-arrow** seems to be the most variable.
- The total amount of outliers of both types is less than the 0.01% and decreases even further when isolation techniques are applied.

3.1 Effect of sending rate

The sending rate of the PING packet seems to affect both ts_1**-offset** and ts_2**-arrow**.

Figure 2 shows 5 different tests in which we can see the density, offset and shape of the two periodic outliers types change. Figure 2a, Figure 2b, Figure 2c have sending periods of respectively $999\mu s, 1000\mu s$ and $1001\mu s$, which result in quite noticeable differences in the latency plots. We can observe that ts_1**-offset** (left plot) changes in terms of density and offset, respectively of $\approx 90\mu s$, $\approx 160\mu s$ and $\approx 50\mu s$. ts_2**-arrow** also varies in angle, density and periodicity. The 2 bottom plots Figure 2d and Figure 2e show results with sending periods of respectively $200\mu s$ and $50\mu s$. Here ts_2**-arrow** is replaced by a "line" which is similar to ts_1**-offset**. This characteristic was often observed in sending intervals smaller than $1000\mu s$. One observation is that the density of ts_1**-offset** seem to correspond to the one of ts_2**-arrow** in several cases, as for Figure 2a, Figure 2c, Figure 2d, Figure 2e in the picture. However, there is no clear cause-effect relationship as in several other tests ts_2**-arrow** occurs before ts_1**-offset**.

Note on loose reproducibility The same sending interval leads *roughly* to the same result in different tests. However, changing servers and other settings sometimes leads to minor changes in

the properties of ts_1-**offset** and ts_2-**arrow**. It's important to note that results are similar and not identical, e.g. the first arrow in ts_2-**arrow** starts at an arbitrary point, it'same discontinuous or does not appear in the test interval (but usually always appears in longer runs).

3.2 Possible causes and observations

Below we draft a list of possible causes and observations in Q&A format in order to outline what we have already tried to find the root cause of the outliers.

Are the peaks related to a particular bypass or software method? No, both peak types show with all RDMA, XDP and normal network stack. For instance, Figure 2b was tested with all implementations and leads to the same pattern.

OS jitter interrupts, context switching etc. ? We ran experiments by reserving a number of cores exclusively for our programs, shielding such cores from interrupts, using the `isolcpus`, `nohz` kernel boot settings; we disable C-states, setting the CPU power to the max. The full settings can be found in the project repository. These settings improve the overall latency and reduce the number of random outliers but do not eliminate the periodic outlier types.

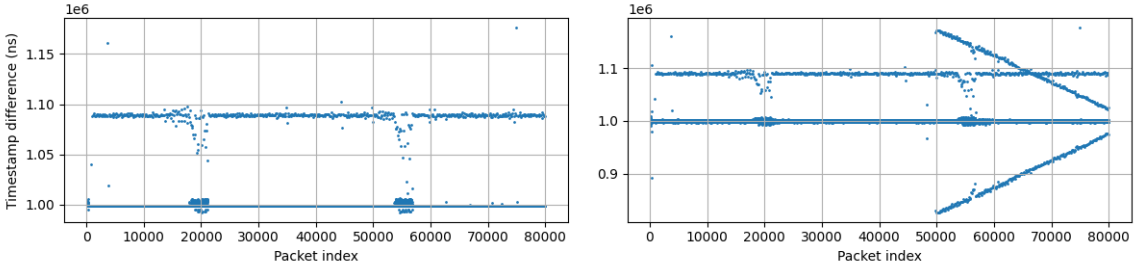
Interrupts coalescence? No, the tests were performed with the interrupt coalescence settings of both NICs disabled.

Code-related issue? Very unlikely. We test with 3 different programs. The send loop is just a busy loop pinned to a dedicated uninterrupted core. The simplest logic on the PONG host is a XDP hosts that takes a timestamp as soon as a packet arrives, inserts in the packet and sends it back. This executes as early as possible in the network stack, right after the call to the driver. This means that the ts_2-**arrow** effect occurs before we can take any action.

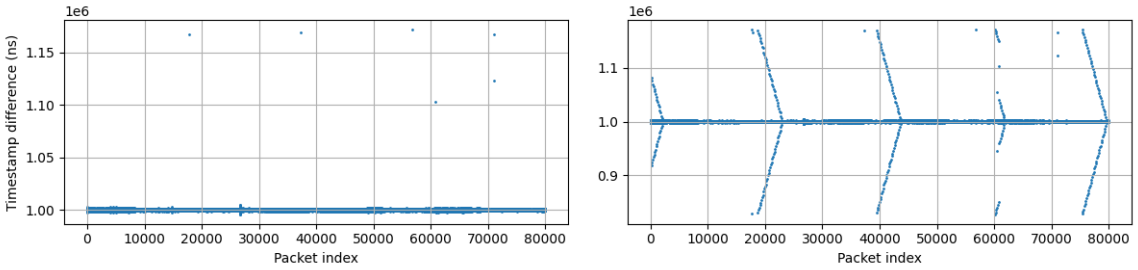
Caused by the network? ts_1-**offset** occurs before the packet is actually sent out, so the network can be excluded for it. It could play a role in ts_2-**arrow**, however CloudLab (our test cluster) guarantees experiment reproducibility – it is unlikely that concurrent traffic is present. Side effect of switching?

Why do the outliers show in the PING transfer and never in PONG? If ts_2-**arrow** is related to packet reception / processing, we should observe it also at reception of the PONG packet, but this is not the case. One explanation could be that ts_2-**arrow** is somehow caused by ts_1-**offset**, hence related to the sending loop which happens only at the beginning. However, no clear cause-effect has been identified up to now.

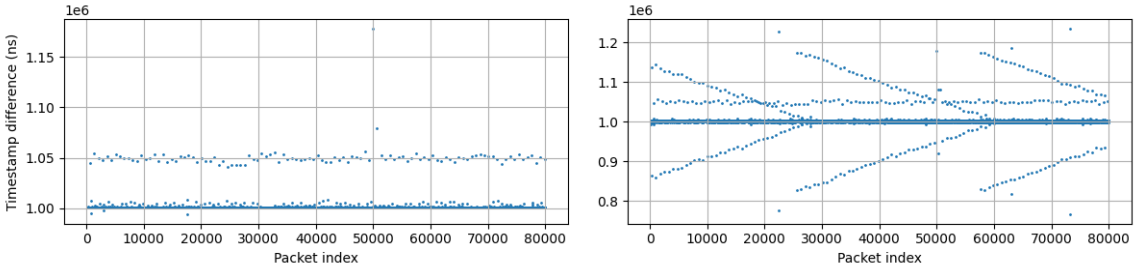
Unmaskable timing interrupts? Some kernel interrupts are unmaskable and they could explain outlier periodicity. However, they would not explain why the period changes according to the sending interval.



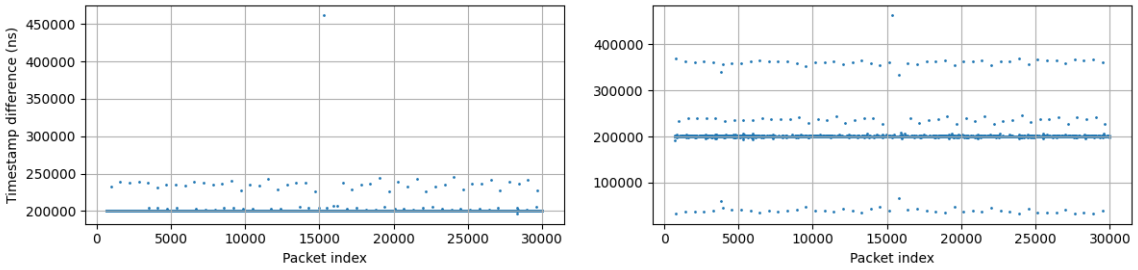
(a) Sending interval: 999μs, method: AF_XDP.



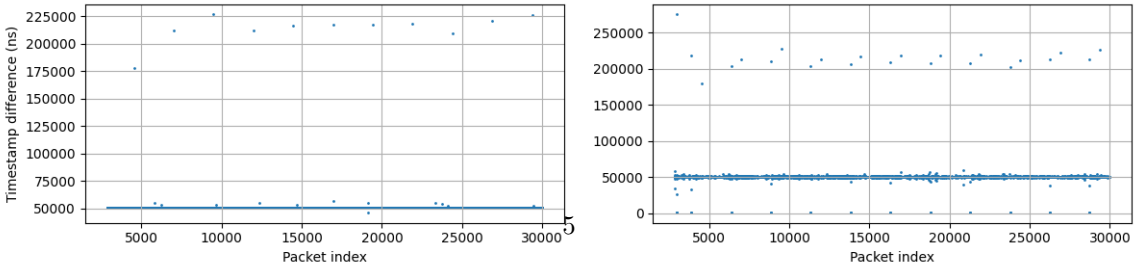
(b) Sending interval: 1000μs, method: XDP offload. CPU isolation, interrupt shielding and no interrupt coalescence are applied in this benchmark



(c) Sending interval: 1001μs, method: RDMA.



(d) Sending interval: 200μs, method: AF_XDP.



(e) Sending interval: 50μs, method: AF_XDP.

Figure 2: Different examples of periodic outliers types ts_1**-offset** and ts_2**-arrow** observed with varying sending intervals and different implementation methods. All tests were run with CPU cores at max frequency (C0 state)

Memory effects? Paging, TLB / cache misses can introduce delays that can appear as periodic if we repeat the same action in a loop and use same size data structures as we do. However, it is harder to justify the relationship between the sending interval and the outliers. Take for example Figure 2b and Figure 2c: a change of $1\mu s$ in the sending period causes $> 10\times$ more outliers - how would it relate to memory effects?