R2P2 and UDP support in Lancet

Morales Gonzalez Mikael
EFPL, Switzerland
June 7, 2019

Abstract

LANCET is a self-correcting tool designed to measure the open-loop tail latency of µs-scale datacenter applications with high fan-in connection patterns [2]. LANCET is self-correcting as it relies on online statistical tests to determine situations in which tail latency cannot be accurately measured from a statistical perspective, including situations where the workload configuration, the client infrastructure, or the application itself does not allow it.

We present the implementation of multiple agents in LANCET. These agents support two different transport protocols: UDP and R2P2, a UDP-based transport protocol specifically designed for RPCs inside a datacenter [3]. These implementations add full support for two different transport protocols, in addition with TCP, which was already supported in the original implementation of LANCET.

Each protocol has a specific agent supporting NIC-based hardware timestamping, which when available, is used by LANCET to measure RPCs end-to-end latency eliminating the client bias. Otherwise, it uses an asymmetric setup with a latency-agent that leverages busy-polling system calls.

Our experiment shows LANCET accurately reports the latency distribution with a service time of $\bar{S} = 10\mu s$.

When leveraging NIC-based hardware timestamping, eliminating the client bias, LANCET reports a latency that is $\sim$10µs lower compared to software-based timestamping when using both UDP and R2P2 as transport protocol.

1 Introduction

Web-scale datacenters rely on the decomposition of extensive queries into smaller subqueries that are processed individually in thousands of interconnected servers. Today's technology allows µs-scale interactions between these interconnected servers that internally communicate using RPCs.

These interconnected servers most commonly layer RPCs on top of TCP, either trough RPC framework (*e.g.,* Thrift) or through application-specific protocols (*e.g.,* Memcached). Given the byte-oriented nature of TCP and the message oriented nature of RPCs, this leads to a mismatch. This mismatch has made RPC load distribution challenging. For this reason, R2P2, a UDP-based transport protocol specifically designed for RPCs inside a distributed architecture, was implemented [3].

The number of components involved in a single query has significantly increased and the extensive use of high fan-in, high fan-out patterns with the emergence of µs-scale interactions requires to focus on tail-latency.

Tail latency is harder to measure compared to others metrics such as throughput. Indeed, tail-latency depends on various number of factors beyond the workload itself. These factors, for instance, include, the overheads in the tool being used to measure the tail-latency and the experiment methodology.

LANCET is a self-correcting latency measurement tool designed to measure, in a statistically sound manner, the end-to-end tail latency of remote procedure calls in a testing environment [2]. LANCET is self-correcting as it relies on on-line statistical tests to determine situations in which tail latency cannot be accurately measured.

LANCET relies on state-of-the-art, hardware-based measurement techniques that combine NIC timestamping in hardware and userlevel matching of packets to RPCs allowing to measure end-to-end tail latency of remote procedure calls.

NIC-based hardware timestamping is performed using exclusively the standard Linux API when the NIC being used is compatible with hardware timestamping [6]. Three different transport protocols are currently supported in LANCET, (i) R2P2 (ii) UDP (iii) TCP, where each of them provides hardware-based timestamping to measure RPCs latencies. By doing so, we improve the measurement accuracy without having to kernel-bypass to achieve precise µs-scale client-side measurements. This report focuses on the implementation of UDP and R2P2 measure agents.

Our experiment with the different agents with synthetic service times demonstrates that LANCET (i) accurately reports the latency distribution with service time $\bar{S} = 10\mu s$; and (ii) reports a latency that is $\sim$10µs lower when the NIC times-

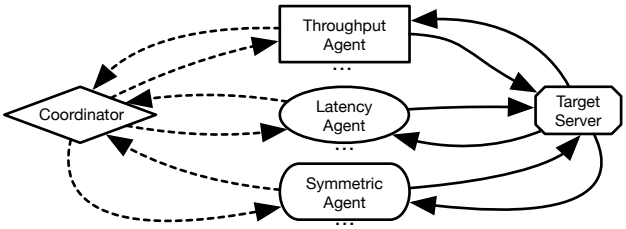

Figure 1: Lancet's architecture depicting a coordinator (C), throughput agents (TA), latency agents (LA), symmetric agents (SA), and the target server under test. The dashed arrows correspond to the LANCET API while the solid ones are application RPCs

tamping capability is available compared to software-based timestamping when using both UDP and R2P2 as transport protocol.

We discuss the necessary background (§2) and discuss the implementation of the agents (§3). We then analyze a latency experiment ran on R2P2 and UDP (§4), and conclude (§5).

2 Background

Lancet agents: LANCET is composed of a coordinator, measure agents and a target server. The coordinator will use the correct methodology to perform the measurements and communicate with the agents via the LANCET API. The agents generate the workload using application RPCs that are generated based on application-specific random distribution.

They are three types of agents available to match the capabilities of the available hardware, the measuring methodology and the target experiment granularity. The agents available are: latency agents (LA), throughput agents (TA) and symmetric agents (SA). The focus of this report is on the implementation of the three types of agents with two different transport protocols, namely R2P2 and UDP. Which complete the existing work made by implementing agents supporting TCP.

Figure 1 illustrates how LANCET components interact with each other.

LANCET supports both symmetrical and asymmetrical deployments. The asymmetrical model is used when the latencies are captured in software. This model splits the client machines between load-generating and latency-measuring, which reduces jitter but also collect fewer sample per time period. The symmetrical model is used when the NIC can timestamp all incoming and outgoing Ethernet frames and the Linux operating system exposes the information to userspace. In this model, all client machines generate load and measure latency. The hardware timestamping of packets is associated to end-to-end latency of RPCs.

UDP: UDP is a connectionless, message-oriented transport

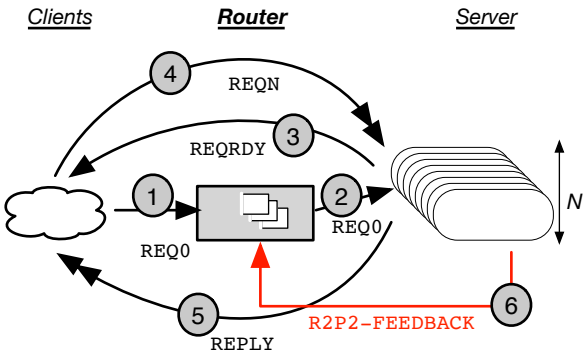

Figure 2: The R2P2 protocol for a request-reply exchange. Each message is carried within a UDP packet. Single arrows represent a single packet whereas double arrows represent a stream of datagrams.

protocol. It does not provide any guarantees on message delivery and each request/reply does not require any state across requests. UDP only provide a 'best effort' service, as messages can be lost or delivered out-of-order to the application. No services such as packet-ordering, flow control or congestion control is provided [7].

Each message in UDP is called a datagram. Given the datagram-oriented nature of the protocol, each request by an application produces exactly one UDP datagram, which causes one IP datagram to be sent. This is in contrast to a stream-oriented protocol such as TCP where the amount of bytes returned can have little relationship to what is sent in a single IP datagram. We can use this one-to-one matching property to easily compute the end-to-end latency. We discuss this in more details in §3.

R2P2: R2P2 (*Request-Response Pair Protocol*) is a UDP-based transport protocol specifically targeting latency-critical RPCs within a distributed infrastructure (*i.e.,* a datacenter). R2P2 is a connectionless transport protocol and unlike traditional methods that layer RPCs on top of TCP which is stream-oriented, R2P2 uses UDP which makes it an inherently request/reply-protocol that requires no state across requests.

Each request-response pair is identified by <*src_IP, src_port, req_id*> making each request completely independent of the actual server that will reply.

Figure 2 describes the interactions and the packets exchanged within a distributed infrastructure that uses a request router to load balance requests across the servers. It illustrates the general case of multi-packet request and response.

1. A `REQ0` message opens the RPC interaction, uniquely defined by the combination of source IP, UDP port, and an RPC sequence number. The datagram may contain the beginning of the RPC request itself.
2. The router identifies a suitable target server and directs the message to it. If there is no available server, requests can

temporarily queue up in the router.
3. If the RPC request exceeds the size of data available in the `REQ0` payload, then the server uses a `REQready` message to inform the client that it has been selected and that it will process the request.
4. Following (3), the client directly sends the remainder of the request as `REQn` messages.
5. The server replies directly to the client with a stream of `REPLY` messages.
6. The servers send `R2P2-FEEDBACK` messages to the router to signal idleness, availability, or health, depending on the load balancing policies.

Given that long RPCs request can be fragmented in `REQn` messages which internally will result in a stream of `REPLY` messages, special care had to be taken to correctly compute the end-to-end latency. Indeed, every outgoing and incoming packet is timestamp, but to correctly compute the end-to-end latency, we needed to make sure that the first request and last reply were used to compute the end-to-end latency. We developed more on how this constraint was ensured in §3.

3 Implementation

We implemented three different agents. All agents are implemented in a combination of C and Python and have a modular design. Each agent is a multi-threaded process split between a Python control plane and a C data plane communicating over shared memory. The control plane written in Python is in charge of communicating with the coordinator and performs the statistical computations using libraries such as NumPy and SciPy. The data plane written in C is in charge of accessing low level socket APIs to reduce the client overhead and to collect and store samples.

Throughput Agent: This agent leverages `epoll_wait` [1] to manage connections and is in charge of loading the server without measuring latencies. It is used only in asymmetrical deployments in cooperation with one of the two following agents, that can measure RPC latency.

Latency SW-timestamping Agent: This agent performs purely software timestamping. It relies on the busy polling functionality [5] introduced in Linux 3.11 which allows to poll the NIC instead of depending on interrupts. Given the blocking nature of this agents, it limits the load and the inter-arrival distribution of requests. Thus it needs to be used in an asymmetric setup with throughput agents to correctly measure latency and to generate the necessary load according to the expect inter-arrival distribution.

Symmetric timestamping Agents: The third type of agent is a symmetric agent that timestamp packets to measure RPC end-to-end latency. Two different implementations of this agent are provided. The first one relies on NIC-based hardware timestamping to measure RPC end-to-end latency. This version entirely rely on the Linux kernel functionnality for hardware timestamping and it can be used only if a NIC with the ability to hardware timestamp packets is available. The second version only relies on software timestamping to measures RPC end-to-end latency. While this version is less precise than the hardware timestamping, it has the advantage to be compatible with every Linux distribution regardless of the NIC being used.

The challenging parts of the implementation are symmetric agents, since they are not blocking and thus requires extra logic and data structures to correctly compute end-to-end latencies.

The Linux kernel provides an asynchronous API to collect timestamps [6]. For every `sendmsg` system call, the corresponding TX timestamp is propagated to the userspace through an `EPOLERR` for the equivalent socket. This API ensures that if an outgoing packet is fragmented, then only the first fragment is timestamped and returned to the sending socket. While this API ensures that if the packet is sent, the corresponding TX timestamp will be propagated to the userspace, there are no guarantees that the events received by the socket that is handled by `epoll_wait` contains the TX timestamp event before the response from the server. Thus, special care needs to be taken when receiving a response to make sure that the TX timestamp from the corresponding request was already collected. If the TX timestamp was not yet collected, we need to manually extract it before computing the end-to-end latency with the extracted RX timestamp from the response.

The Linux kernel provides a synchronous API to retrieve the RX timestamp [6]. The RX timestamp is part of the metadata to the `recvmsg` system call and corresponds to the receive timestamp of the frame that carried the last byte return by the system call. The API ensure that, for message-based sockets, that the entire response will be read by the `recvmsg` system call in a single operation [4].

R2P2: In order to provide packet timestamping in LANCET using R2P2 as transport protocol, we decided to extend R2P2 with the ability to timestamp packets instead of bypassing the R2P2 API in LANCET, which could resort in unwanted side effects.

By doing so, the client can easily timestamp packets using the R2P2 API. The API exposed is a non-POSIX API that has an asynchronous design allowing applications to easily send and receive independent RPCs. The client can decide if he wants to enable hardware timestamping or if he will manually timestamp packets in software. A convenient data structure called `r2p2_struct` has been extended to carry the TX timestamp when calling `r2p2_send_req`. This structure can either contain the software timestamp created by the client or it will automatically carry the generated hardware timestamp, if enabled by the client.

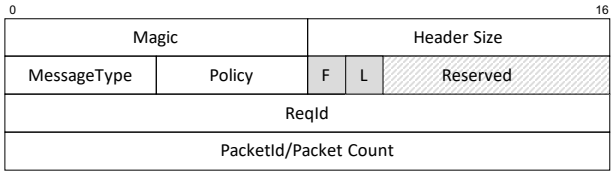

Figure 3: R2P2 Header Format

The success callback function will contain the TX and RX timestamps when hardware timestamping is enabled or the TX timestamp when using software timestamping. With this architecture, writing a timestamping R2P2 agent only requires ∼50 LOC.

Having the TX and RX timestamps in a single callback function; it becomes trivial to compute the end-to-end latency for R2P2 messages in LANCET.

To extend R2P2 with timestamping, we used the fact that R2P2 is a UDP-based transport protocol. Thus, we were able to extend it with the ability to hardware timestamp packets using the Linux API.

Given that long RPCs requests can be fragmented, we needed to make sure that we use the timestamp of the first packet sent when computing the end-to-end latency. To do that, a simple function is used when collecting the TX timestamps that keeps the lowest TX timestamp value received for a specific requests. This ensure that the first packet leaving the NIC is used when computing the end-to-end latency.

Since multi-packet requests induce multi-packet responses, we needed to be vigilant and use the RX timestamp of the last packet received. To ensure that the message received is indeed the last one, we can use the corresponding flags present in the header of a R2P2 message (Figure 3). There are currently two flags (`F, L`) which respectively denote the first and last packet of a request.

UDP: Implementing timestamping agents in LANCET using UDP as transport protocol is similar to the implementation of R2P2 timestamping agents, except that it does not require as much work as in R2P2 given that most of the work is done by the Kernel.

Indeed, if packet fragmentation happens when sending a datagram, the Linux API ensures that only the timestamp of the first packet sent will be propagated to the userspace via `EPOLERR`. Similarly, for message-based sockets, such as UDP, the entire response will be read by the `recvmsg` system call in a single operation [4]. Thus, we do not need to provide extra logic to deal with fragmentation when receiving a response. Having all this guarantees, we did not need to take extra precautions when implementing timestamping UDP agents. Reading and storing the TX and RX timestamps as they arrive was enough to correctly compute the end-to-end latency.

The only precaution that was taken with all transport protocols is to make sure that before computing the latency, the TX timestamp was correctly collected given that no order is guaranteed when polling for new events of a socket.

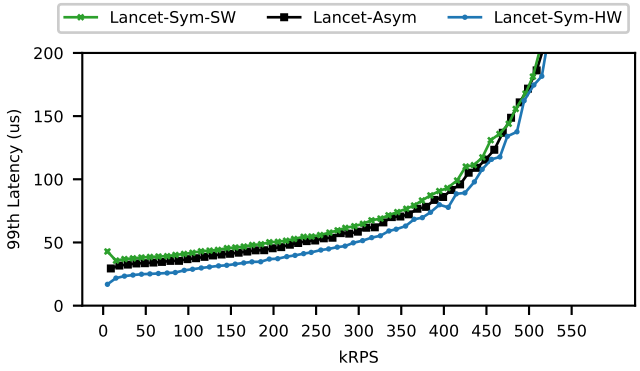

Figure 4: Latency vs throughput graphs for a 5-client experiment with average service time of $\bar{S} = 10\mu s$ using R2P2 as transport protocol.

4 Experiment

Our experiment aims to compare three different configurations in LANCET. The experiment is performed on UDP and R2P2 agents, with both the asymmetrical and the symmetrical deployments. To evaluate the accuracy of NIC-based hardware timestamping, the asymmetrical deployments are executed with software and hardware timestamping.

4.1 Experimental setup

The experiment uses 5 clients and one server machine. Each client machine is configured with 15 threads and a service time of $\bar{S} = 10\mu s$. The server is an RPC server on top of DPDK with 6 cores and synthetic service times.

4.2 R2P2 experiment

Figure 4 shows the 99^{th}-percentile tail latency as a function of the throughput for the three configurations. The symmetrical deployment using throughput and latency agents measures the tail-latency with more precision than the asymmetrical deployment using software timestamping. The reason is that the latency agents rely on the busy polling functionality in Linux, while the asymmetrical software timestamping agents are not blocking and leverages `epoll_wait`. Because of the asynchronous nature of the asymmetrical agents, computing the tail-latency manually becomes naturally less precise.

The benefits of using hardware timestamping can be seen in Figure 4. The agents systematically measure tail-latency more accurately than the other two configurations.

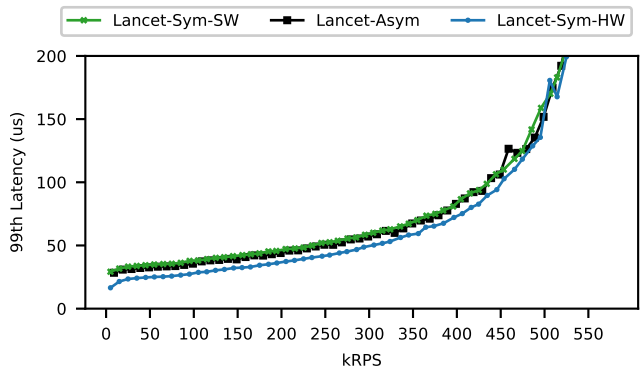

Figure 5: Latency vs throughput graphs for a 5-client experiment with average service time of $\bar{S} = 10\mu s$ using UDP as transport protocol.

4.3 UDP experiment

Figure 5 shows the 99^{th}-percentile tail latency as a function of the throughput for the three configurations. As in Figure 4, we notice that the asymmetrical deployment using NIC-based timestamping measures the tail-latency more accurately than the other two configurations.

The same behavior as in R2P2 is observed using UDP. LANCET reports a latency that is $\sim$10µs lower when NIC timestamping is available compared to software-based timestamping. This confirm the original expectation of the benefits of NIC-based timestamping. Making it the ideal configuration to eliminate the client bias and precisely measure RPCs end-to-end latency in LANCET.

5 Conclusion

LANCET now supports three different transport protocols, namely TCP, UDP and R2P2 [3]. Each transport protocol is composed of multiple agents allowing to efficiently measure RPCs end-to-end latency in several configurations.

Each protocol has a specific agent that can leverage NIC-based hardware timestamping to measure RPCs end-to-end latency eliminating the client bias. Our experiment shows that using this agent results in more accurate measurements.

References

[1] *epoll(7) Linux User's Manual*, May 2019.

[2] Lancet: A self-correcting latency measuring tool. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, 2019), USENIX Association.

[3] R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, 2019), USENIX Association.

[4] *recvmsg(2) Linux User's Manual*, May 2019.

[5] *socket(7) Linux User's Manual*, May 2019.

[6] Kernel timestamping documentation. https://www.kernel.org/doc/Documentation/networking/timestamping.txt.

[7] KUROSE, J. F., AND ROSS, K. W. *Computer Networking: A Top-Down Approach (6th Edition)*, 6th ed. Pearson, 2012.