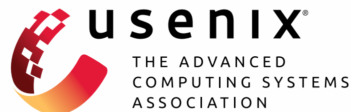


Lancet: A self-correcting Latency Measuring Tool

Marios Kogias, *EPFL;* Stephen Mallon, *University of Sydney;* Edouard Bugnion, *EPFL*

https://www.usenix.org/conference/atc19/presentation/kogias-lancet

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Lancet: A self-correcting Latency Measuring Tool

Marios Kogias[1] Stephen Mallon[2] Edouard Bugnion[1]

[1]*EPFL, Switzerland* [2]*University of Sydney*

Abstract

We present LANCET, a self-correcting tool designed to measure the open-loop tail latency of µs-scale datacenter applications with high fan-in connection patterns. 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. The workload configuration, the client infrastructure, or the application itself could, under circumstances, prevent accurate measurement. Because of its design, LANCET is also extremely easy to use. In fact, the user is only responsible for (i) configuring the workload parameters, *i.e.,* the mix of requests and the size of the client connection pool, and (ii) setting the desired confidence interval for a particular tail latency percentile. All other parameters, including the length of the warmup phase, the measurement duration, and the sampling rate, are dynamically determined by the LANCET experiment coordinator.

When available, LANCET leverages NIC-based hardware timestamping to measure RPC end-to-end latency. Otherwise, it uses an asymmetric setup with a latency-agent that leverages busy-polling system calls to reduce the client bias.

Our evaluation shows that LANCET automatically identifies situations in which tail latency cannot be determined and accurately reports the latency distribution of workloads with single-digit µs service time. For the workloads studied, LANCET can successfully report, with 95% confidence, the 99th percentile tail latency within an interval of $\leq$ 10µs. In comparison with state-of-the-art tools such as Mutilate and Treadmill, LANCET reports a latency cumulative distribution that is $\sim$20µs lower when the NIC timestamping capability is available and $\sim$10µs lower when it is not.

1 Introduction

Today's webscale datacenter applications such as search, social networking, and e-commerce all rely extensively on the decomposition of online, data-intensive queries into smaller subqueries that process data directly from the memory of hundreds or thousands of tightly-interconnected servers to ensure service-level objectives, scalability and availability [4,11,12,33,34]. The combined advancements in hardware technology (*e.g.,* 10-100Gbps NICs, cut-through switches, NVMe), system software (*e.g.,* dataplanes [6,49]), and data management systems (*e.g.,* in-memory databases and key-value stores [17,47,48,55]) now allow µs-scale interactions between application components [5]. The increased number of components involved in a single query and the extensive use of high fan-in, high fan-out patterns have shifted the performance focus to tail-latency considerations [11].

This emerging µs-scale computing era is characterized by new key performance metrics such as the tail-latency service-level objective (SLO), *e.g.,* 99th *percentile* $\leq$ 500µs [6,29]. To put this into perspective, 500µs is one order of magnitude longer than an in-memory relational database processing TPC-C [55] and two-to-three orders of magnitude longer than basic operations on a key-value store [17,35,48]. Yet, it is shorter than an operating system quantum, a TCP retransmission time-out, or the DVFS governor's reaction time [36]. This requires complete rethinking of traditional assumptions about systems, stacks, protocols, and applications [5,17,48,53]. A large body of research focuses on the systematic characterization and reduction of tail latency effects [6,13,14,23,27–29,34,36,50].

While throughput can easily be measured, tail latency is harder to capture and characterize in a statistically meaningful manner, as it depends on a number of factors beyond the workload itself. These factors include the choice of a tool with overheads and biases, its precise configuration, and the experimental methodology. The literature describes many pitfalls specific to latency, *e.g.,* Treadmill [58] discusses situations in which: (i) the inter-arrival request distribution does not match the production environment; (ii) the measuring methodology silently masks some tail behaviors; (iii) the measuring tool affects the measured end-to-end latency because the measuring granularity is too coarse-grained or because the clients are overloaded.

This matches our own experience in building and evaluating multiple research systems for µs-scale computing [6,30,

50, 51], which used either modified versions of Mutilate [33] or home-grown latency-measurement and load-generation tools. While these tools measured the tail latency of our systems as a function of the load, we were required to unscientifically tweak a large number of workload and system parameters in an ad-hoc manner by: (i) repeatedly increasing the number of load-generating clients until stability; (ii) repeatedly increasing the number of outstanding requests (*e.g.,* number of connections) until the tail-latency diverges at saturation, as expected in an open-loop process; (iii) and last, but not least, running each experiment "longer" with the hope of reducing result jitter.

This paper introduces LANCET, 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. LANCET is self-correcting as it relies on on-line statistical tests to determine situations in which tail latency cannot be accurately measured. This includes situations when (i) the workload configuration, and in particular the number of client connections, leads to closed-loop behavior; (ii) the infrastructure (*e.g.,* number of machines) cannot generate the desired load without introducing client bias; (iii) the service time of the workload itself is heavy-tailed distributed.

Because it relies on statistical methods within its control system, LANCET is also easier to use than existing tools. While the scientist specifies the infrastructure used for an experiment (*e.g.,* number of client machines), and the workload itself (*e.g.,* mix reads and writes, distribution of keys and values, number of client connections, maximum number of outstanding requests per connection *etc.*), LANCET then automatically determines, using statistical tests, what can be measured and at which confidence interval. LANCET's control system internally sets additional experimental parameters such as the duration of the experiment and its warmup phase.

Finally, LANCET relies on state-of-the-art, hardware-based measurement techniques that combine NIC timestamping in hardware and userlevel matching of packets to RPCs. This approach noticeably eliminates the client bias, and increases the accuracy of individual measurements without creating a long-term dependency on immature kernel-bypass protocols stacks and libraries.

This paper contributes the methodology, design, and implementation of LANCET, with the following novel features:

- LANCET measures the open-loop tail latency of a workload using only two user-provided parameters: the target load level and the desired confidence interval at the target tail percentile. For this, it relies on proven statistical methods such as hypothesis testing to configure the experiment methodology parameters.

- LANCET is self-correcting and reports "N/A" when no statistically-sound tail latency can be measured. This can be due to limitations in workload specification, client infrastructure, or because the service time distribution has high variability.

- LANCET clearly separates (i) the methodological considerations, implemented by the LANCET controller, (ii) the measurement tool, implemented by a combination of agents, and (iii) the workloads and application-level protocol support, implemented in an extensible manner by the LANCET agent's internal API.

- LANCET is designed with stability and production deployment in mind, with a focus on Ethernet-based protocols. It therefore uses exclusively the standard Linux kernel-based implementations of networking protocols. For applicable NICs, LANCET supports hardware-based timestamping to measure TCP-based RPC latencies for improved measurement accuracy. Our work demonstrates that kernel-bypass is not necessary to achieve precise µs-scale client-side measurements.

Our evaluation of LANCET with workloads with synthetic service times demonstrates that it (i) automatically identifies the right number of samples necessary for the target experiment accuracy and result convergence; (ii) accurately reports the latency distributions for workloads with service time as short as $\bar{S} = 1\mu s$; and (iii) provides substantially more accurate results than Treadmill [58] and Mutilate [33], state-of-the-art tail latency measurement tools.

LANCET is open-source and can be found at `https://github.com/epfl-dcsl/lancet-tool`. The rest of the paper is structured as follows. We discuss the necessary background (§2), analyze a latency experiment (§3), and discuss the design (§4) and implementation of LANCET (§5). We then evaluate LANCET (§6) and compare our methodology to existing tools (§7), and conclude (§8).

2 Background

The accurate measurement of the latency of any software application serving RPCs requires the appropriate combination of metrics, tools, workloads and experimental methodology: (i) the metrics determine which type of latency is being measured for a certain load, whether mean, median, tail (*e.g.,* 99th percentile), or the empirical cumulative distribution function (ECDF); (ii) the choice of tool determines the precision of the benchmark; (iii) the choice of workload determines the degree of realism, generality, and relevance of the experiment; (iv) the choice of methodology determines the overall soundness of the results, their accuracy, and reproducibility.

The high-level process is straightforward: the tool acts as an RPC client which generates requests for the system(s) under test. The tool timestamps requests and corresponding replies to determine the end-to-end latency. The requests themselves are determined in a workload-specific manner (*e.g.,* a mix of

get and set with a specific distribution of keys). The individual request inter-arrival time typically follows a Poisson process for a given rate λ. For a fixed rate λ, scientists typically report the full ECDF or the complementary cumulative distribution or tail distribution (CCDF), often on a log scale, to highlight the tail latency levels (99th, 999th, *etc.*). To study the impact of load on latency, scientists repeat the fixed-rate experiment for different λ and report the tail latency as a function of the load [6, 14, 22, 29, 29, 33, 37, 50, 58]. Finally, for dynamic experiments that mimic daily datacenter patterns, the tool dynamically adjusts λ according to a diurnal (or accelerated) time pattern [4, 6, 51, 56].

2.1 Taxonomy of tools

We attempt to make a taxonomy of the existing tools for generating load and measuring latency, the techniques used and the main design decision.

Packet vs. RPC generators: At the highest level, latency measuring tools can be easily classified into packet generators, which measure a network device or a network function, and application RPC generators, which measure a server.

Packet generators use stateless network packets to measure the throughput and the latency of datacenter network equipment such as switches and routers as defined in RFCs [7, 39]. These tools can be implemented to achieve different levels of precision in software. For example, MoonGen [19], TRex [8], and netperf [46], rely on hardware timestamping facilities in modern NICs (*e.g.,* MoonGen) or use custom hardware appliances such as Spirent [54] or IXIA [24].

Application RPC generators measure the latency of client-server interactions using protocols such as http or memcached's binary protocol, typically implemented on top of TCP or RDMA connections. These tools provide advanced workload-generation capabilities. For example, Mutilate [33, 45] can model Facebook's various uses of key-value stores [4], YCSB [10, 57] can generate a Zipfian distribution of keys, and CloudSuite [9, 20] offers a mix of applications. For the rest of the paper, we will focus on RPC generators.

Open-loop vs. Closed loop: There are two main ways to control the flow of requests to the target. An open-loop system models $n = \infty$ clients that send requests to the target according to a rate λ and an inter-arrival distribution, *e.g.,* Poisson. A closed-loop system bounds the maximum number of possible outstanding requests at any given time. The distinction between an open and a closed loop system is a property of a specific deployment and the same system can be deployed under different scenarios, *e.g.,* a key-value store may serve only a few blocking clients (*i.e.,* closed-loop) or thousands of application servers, which is best modelled as open-loop. Testing for the right scenario is crucial because open-loop systems can lead to large queuing, and thus longer tail latencies, whereas closed-loop tail latencies are typically bounded by the number of possible outstanding requests. Tools such as Treadmill and Mutilate are open-loop systems, while others such as YCSB are closed-loop systems.

Generating the necessary load: Precise tools are typically used to evaluate the benefit of innovations in new hardware, protocol designs, kernel bypass architectures, networking stacks, operating system configurations, or applications. Leading research systems today can deliver high-throughput solutions that easily scale to millions of requests per second, even on commodity hardware. As a consequence, the load-generation and latency-measuring tools, which typically run on reference vanilla Linux infrastructure, must be distributed on multiple client machines to saturate a single server [6, 33, 50].

Multi-machine setups follow two basic design patterns. First, in symmetric generators, all client machines generate load and measure latency. Then, an external agent accumulates and processes the collected results to report the aggregated verdict. This category includes YCSB [10], Treadmill [58], CloudSuite [20], memaslap [43], *etc.* Unfortunately the open-source versions of these tools provide no coordinator or aggregator script to run them in a distributed fashion.

Second, the asymmetric design splits the client machines between load-generating and latency-measuring. The bulk of the load is generated by client machines that generate requests according to a specific inter-arrival distribution, *e.g.,* Poisson, in an open-loop manner without measuring latency, while a separate, dedicated client machine makes closed-loop requests to the same server and measures its latency. By reducing the system load on the latency-generating thread, such tools reduce client bias in the measurement. Mutilate [33] is the most well-known tool in that category.

Point of measurement: Latency can be measured at different points in the system resulting in different levels of accuracy. This includes the actual wire, the NIC, the Ethernet driver, the in-kernel socket layer, or the application itself. The point of measurement has a large impact on precision. According to Primorac *et al.* [52], (i) the packet generators using hardware-based NIC timestamping such as MoonGen can accurately measure the latency of stateless network functions up to the 99.99^{th} percentile whereas (ii) the best software solution relying on kernel bypass can only measure up to the 99th percentile, and (iii) the solutions relying on the traditional networking stack should not be used at all for µs-scale latency measurements.

NIC-based timestamping is available on mainstream NICs. Intel NICs, such as 10Gbe 82599 and x54, or 40Gbe x710, implement hardware timestamping only to support IEEE 1588 Precision Time Protocol [18]. This restricts the type and amount of packets that can be hardware-timestamped. The MoonGen packet generator takes advantage of this precise, yet

restrictive mechanism. The Mellanox NICs, *e.g.,* ConnectX-4 [42] or newer, offer general-purpose hardware timestamping support to all incoming and outgoing packets. The Linux kernel provides support for hardware timestamping via the Linux socket interface, yet deriving RPC timestamps from packet timestamps is challenging, as later described.

Another way to increase precision and reduce jitter is to leverage kernel bypass and NIC polling at the client. Tools such as MoonGen and T-Rex use the DPDK [16] toolkit for better performance and precision. Unfortunately, kernel bypass limits application and protocol support, and requires using less-proven protocol stacks as part of the experiment.

Reporting results: From a methodology perspective, most tools depend on histograms to compute latency percentiles, thus avoiding keeping all the recorded latency samples. Histograms with fixed bucket sizes, as used by Mutilate, can affect the reported results by masking tail phenomena, if not configured properly. Some tools such as Treadmill [58] propose a user-defined calibration phase to determine the bucket allocation. Other tools, such as TailBench [29], use dynamic histograms, whose bucket sizes change over the execution of the experiment. Finally, few tools, *e.g.,* Mutilate, allow collecting all latency samples and save them in a file to be used for plotting the ECDF.

2.2 Configuration burden

Load generators put the methodological burden on the scientist who configures it. For example, a scientist using Mutilate must first determine the time for each load experiments (default=5s), which must be long enough to be statistically sound; then specify the number of machines, threads and overall number connections for the load-generating agents, and the maximum number of outstanding requests per connection; and finally specify the configuration of the latency-measurement agent, which operates as a closed-loop with one outstanding request a time.

This configuration setup has subtle implications as (i) increasing the number of machines reduces client bias [58]; (ii) increasing the number of open sockets reduces the throughput of the server because of operating system overheads [6]; (iii) increasing the maximum number of outstanding requests per socket allows for batching and increases throughput; (iv) the product of the number of connections $\times$ the number of outstanding requests must be larger than the bandwidth-delay product of the workload if the scientist wishes to measure the open-loop tail latency of the service.

Figure 1 illustrates the challenge via the study of an out-of-the-box memcached/Linux deployment with Mutilate configured with 320 and 144 connections with one outstanding request each. We report the 99th tail latency as a function of the load. The orange curve (144 Connections) operates as a closed-loop, with the clients unable to generate the target rate,

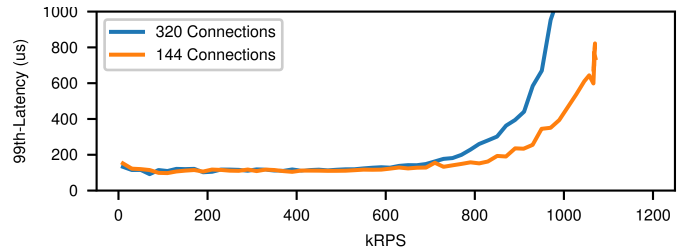


Figure 1: 99th percentile latency for memcached USR measure with Mutilate with 144 and 320 connections with 1 outstanding request per connection.

and without ever saturating the server. The lower reported tail latency is merely a reflection of the limited number of outstanding requests. This experiment can lead to false conclusions, *e.g.,* on the maximum throughput that meets a SLO (*e.g.,* $\leq 300\mu s$).

2.3 Statistics Background

This section provides the sufficient background to understand our use of statistical methods in LANCET.

Hypothesis testing: Statistical testing follows a specific thought process. Initially, the statistician formulates a null hypothesis implying that there is no relation between two populations and the observations are the results of pure chance. She then identifies a test statistic that can assess the truth of the null hypothesis and computes the p-value. p-value gives the probability of the given test statistic resulting in the observed value if the null hypothesis is true. The smaller the p-value, the stronger the evidence against the null hypothesis. Finally, she compares the p-value to the α value, which corresponds to the level of confidence. If the p-value is less than α, she rejects the hypothesis and therefore conclude that the effect she observed was not due to random chance.

LANCET uses the following tests. First, the **Anderson-Darling** test checks whether a group of samples comes from a certain probability distribution and was chosen because it is less sensitive to outliers compared to similar tests, *e.g.,* the Kolmogorov-Smirnov test [31]. We use that to validate the inter-arrival request distribution. Second, the **Augmented Dickey Fuller** test [15] checks a series of samples for stationarity. We use the ADF test to determine the duration of the warm-up phase and whether the experiment results change over time. Finally, we use the **Spearman** rank correlation coefficient and the associated p-value to check if a series of samples is autocorrelated when checking for *iid*-data.

IID-data: Most types of hypothesis testing or general statistical processing, such as the calculation of confidence intervals, require samples that are *independent and identically distributed (iid)*. When running a latency experiment, latency

Latency Experiment Concerns			
Workload		**Methodology**	**Measuring Tool**
transport protocol	connection balance	system stability	workload-compliant
application protocol	open/closed queueing	unbiased result processing	methodology-compliant
request types and ratio	outstanding requests/connection	result convergence	measuring bias free
connection count	inter-arrival distribution	distribution coverage	

Table 1: Classification of concerns related to running a latency experiment into workload, methodology, and measuring tool-specific components. We advocate that the Workload column has to be user defined, while Methodology and Measuring Tool columns have to be handled systematically by the measuring framework.

samples are naturally identically distributed since they come from the same target server. Sample independence, though, is challenging to meet because of queuing effects. The end-to-end latencies of two requests that are queued back-to-back are dependent because the latency of the latter request includes the service time of the prior. While independence cannot be taken for granted, it can be tested, with autocorrelation being the standard way to check independence for a series of samples.

Confidence Intervals: We focus here on the confidence intervals for tail latency of a *single* execution, assuming that the system environment remains identical and stable during the entire experiment. The confidence intervals for a distribution's percentiles can be computed in closed form when the data are *iid*. The formula identifies, with a certain level of confidence, two threshold values that belong to the collected samples, between which the value for the specific percentile is expected to be found. Formulas 1, 2 give the indices of those two threshold values in the sorted of collection of samples [31] for a certain confidence level γ.

$$j \approx \lfloor np - \eta\sqrt{np(1-p)} \rfloor \quad (1)$$

$$k \approx \lceil np + \eta\sqrt{np(1-p)} \rceil + 1 \quad (2)$$

where n is the number of samples, p is the percentile, and η is defined as $N_{0,1} = \frac{1+\gamma}{2}$. For example, for 10,000 *iid* samples ($n = 10000$), the confidence interval for the 99-th percentile with 95% confidence ($\gamma = 0.95$, so $\eta = 1.96$) will be between the values with indices $j = 9880$ and $k = 9921$.

Note that determining confidence across different executions of the same experiment is challenging as the system's boot-time and application initialization can have a persistent effect on performance, leading to the hysteresis problem described in Treadmill [58].

3 Experiment Decomposition

Our goal is to build a latency-measuring tool that is precise and simplifies the configuration burden discussed in §2.2, with the explicit objective to identify situations in which the configuration cannot lead to a statistically meaningful result.

Table 1 classifies concerns related to a latency experiment into three main categories: workload, methodology, and measuring tool. These concerns often correspond to user-defined parameters in most of the existing tools and can be easily misconfigured. This decomposition will guide the design of modular, self-correcting latency-measuring tools. We claim that the workload-specific parameters have to be user defined, otherwise the experiment is insufficiently described. The methodology and measuring tool concerns have to be systematically managed by the measuring framework to reduce the pitfalls induced by the user misconfiguration.

Workload: The first aspect of a latency experiment is the actual workload and a large set of the configuration parameters refer to the workload specification. The experiment workload is both application- and deployment-specific, meaning that the same application should be tested differently if the deployment environment is also different. The workload includes the application specific parameters (*e.g.,* `get:set` ratio, request size distributions, TPC-C request mix, *etc.*), the application-level protocol (*e.g.,* `HTTP`, binary memcached, *etc.*) and the network-level protocol (*e.g.,* UDP vs. TCP). The definition of the workload also includes the client assumptions, *i.e.,* the number of expected client connections, the maximum number of outstanding requests per connection, and whether clients operate in an open- or a closed-loop system.

Critically, the specification of the workload is independent of the measuring tool , but affects the results, which could lead to unrealistic or wrong conclusions. For example, one cannot meaningfully report the open-loop tail latency of a workload with an insufficient number of connections, or insufficient outstanding requests per connection.

Measuring methodology: The second aspect of a latency experiment is the methodology, which describes how the latency samples are collected and processed. Examples of configuration parameters that are relevant to the methodology are the experiment duration, the number of collected samples, and the number and size of the histogram buckets. Reducing the number of configuration parameters related to methodology is a major goal of our design.

Regarding the latency sample collection, a good method-

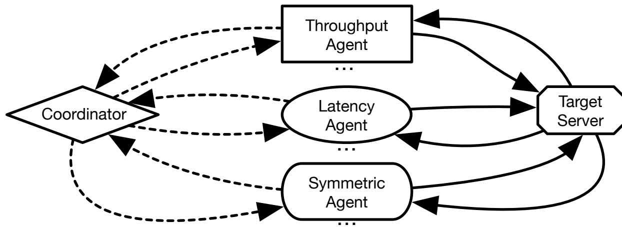


Figure 2: Lancet's architecture depicting a coordinator (C), throughput agents (TA), latency agents (LA), symmetric agents (SA), and the target server under test. The dashed arrows correspond to the LANCET API while the solid ones are application RPCs

ology should first ensure that the system under test is in a steady state to avoid measuring transient phenomena. Then it should ensure that the collected results converge and that all desired tail behaviors are covered. Finally, during the result processing, it should avoid adding statistical bias, *e.g.,* by the misconfiguration of histograms.

Measuring tool: Finally, the last part of a latency experiment is the actual client software used to collect the latency samples. Examples of parameters related to the tool are the number of client machines or threads used in the experiment, and whether hardware or software timestamping is used. The tool should be able to implement the specific methodology, generate the target workload accurately, and measure latency without adding too much client bias.

4 Design

4.1 LANCET infrastructure

Figure 2 shows the basic LANCET overview, which splits the methodology from the actual measuring tool and workload generator according to §3. LANCET is a by-design distributed tool that consists of a coordinator (C) and various measuring agents. The coordinator is in charge of the experiment methodology (see §4.3) and communicates with the agents over the LANCET API (Table 2). The measuring agents drive the workload via application RPCs generated based on application-specific random distributions. The agents also measure latency precisely, identify cases of workload violations, and run statistical tests.

Figure 3 describes a typical agent state transitions triggered by the coordinator via the API for a fixed-load experiment. From an idle state (*Idle*), the agent transitions into the loading phase (*Load*), where it attempts to issue *l* requests per second to the server. During that period the agent does not record latency. The agent eventually transitions into the measurement phase (*Measure*) specified by a sampling rate (*sr*) and a number of latency samples to collect (*s*). The agent can

Request Type	Request Params	Reply
start_load	load (rps)	ACK
start_measure	#samples sampling rate(sr)	ACK
get_throughput	None	Throughput (rps) Correct IA (T/F)
get_latency	None	Latency CI Stationary (T/F) IID (T/F, sr)
exit	None	ACK

Table 2: The LANCET coordinator API with the information returned by the agents on each call. For the `get_throughput` and `get_latency` requests, the agents also reply information related to the Inter-Arrival distribution (IA), the latency Confidence Intervals (CI), and whether the collected samples are stationary and *iid*. If they are not *iid*, the reply contains the target sampling rate necessary to get *iid* data.

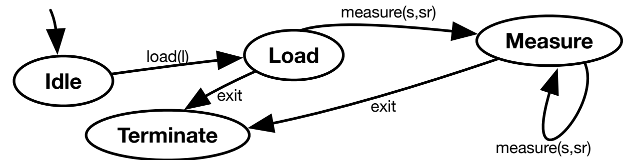


Figure 3: Lancet agent's state transition. Arrows represent messages from the coordinator.

stay in that state while the *sr* and *s* parameters can change. Finally, the coordinator decides to terminate the experiment (*Terminate*) via an `exit` message. At any point in time, while the agent is in the *Load* or *Measure* phase the coordinator can ask for the current throughput and latency.

4.2 Measurement options

Figure 2 shows that LANCET implements three agent types, selected to match the capabilities of the available hardware, the measuring methodology and the target experiment granularity. This way LANCET can support both symmetrical and asymmetrical deployments described in § 2.1.

LANCET uses the asymmetrical model when the latencies are captured in software. This model reduces jitter by dedicating cores and even machines to only measure latency. The drawback is that the experiment collects fewer samples per time period. Furthermore, special care must be taken to ensure that the collected samples are representatives of the workload. For example, a latency agent should open multiple connections (*i.e.,* emulate multiple clients) to ensure that a server configured with an RSS NIC will use all cores.

LANCET uses the symmetrical model when the NIC offers the capability to timestamp all incoming and outgoing Ethernet frames and the Linux operating system exposes the infor-

mation to userspace (v4.14+ kernels). LANCET associates the hardware timestamping of packets to the end-to-end latency of RPCs. This is not straightforward because of the inherent mismatch between the stream-oriented TCP protocol and the message-oriented RPCs. Implementation details follow in §5.

4.3 LANCET's self-correcting methodology

LANCET's primary contribution is its novel, self-correcting methodology which follows the experiment decomposition and split of concerns described in Table 1, and tries to systematically and based on statistics, identify: (i) when the server is in a stable state to start measuring latency (managed by the user-defined warm-up time in other tools); (ii) if the collected latency samples converge and whether tail phenomena are fully covered (controlled by the user-defined experiment duration in other tools); (iii) how to process the collected samples and report latency without introducing statistical bias (histograms are mainly used for that purpose in other tools); (iv) the confidence intervals of the latency results (unlike most tools which simply report latency percentiles).

Figure 4 illustrates the state machine transitions of the coordinator when measuring the open-loop tail-latency of a server under a certain load. To run such an experiment, the scientist needs to provide, apart from the necessary workload specification (first column in Table 1), the following:

- the target load (`l`).
- the target confidence interval for a specific latency percentile, *e.g.,* 10µs interval for the 99th percentile with 95% confidence.

The output of such an experiment will be either the tail-latency percentiles with the corresponding confidence intervals or an indication that the specific experiment cannot be executed because some of the assumptions are violated, *e.g.,* the target load cannot be reached, the service time has high variability and the computed latency confidence interval is wider than the target, the client does not respect the workload specifications, *etc.*

System Stability: Initially, the methodology ensures that the target load can actually be reached before starting measuring latency, thus eliminating transient phenomena. Then, the methodology ensures that agents load the server while respecting the workload's specified inter-arrival distribution. This second confirmation is essential to avoid reporting misleading latencies. For this, every agent records the inter-transmission intervals of requests by recording request transmissions, ideally in hardware, but if necessary at the socket interface. Every agent runs an Anderson-Darling test to check whether the inter-transmission intervals follow the target inter-arrival distribution, *e.g.,* exponential in the case of Poisson inter-arrival. The controller exits the system stability step only when the load is reached according to the correct inter-arrival distribution.

Unbiased Result Processing: Each agent collects, according to the parameters (s, sr) set by the controller, s samples, each randomly sampled among the RPCs at rate of sr, *e.g.,* collecting 10,000 samples with a 1:20 sampling rate would require $\sim$200K RPCs. Sampling is necessary because the collected samples need to be *iid* to compute the confidence interval correctly. Computing confidence intervals on non-*iid* data will underestimate their size.

The *iid*-ness is confirmed or rejected by computing the autocorrelation of the collected latency samples sorted by their transmission time. To do so, latency-measuring agents compute the Spearman correlation of the collected latency samples shifted over time. We leverage the associated p-value to determine whether the correlation is significant or not. This correlation being significant implies that data that are close in time depend on each other, which is the result of them being queued back to back in the servers queue.

A way to reduce the autocorrelation is to decrease the sampling rate. The LANCET built-in parameters initialize the measuring phase with 10,000 collected samples with a first sampling rate of 1:5. If the autocorrelation is non-significant, the latency measuring agents report that the samples are *iid*. Otherwise, they report how much to reduce the sampling rate to achieve non-significant correlation. The latency measuring agents report the Pearson correlation co-efficient back to the coordinator as part of their latency results. To do so, the agents compute the autocorrelation for different lags and report the one that leads to a non-significant correlation. Based on the agents' replies, the coordinator decides whether to proceed to the next state or reduce the sampling rate accordingly if it fails to confirm *iid*-ness.

Result Stationarity: The methodology needs to identify whether the number of samples collected is sufficient for the results to converge to a stable distribution of latencies that does not change over time. To ensure stationarity, the methodology leverages an Augmented Dickey Fuller test [15]. Each latency-measuring agent sorts the collected latency results based on their transmission timestamp and runs the test. Again, the latency measuring agents report the result of the test to the coordinator. In cases where lack of stationarity is detected, the coordinator decides to increase the number of samples by 10,000 and retry. Otherwise it proceeds with the next check.

Determine the confidence interval: Finally, the methodology has to check if the results converge within the target confidence interval size. For that, we use the Formulas 1 and 2. Each latency measuring agent reports the confidence intervals for the latency percentiles to the coordinator. The coordinator ensures that the intervals from different agents

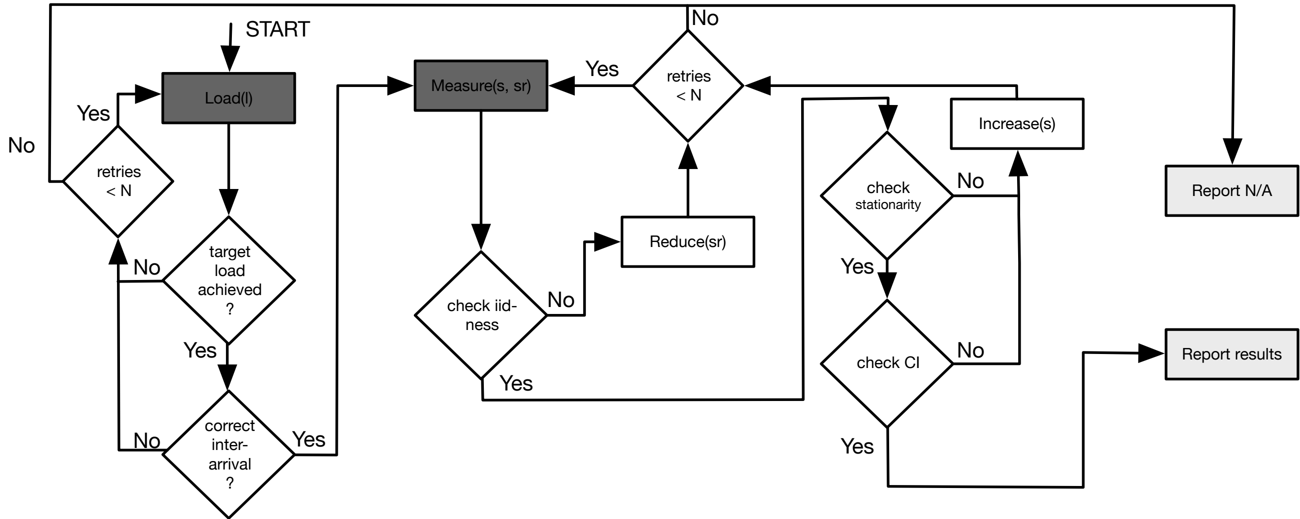


Figure 4: LANCET's experiment methodology implemented by the coordinator. Dark grey boxes correspond to messages from the coordinator to the agents. Light grey boxes show the experiment end.

are overlapping and computes their average. If the final confidence interval is wider than the user-selected target, the coordinator increases the number of samples by 10,000 and continues the experiment.

Termination: If the target confidence is reached, the coordinator finishes the experiment and reports the final latency percentiles with the equivalent confidence intervals. If the coordinator cannot reach the target confidence after a fixed number of failed retries, or if the experiment duration is above a certain threshold, it terminates the experiment, and reports that the specific experiment is not conclusive, the reasons why, along with the collected results so far.

5 Implementation

In addition to the design goals of §4, LANCET was implemented with robustness and long-term relevance in mind. LANCET is therefore built purely on functionality provided by the Linux kernel, using built-in drivers and protocol stacks. During development we identified some inconsistencies regarding hardware timestamping in the Linux kernel; our patch was merged in Linux kernel 4.19.4 [38].

The LANCET coordinator (Figure 2) is in charge of deploying the agents, communicating with them over sockets, driving their state machine according to Figure 3, and implementing the methodology of Figure 4. The coordinator is implemented in Golang. It relies on *goroutines* for easy distributed coordination and failure management, and consists of ~1000 lines of code. From those lines, ~300 of them implement the methodology described in § 4.3 and the rest implemented the LANCET API to communicate with the agents, manage collected results, *etc.* Thus, implementing a new coordinator logic for different experiment methodologies is relatively easy.

We implemented three different agents that can be used according to the available hardware, the measuring methodology, and the necessary experiment granularity. Our agents can achieve better measuring granularity compared to previous tools and can be used in both a symmetrical and asymmetrical deployment, independently of the available hardware. The agents are implemented in a combination of C and Python, and can be easily extended with new transport and application protocols.

Figure 5 depicts the structure of a multi-threaded LANCET agent. Each agent is split between a Python control plane and a C data plane communicating over shared memory. The Python control plane is in charge of communicating with the coordinator and performing the statistical computations. The choice of Python allowed us to take advantage of the rich Python ecosystem using libraries such as NumPy and SciPy. The choice of C for dataplane gave us direct access to low level socket APIs and reduced the client overhead. LANCET lancet currently supports TCP, UDP, and R2P2 [30] as transports, and Memcached, Redis, and HTTP as application protocols.

Throughput Agent: This agent leverages `epoll_wait` 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 depends on software timestamping and does not have any hardware dependencies. It improves the measuring precision over

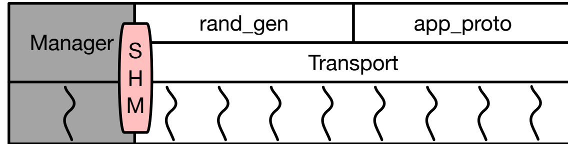


Figure 5: The structure of a LANCET agent. The grey part corresponds to the Python control plane, while the white part corresponds to the C dataplane communicating over shared memory (SHM).

other software-based tools, though, by leveraging the busy polling functionality introduced in Linux 3.11. Specifically, the `SO_BUSY_POLL` socket option allows blocking system calls to poll the NIC instead of depending on interrupts. While still dependent on userspace timestamping, this agent reduces client bias and measures latency with similar accuracy to kernel-bypass approaches. The blocking nature of this agent limits the load and the inter-arrival distribution of requests the agent can achieve. Consequently, this agent can only be used in asymmetric setups in conjunction with throughput agents that generate the necessary target system load according to the expected inter-arrival distribution.

Symmetric HW-timestamping Agent: Finally, we implemented a symmetric agent that leverages hardware timestamping to measure RPC end-to-end latency. This agent depends on the Linux kernel functionality for hardware timestamping added in kernel 4.14 for TCP. It also requires a NIC that timestamps all the incoming and outgoing packets, *e.g.*, the Mellanox ConnectX-4 [42]. The preferred deployment is based on symmetric HW-timestamping agents, as they improve on the latency agent in terms of precision and they can scale throughput while the coordinator can symmetrically collect results from all client machines, thus increasing the experiment accuracy.

The most challenging part of the implementation was the attribution of RPC latencies when requests and replies are layered on top of the stream-based TCP protocol, as used in the popular protocols, such as Memcached.

For TX timestamps, the Linux kernel provides an asynchronous API to collect timestamps, returning asynchronously one timestamp for each `sendmsg` system call. The notification is propagated to the userspace through an `EPOLERR` for the equivalent socket that is handled by `epoll_wait`. Along with the timestamp, the kernel also returns the number of the last transmitted byte this timestamp corresponds to. For example, if the first request has a size of 20 bytes, the notification will mention that this timestamp is associated with byte 20. For the second request of the same size, the notification will mention byte 40, *etc.* The same information is maintained by LANCET in userspace for validation purposes, and to deal with cases of coalescing or resubmissions.

The kernel provides a synchronous API to retrieve the RX timestamp: 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 returned by the system call. The LANCET application-parsing logic leverages this information to associate timestamps to replies of variable sizes: if the content returned by `recvmsg` consists of an incomplete reply, that timestamp is ignored; if the content contains the replies to multiple requests (which is possible because of TCP's streaming nature and coalescing in the socket layer), LANCET only considers the timestamp for the last reply returned in that call. The Linux kernel coalesces `sk_buffs` internally and keeps a single timestamp per `sk_buff` corresponding to the last arrival. Consequently, earlier received responses might appear to have later receive timestamps.

Our contribution to the kernel 4.19.4 [38] guarantees that each `recvmsg` system call will return the hardware timestamp that corresponds to the last byte read. Previously, this was only the case for software in-kernel timestamping.

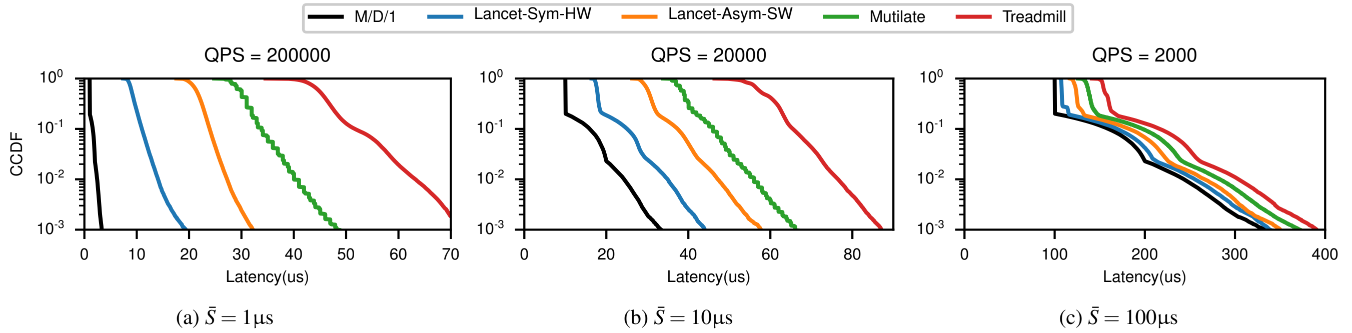
6 Evaluation

Our evaluation aims to answer three fundamental questions: (i) how does LANCET compare with existing RPC load-generating tools such as Mutilate and Treadmill (ii) how does LANCET's self-correcting methodology work in practice (iii) how LANCET performs in characterizing a server's behavior across different loads.

We answer these questions using a methodology in which the server's execution time is explicitly controlled. Doing so enables comparing the client-side measurements made by the tools to an idealized queueing theoretic model. We leverage an RPC server with synthetic service times following well-known distributions. Specifically, we tried a fixed, an exponential, and a bimodal distribution where 10% of the requests take $\sim 10\times$ longer to execute. To further reduce server-side overheads, our server uses the open-source IX operating system [6] configured with 1 CPU and adaptive batching disabled. The operating system overhead is $\sim 1\mu s$ of CPU execution time per request, which includes driver and network processing overheads. As baselines we use the opensource versions of Mutilate [1] and Treadmill [3]. For Treadmill, we had to make changes in order to build it for our setup.

To be able to compare with other tools, our synthetic server uses the `ascii-memcached` protocol. Clients submit `get` requests with for a 19-byte key (similarly to Facebook's USR [4]), the server spins for a configurable amount of time, and replies that that key was not found. We chose `ascii-memcached` because it is the only protocol supported by both Treadmill and Mutilate.

The idealized models correspond to the expected latency distribution, as determined by a discrete event simulation, assuming zero operating system overheads, zero network propagation delays, and zero client-side measurement overheads.



(a) $\bar{S} = 1\mu s$ (b) $\bar{S} = 10\mu s$ (c) $\bar{S} = 100\mu s$

Figure 6: Latency ECDF for an M/D/1 model and three deterministic workloads at 20% load.

For all of our experiments, we configure each client machine with 15 threads and 4 connections per thread with 1 outstanding request per connection. Also, we consider a Poisson inter-arrival distribution of requests.

6.1 Experimental setup

Our experimental setup uses 5 clients and one server machine connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The client machines are equipped with a Xeon E5-2637 @ 3.5 GHz and a Mellanox Connect-X4 NIC. The machines run an Ubuntu LTS 16.04 distribution running Linux kernel version 4.19.4. The systems are tuned to reduce jitter: all power management features, including CPU frequency governors and TurboBoost, and support for transparent huge pages, are disabled. The server is a Xeon E5-2665 @ 2.4 GHz with an Intel x520 NIC running the IX operating system.

6.2 Benefits of hardware timestamping

First, we compare the measuring granularity of LANCET with the measuring granularity achieved by Mutilate and Treadmill. For LANCET we consider both the hardware timestamping, symmetrical setup and the asymmetrical one based on the busy-polling agent. LANCET and Mutilate provide a way to run an experiment based on multiple machines, but for Treadmill there is no opensource coordinator script. Thus, we run one Treadmill instance on each client that contributes 1/5 of the load. Also, we modified Treadmill to save the collected latencies at the end of the experiment.

From a methodology perspective, we plot the latency CCDF for a deterministic service time distribution with different average service times. The load is set at 20% of the theoretical saturation, we range the average service time from $\bar{S} = 1\mu s$ to $\bar{S} = 100\mu s$. We collected 1M samples for each tool.

Figure 6 summarizes the experiment results. We observe that LANCET, for both configurations and in all three experiments, achieves lower measuring granularities when compared to the other tools because it reduces the client measuring overheads. Specifically, for $\bar{S} = 1\mu s$ hardware timestamping measures a 99th percentile tail of $14.1\mu s$ and the LANCET polling agent one of $27.3\mu s$. Mutilate measures $40\mu s$ and Treadmill reports $63\mu s$. Figure 6a also shows that Mutilate's line is not smooth because of the µs reported granularity, as opposed to nanoseconds reported by the other tools. Also, we see that LANCET aligns better with the theoretic results. For example, with $\bar{S} = 10\mu s$, the blue line nicely tracks the model; the offset between the two ($\sim 10\mu s$) is essentially due to the operating system overhead and the propagation delay. Finally, Figure 6c shows that the tools make a difference even for coarser grain tasks ($\bar{S} = 100$), where the operating system and propagation delay overheads are comparatively small.

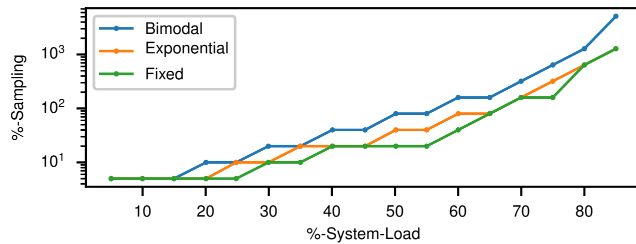
For the rest of our evaluation we will focus on the symmetric hardware-timestamping agent as it reports the most accurate results.

6.3 LANCET self-controlling dynamics

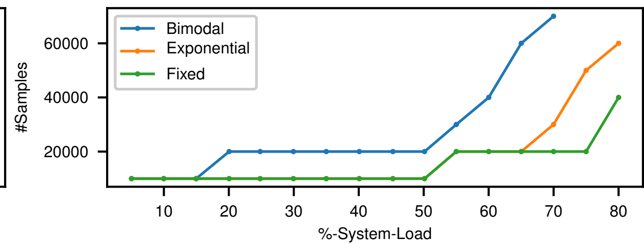
In the next series of benchmarks, we want to identify the impact of the self-correcting methodology and how the coordinator controls the experiment parameters based on the different service time distributions and the system load. To do so, we run the three different service time distributions across a variety of loads and we collect the necessary level of sampling to achieve *iid*-ness, and the number of samples necessary for a target confidence interval size of $10\mu s$.

Figure 7a shows the sampling rate that is necessary to the unbiased processing of the results caused by queuing effects. We observe that high-dispersion workloads (*e.g.,* bimodal) and higher load levels require lower sampling rates. This is expected as increasing either service time dispersion or load level leads to more queuing, thus more dependent samples.

In Figure 7b, we set the size of the target confidence interval for the 99th percentile latency to be $10\mu s$ with 95% confidence. The figure shows the number of collected samples, as decided by the coordinator, that are required to satisfy the result target. We observe that more samples are necessary to fulfill the constraint as the load increases, since higher



(a) Required level of sampling to guarantee *iid*-ness



(b) Required number of samples to achieve the target CI of 10µs

Figure 7: Dynamics of LANCET's self-correcting methodology based on load for three service time distributions with $\bar{S} = 10\mu s$

system load leads to higher latency variability.

The bimodal distribution shows an interesting behavior of the tool: With load $> 70\%$, execution stops after the maximum number iterations ($N = 10$) but the target CI expectations can not be met. Our experiment logs showed that the collected 99th percentile latency at 75% load is 411.333µs [-5.87µs, 7.56µs] at 95% confidence; this interval is $> 10\mu s$.

We also tested LANCET's self-correcting behavior with the lognormal distribution, which is a heavy tailed distribution. LANCET terminates without ever being able to confirm results convergence(CI < 10µs for the 99-th percentile latency), even at a low load of 20%. Thus, LANCET is effective in detecting heavy-tailed service time distributions.

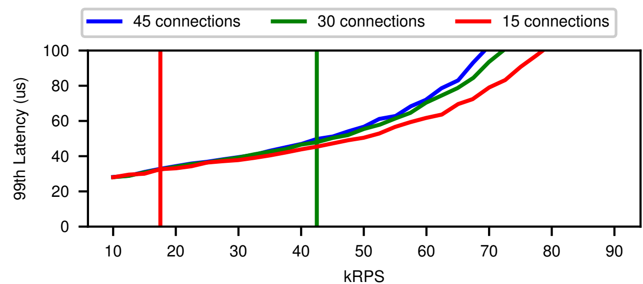


Figure 8: Impact of the inter-arrival distribution to tail-latency for a fixed with $\bar{S} = 10\mu s$. Vertical lines correspond to the load levels where Lancet reports inter-arrival distribution violations

6.4 Inter-Arrival distribution Impact

In the following experiment we try to showcase the impact of the inter-arrival distribution on the latency results and how LANCET identifies cases of inter-arrival distribution violations. We use the fixed synthetic time distribution with $\bar{S} = 10\mu s$ and we run a latency experiment across a variety of loads with different number of connections. We disable LANCET's checks for inter-arrival distribution and we only report whether there is a workload violation. To eliminate any system interference we configure LANCET with one connection per thread, and add connections by adding client machines.

Figure 8 shows the 99-th percentile latency as a function of throughput for the different connection count configurations. The vertical lines correspond to the load level that the equivalent configuration started violating the inter-arrival distribution. We observe that once LANCET reports a violation the curves start deviating. This experiment shows that cases as the one described in Figure 1 can be avoided by LANCET's self-correcting methodology.

6.5 Server characterization

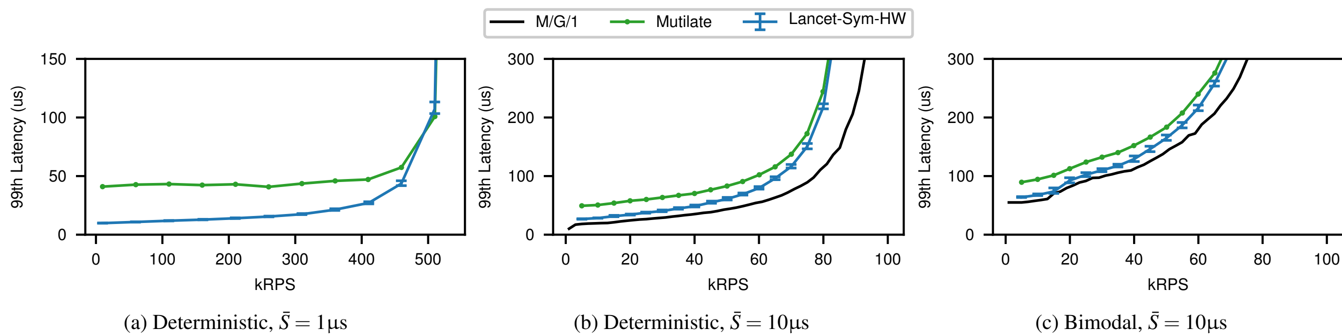
Figure 9 shows the 99th-percentile tail latency as a function of the load for three workloads. We compare LANCET with Mutilate as well as the idealized, zero-overhead theoretical model. Both tools use 5 machines – necessary to achieve the high loads required. For LANCET, we additionally report the confidence interval of each measurement. This experiment does not include Treadmill as Treadmill's open-source distribution does not support multi-machine deployments. Note that because of system overheads, the IX server cannot get close to the expected maximum load for Figure 9a which would be 1M RPS, thus we do not plot the theoretic curve.

We observe that LANCET reports latencies that closely match the idealized model across the entire load spectrum, to the point that it accurately reflects the two inflection points of the binomial distribution. We also observe how the size of the confidence intervals change across different distributions and system loads. For low loads and low service time dispersion, the interval is shorter than the maximum configured (10µs). For the bimodal distribution, the reported confidence interval is at its maximum configuration even for low loads.

7 Related Work

LANCET is one of the many contributions towards enabling reproducibility and accurate experimentation in systems research [25, 40].

µs-scale computing: Recent research focuses on µs-scale



(a) Deterministic, $\bar{S} = 1\mu s$ (b) Deterministic, $\bar{S} = 10\mu s$ (c) Bimodal, $\bar{S} = 10\mu s$

Figure 9: Latency vs throughput graphs for a 5-client experiment with average service time of $\bar{S} = 1$ and $\bar{S} = 10$

computing [5] both in operating systems and networking and either aim to optimize [6,26,35,49,50], or attribute the sources of tail-latency [29, 33, 34, 52, 58]. LANCET does not attempt to attribute the sources of the jitter. Instead, it provides a tool to measure µs tail latency precisely on an end-to-end basis to be used in similar research efforts.

Precise measurements: RPC generators [10, 20, 33, 43, 58] use software timestamping. However, researchers need more accurate tools to evaluate system's latency, *e.g.,* ZygOS used a modified version of Mutilate based on DPDK [16], and MICA [35] used a custom version of YCSB on DPDK. Software-based packet generators [8, 19, 46] also used DPDK for increased precision [52]. Hardware-based packet generators [24, 54] provide sub µs-scale precision with little jitter [52]. Some tools repurposed the IEEE PTP feature of standard NICs to measure packet latencies [19,32,44]. LANCET is the first tool that leverages hardware-based NIC timestamping for capturing latency for RPCs over TCP with even higher precision. In addition, LANCET uses the standard Linux networking stack for all experiments, proving a more realistic simulation environment. While we used Mellanox ConnectX-4 NICs in our experiments, hardware timestamping of all packets is also available on Solarflare NICs [2].

Methodology: Although tail-latency is a widely used system metric, there is no widely accepted experiment methodology for measuring it, and usually tools are bounded to specific methodologies. LANCET attempts to split the methodology from the actual tool and reason about them separately. Measurement bias from non-determinism can be avoided via setup randomization [21, 41, 58]. Repeated runs eliminate hysteresis effects in systems [58]. Distributed benchmarking tools seek to minimize client side queuing bias by reducing the client load, in asymmetric *e.g.,* Mutilate [33], or symmetric setups [58]. LANCET's use of hardware timestamping eliminates client bias in the point of measurement. Treadmill [58] avoids issues of imbalance by leveraging a symmetric measurement model, and bias from outliers by computing interested metrics on individual instances and combining them using aggregation functions. LANCET also supports the symmetric setup to detect imbalance across client machines. Most tools use histograms to capture latencies. Treadmill determines bucket ranges during a calibration phase. YCBS [10] and Tailbench [29] have dynamic range histograms. LANCET relies on on-line sampling but keeps all sampled results to determine both the CCDF and the confidence intervals. Confidence intervals can also be used to determine statistical convergence of results [21,41]. LANCET's self-correcting controller relies on statistical tests to ensure stability and results convergence similarly to [40].

8 Conclusion

LANCET is a new latency-measuring tool designed with the explicit goal to accurately measure µs-scale tail-latencies while reducing methodological pitfalls in a principled manner. Its self-correcting methodology uses proven statistical methods to detect situations where application tail latency cannot be reliably measured. LANCET's agents uniquely leverage NIC-based timestamping to measure the end-to-end latency of TCP-based applications, completely eliminating client bias. LANCET measures latency distributions with more accuracy than popular tools such as Mutilate and Treadmill. Our evaluation with µs-scale workloads shows that it robustly self-corrects as a function of the load for workloads with challenging service time distributions.

Acknowledgements

We would like to thank our shepherd Charlie Curtsinger, Vincent Gramoli, Guillaume Jourjon, and the anonymous reviewers for their valuable comments on the paper, Prodomos Kolyvakis for his insights on the experiment methodology, Mikael Gonzalez Morales for implementing hardware timestamping for R2P2, and Christos Kozyrakis for the early discussions on the topic. This work was funded in part by a VMWare grant and by the Microsoft Swiss Joint Research Centre. Marios Kogias is supported in part by an IBM PhD Fellowship.

References

[1] Mutilate codebase (commit d65c6ef). https://github.com/leverich/mutilate.

[2] SolarFlare networking interfaces. https://www.solarflare.com/.

[3] Treadmill codebase (commit 1bf2082). https://github.com/facebook/treadmill.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[5] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[6] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[7] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. RFC 2544 (Informational), March 1999. Updated by RFCs 6201, 6815.

[8] Cisco Systems. T-Rex: Cisco's realistic traffic generator. https://trex-tgn.cisco.com.

[9] CloudSuite Benchmarking Suite. http://cloudsuite.ch/.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, 2010.

[11] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[13] Christina Delimitrou and Christos Kozyrakis. Quality-of-Service-Aware Scheduling in Heterogeneous Data centers with Paragon. *IEEE Micro*, 34(3):17–30, 2014.

[14] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, pages 127–144, 2014.

[15] David Dickey and Wayne A Fuller. Likelihood ratio statistics for autoregressive time series with a unit root. *Econometrica*, 49(4):1057–72, 1981.

[16] Data plane development kit. http://www.dpdk.org/.

[17] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

[18] John Eidson and Kang Lee. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *Sensors for Industry Conference, 2002. 2nd ISA/IEEE.*

[19] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the 15th ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 275–287, 2015.

[20] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII)*, pages 37–48, 2012.

[21] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 57–76, 2007.

[22] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2015.

[23] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas F. Wenisch, Jason Mars, Lingjia Tang, and Ronald G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the 21st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 271–282, 2015.

[24] Ixia. Ixia traffic generator. https://www.ixiacom.com.

[25] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay F. Lofstead, Kathryn Mohror, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Popper Convention: Making Reproducible Systems Evaluation Practical. In *Proceedigns of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 1561–1570, 2017.

[26] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for µs-scale tail latency. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[27] Svilen Kanev, Kim M. Hazelwood, Gu-Yeon Wei, and David M. Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *Proceedings of the 2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 31–40, 2014.

[28] Harshad Kasture and Daniel Sánchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, pages 729–742, 2014.

[29] Harshad Kasture and Daniel Sánchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 3–12, 2016.

[30] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[31] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.

[32] Changhyun Lee, Chunjong Park, Keon Jang, Sue B. Moon, and Dongsu Han. Accurate Latency-based Congestion Feedback for Datacenters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 403–415, 2015.

[33] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the 2014 EuroSys Conference*, pages 4:1–4:14, 2014.

[34] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 2014 ACM Symposium on Cloud Computing (SOCC)*, pages 9:1–9:14, 2014.

[35] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[36] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 301–312, 2014.

[37] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 450–462, 2015.

[38] Stephen Mallon. tcp: Fix sof_timestamping_rx_hardware to use the latest timestamp during tcp coalescing. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=cadf9df27e7cf40e390e060a1c71bb86ecde798b, 2018.

[39] R. Mandeville and J. Perser. Benchmarking Methodology for LAN Switching Devices. RFC 2889 (Informational), August 2000.

[40] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, Robert Ricci, and Ana Klimovic. Taming Performance Variability. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, pages 409–425, 2018.

[41] David Meisner, Junjie Wu, and Thomas F. Wenisch. BigHouse: A simulation infrastructure for data center systems. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–45, 2012.

[42] Mellanox Corporation. ConnectX-4 NIC.

[43] Memaslap Load Generator. https://libmemcached.org/libMemcached.html.

[44] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily R. Blem, Hassan M. G. Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 537–550, 2015.

[45] Mutilate Load Generator. https://github.com/leverich/mutilate.

[46] Netperf. http://www.netperf.org/netperf/.

[47] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[48] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[49] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, 2016.

[50] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[51] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC)*, pages 342–355, 2015.

[52] Mia Primorac, Edouard Bugnion, and Katerina J. Argyraki. How to Measure the Killer Microsecond. In *Proceedings of the 2017 Workshop on Kernel-Bypass Networks (KBNETS@SIGCOMM)*, pages 37–42, 2017.

[53] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's Time for Low Latency. In *Proceedings of The 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2011.

[54] Spirent Communications. Spirent test modules and chassis. https://www.spirent.com/Products/TestCenter/Platforms/Modules.

[55] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, 2013.

[56] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[57] YCSB Load Generator. https://github.com/brianfrankcooper/YCSB.

[58] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 456–468, 2016.