A Study of Long-Tail Latency in n-Tier Systems: RPC vs. Asynchronous Invocations

Qingyang Wang[†], Chien-An Lai[*], Yasuhiko Kanemasa[‡], Shungeng Zhang[†], Calton Pu[*]

[†]*Computer Science and Engineering, Louisiana State University*
[*]*College of Computing, Georgia Institute of Technology*
[‡]*Software Laboratory, FUJITSU LABORATORIES LTD.*

***Abstract*—Long-tail latency of web-facing applications continues to be a serious problem. Most of the previously published research addresses two classes of long latency problems: uneven workloads such as web search, and resource saturation in single nodes. We describe an experimental study of a third class of long tail latency problems that are specific to distributed systems: Cross-Tier Queue Overflow (CTQO) due to a combination of millibottlenecks (with sub-second duration) and tightly-coupled servers in n-tier systems (e.g., Apache, Tomcat, and MySQL) using RPC-style request-response communications. Our experiments show that the appearance of millibottlenecks (e.g., created by short workload bursts) in one server often causes another server (which has no saturated resources) in the synchronous invocation chain to fill up its queues (CTQO) and drop packets, creating very long response time queries. CTQO can be reduced or avoided by replacing the server dropping packets with an asynchronous server. In synchronous n-tier system experiments, long tail latency due to CTQO can be reproduced consistently at utilization as low as 43%. In contrast, when all n-tier servers are replaced by asynchronous versions, CTQO and consequent dropped packets remain absent at utilization levels as high as 83%, despite the same millibottlenecks.**

I. Introduction

Long-tail latency problem in web-facing applications appears when a majority of normal queries (responding within milliseconds) co-exist with a non-trivial number of queries with very long response time (VLRT), on the order of seconds. Long-tail latency is an important practical problem: an Amazon study [17] showed that every 100ms increase in the page load time decreases sales by 1%. Thus it is desirable to remove *all* VLRT queries. However, despite several studies on various aspects of long-tail latency, practitioners continue to report real-world problems in recent years [9], [16], [19], [20], [34]. Many believe that the persistent low utilization levels in data centers [19], [22], [26] are also due to management concerns with long-tail latency.

The technical challenges in long-tail latency research arise from the varied causes of long-tail latency, which can be divided into three classes: (1) uneven resource requirements in apparently uniform workloads [13], e.g., web search of popular terms can return many more results than normal terms; (2) resource contention due to bursty workloads in single nodes [25], [29], [33], e.g., interference by "noisy neighbors"; (3) resource contention with dependencies among distributed nodes, a phenomenon we call *Cross-Tier Queue Overflow* (CTQO), which is the focus of this paper. A more detailed discussion of this classification is included in Section II. Here, we point out that VLRT requests due to CTQO are clearly unrelated to uneven job requirements, since the execution of these VLRT requests by themselves would only take milliseconds. Distinct from single node resource contention, VLRT requests due to CTQO often start to appear under moderate average resource utilization (e.g., 50%) of all participating nodes. Consequently, we believe that CTQO problems may arise independently even in networked systems where techniques proposed for the other two causes have been deployed. Our study is consistent with Mogul's argument [24] that the performance of distributed systems is far more complex than a single server's behavior due to the dependencies and interactions among components.

In this paper, we study a representative category of CTQO problem, the VLRT requests in n-tier systems caused by dropped packets due to strong cross-tier dependencies created by Remote Procedure Calls (RPC) among servers. (To the best of our knowledge, all the known CTQO problems fall into this category.) Detailed experimental data will demonstrate that the following sequence of causal events will lead to dropped packets and VLRT requests at moderate average utilization levels. (1) Resource millibottlenecks happen in some node, e.g., CPU saturated for a fraction of a second due to bursty workloads typical of web-facing applications. (2) A millibottleneck stops the saturated server processing for a short time (order of milliseconds), causing its message queues and thread pools to fill up, initiating a process called Cross-Tier Queue Overflow (CTQO) in which upstream or downstream servers also fill up their queues. (3) When one of the waiting servers (not the server with the originating millibottleneck) have its queues overflown (e.g., when all the threads are busy and TCP buffer overflows), further incoming packets are dropped. (4) The dropped packets are retransmitted several seconds later, creating VLRT requests. Our data will show that these events can reliably reproduce the VLRT requests when appropriate (and reasonable) conditions are met in a classic n-tier system with RPC-style synchronous invocations.

The main contribution of this paper is a methodical experimental evaluation of the effectiveness of event-based asynchronous servers in reducing or preventing CTQO, by removing the strong dependencies caused by RPCs. We replace synchronous servers in n-tier systems with their asynchronous counterparts one by one. First, we replace Apache with Nginx, an event-based web server. Second, we replace Tomcat with an event-based version called XTomcat. Third, we turn on a lightweight queue feature of the InnoDB storage server in MySQL to reduce queuing overhead. Detailed experimental data show that replacing an upstream synchronous server, e.g., Apache with Nginx, can remove the upstream CTQO problem. Symmetrically, replacing a downstream synchronous server, e.g., Tomcat with XTomcat, can remove the downstream CTQO problem. Under moderate resource utilization levels, the CTQO problem disappears completely if (and only if) all the servers are asynchronous.

The second contribution of the paper is a concrete characterization of the CTQO class of VLRT requests, a significant set of distributed phenomena in long-tail latency problems. The two components of CTQO both contain non-trivial classes of problems. First, CTQO is initiated by millibottlenecks that can arise from contention of any hardware or software resources, including CPU, memory, network, disk and other storage devices, and kernel management of these resources. We do not claim novelty of millibottleneck discovery, previously reported [31], [32], but we add to the variety of millitbottlenecks studies. Second, the cross-tier dependencies that cause VLRT requests bring back the fundamental question on the trade-offs between the syntactic simplicity of RPC-style synchronous invocations and scalability advantages of asynchronous communications at the cost of higher programming complexity. It would appear that after more than 30 years of RPC dominance [8], there may be good reasons for serious consideration of asynchronous communications again.

Due to space constraints we assume reader familiarity with n-tier systems and their main components, as well as web-facing applications such as e-commerce. We further assume readers are familiar with general distributed systems tradeoffs, specifically the RPC and asynchronous communications. We include appendices that summarize details on our system for the readers who may be less familiar with these concepts.

The rest of the paper is organized as follows. Section II summarizes the related work. Section III explains the long-tail latency problems due to dropped packets. Section IV describes the experiments that illustrate the sequence of causal events starting from millibottlenecks and ending in dropped packets because of CTQO. Section V shows the evaluation of n-tier configurations with an increasing number of asynchronous servers and Section VI concludes the paper.

II. Related Work

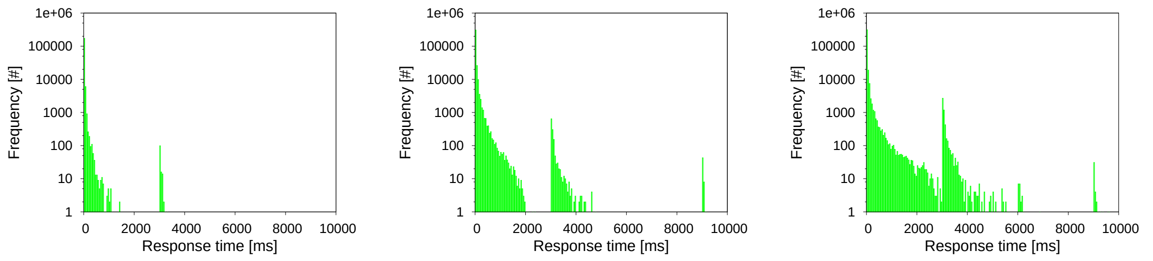
The long-tail latency problem of internet services in cloud has received considerable attention recently [6], [9], [19], [20], [34]. The sources of the tail latency can be divided into three classes: (1) Uneven resource requirements in apparently uniform workloads, (2) Resource contention due to bursty workloads or collocated workloads in single nodes, and (3) Resource contention with dependencies among distributed nodes.

On the first class, uneven resource requirements of workloads means most of requests take very short time to complete while a non-negligible percentage take long time to finish, causing the long-tail latency problem. There are two types of long requests.

Known causes for long requests. This usually happens in web search, while most user search queries are short, a significant percentage are long [13], [14]. Jeon et al. [14] report that the longest queries (e.g., 99th-percentile execution times) can be 10X longer than the average, and even 100X longer than the median. Reducing tail latency in this case is to reduce the long queries' execution time (e.g., through parallel processing). Our research on VLRT requests due to CTQO are unrelated to uneven job requirements, since the execution of these VLRT requests by themselves would only take milliseconds.

Unknown causes for long requests. This is mainly due to the performance variations of computing nodes, which make the processing of a normal request to be unexpectedly long. Several previous research efforts propose solutions to the unexpected long requests without identifying the specific sources. Representative examples include: Dean et al. [9] (use service replication to bypass tail latency in Google's interactive applications) and C3 [27] (adaptive replica selection scheme in storage servers). Their approach typically uses redundant servers to compensate for requests that are taking "too long" to respond, regardless of the cause of the delay. Our study differs from this class because of our focus on the specific class of long-tail latency problems due to dropped packets.

On the second class, resource contention due to bursty workloads or collocated workloads in single nodes cause significant queuing/scheduling delay, which lead to the long-tail latency problem. Typical solutions are



(a) WL 4000, system throughput is 572 req/s. Highest average CPU util. is 43%.

(b) WL 7000, system throughput is 990 req/s. Highest average CPU util. is 75%.

(c) WL 8000, system throughput is 1103 req/s. Highest average CPU util. is 85%.

Fig. 1: Semi-log graph of request frequency by response time at three representative workloads. The latency long-tail problem is due to dropped packets with multi-modal distribution, peaks near 0, 3, 6, and 9 seconds.

providing smart control mechanisms, resource allocation, and priority scheduling. Representative examples include: Cake [30] (reactive feedback-control scheduler to minimize the interference between batch jobs with latency sensitive applications), PriorityMeister [36] (combination of per-workload priorities and rate limit across multiple stages), and Detail [35] (prioritizing latency-sensitive flows to reduce network latency). Our focus on distributed phenomena (dropped packets and CTQO) and solutions (chain of asynchronous servers) differs from, and complements, these previous studies.

On the third class, previous research [31], [32] shows that millibottlenecks with dependencies among distributed nodes not only cause queuing delay in a local server, but also cause significant queueing delay in other servers in the invocation chain and lead to the long-tail latency problem. In these papers, solutions considered primarily the causes of millbottlenecks, for example, Java garbage collectors (GCs) [32] and Dynamic Voltage and Frequency Scaling (DVFS) algorithms in CPU power management [31]. Compared to these early studies, this paper focuses on asynchronous communications among distributed nodes as a general solution, regardless of the specific cause of millibottlenecks.

III. Long-Tail Latency Due to Dropped Packets

Our focus is on the CTQO class of long-tail latency problems caused by dropped packets under moderate average utilization. Such dropped packets form an interesting and challenging class of problems because of a combination of two apparently contradictory factors. First, unlike the skewed work requirement case [13], the VLRT requests only take milliseconds when executed by themselves. This suggests that queueing is a likely cause. Second, the VLRT requests start to appear at moderate resource utilization levels, for example, 50% of CPU for CPU-intensive workloads, where queuing would be unlikely according to classic queuing theory.

The CTQO class of long-tail latency problems is characterized by a distinctive multi-modal distribution of response times. Figure 1 shows that the vast majority of requests return within a few milliseconds. However, additional clusters starting at 3, 6, and 9 seconds demonstrate the existence of long-tail latency. Significant experimental evidence has been presented [31], [32] in support of the hypothesis that the 3, 6, and 9-second clusters in Figure 1 are due to the execution of retransmitted TCP packets.

Figure 1 also shows that packets start to drop at moderate resources utilization levels (e.g., 43% of CPU in Figure 1(a)). Thus we can rule out persistent resource bottlenecks as potential explanations. Instead, very short bottlenecks called *millibottlenecks* in this paper due to their sub-second duration have been linked to dropped packets. Based on an analysis of previous work and our experimental results (see next section), we summarize a set of static conditions for millibottlenecks to produce dropped packets and VLRT requests:

1) The n-tier system is composed of synchronous servers communicating through RPC-style invocations, e.g., Apache, Tomcat, and MySQL.
2) The workload is bursty [10], [21], [23], characteristic of web-facing applications, e.g., RUBBoS [4].
3) The requests are short, taking only milliseconds to execute. (This is a common situation in practice.)
4) All the servers operate at moderate average utilization levels for all resources. (This assumption removes the interferences from persistent bottlenecks and skewed work [13].)

In Section IV, we will show that under these static conditions (which are reasonable assumptions for web-facing applications) millibottlenecks may happen due to several possible reasons including a workload burst. A millibottleneck will trigger a sequence of events called *Cross-Tier Queue Overflow (CTQO)* that will lead to dropped packets and produce VLRT requests under the following appropriate dynamic conditions:

1) Reasonable workload (e.g., 1000 requests/sec).

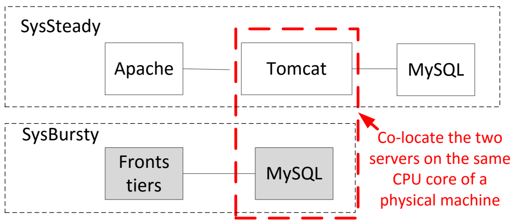

SysSteady
Apache
Tomcat
MySQL
SysBursty
Fronts tiers
MySQL
Co-locate the two servers on the same CPU core of a physical machine

Fig. 2: VM Consolidation: SysSteady-Tomcat co-located with SysBursty-MySQL.

2) Reasonable system configurations (e.g., thread pool size of 150, TCP buffer (backlog) size of 128).
3) Millibottleneck of sufficient length (e.g., 0.4sec).

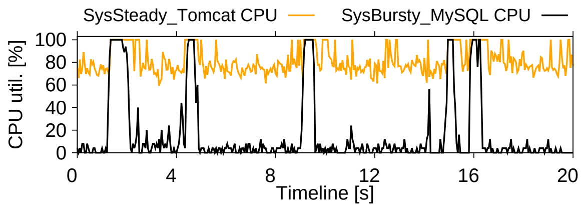
As an illustrative example, consider the sample values above: 1000*0.4=400 requests will arrive during the millibottneck, when the server can only hold 150+128=278 requests. The excess requests will cause the CTQO and dropped packets. Furthermore, these conditions are sufficient for reliably reproducing VLRT requests. We note that the static and dynamic conditions are independent of the specific causes of millibottlenecks. Concretely, Section IV describes the CTQO process for two kinds of millibottlenecks: one is in CPU and the other is in I/O.

IV. Millibottlenecks+CTQO= Dropped Packets

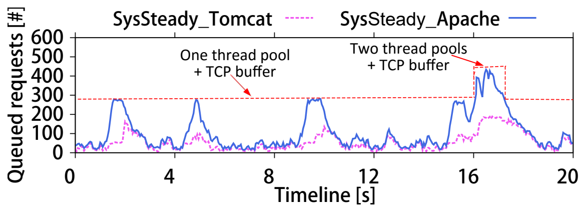
CTQO is initiated by millibottlenecks in a downstream tier server, from which the queueing effect is propagated and amplified to the upstream tiers of the system because of the synchronous RPC-style communication in the long invocation chain. We use a micro-level event analysis to show CTQO in our RUBBoS experiments, where millibottlenecks are observed in two typical scenarios (VM consolidation and server log flushing) in cloud computing environment. The micro-level event analysis exploits the fine-grained measurement data collected in RUBBoS experiments. Specifically, all the messages exchanged between servers are timestamped at millisecond resolution and system resource utilizations (e.g., CPU) are monitored at short time intervals (e.g., 50ms).

A. *CPU Millibottleneck: VM Consolidation*

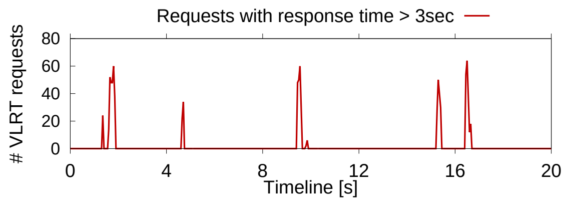
The first illustrative example of upstream CTQO is a CPU millibottleneck due to interferences between two consolidated VMs. VM consolidation is a common practice for cloud services to share infrastructure costs and increase profit [7], [12], [15]. The following VM consolidation experiments consist of two RUBBoS 3-tier applications, called SysSteady and SysBursty, respectively. For clarity of analysis, the consolidation consists of only one shared physical node (single core).



(a) SysSteady-Tomcat consumes about 70% of physical CPU core (yellow line). SysBursty-MySQL requires 100% of CPU during bursts (black line), saturating the physical node and pushing SysSteady-Tomcat into millibottleneck (yellow line reaching 100%) at time markers 2, 5, 9, and 15 seconds.



(b) During the millibottleneck periods (2, 5, 9, 15 seconds), queues are filled in SysSteady Tomcat and CTQO creates longer queues in SysSteady Apache.



(c) Number of VLRT requests (>3 sec) due to dropped packets at 2, 5, 9, and 15 seconds, counted at every 50ms time window. Such VLRT requests contribute to multi-modal response time distribution in Figure 1(b).

Fig. 3: Upstream CTQO due to millibottlenecks in VM Consolidation.

All other servers run on their own dedicated physical nodes. In most experiments SysSteady Tomcat is co-located with SysBursty MySQL, producing millibottlenecks in SysSteady Tomcat. This configuration is shown in Figure 2. For this section, SysSteady has a workload of 7000 clients with the RUBBoS default burst index of 1 (see [23]). SysBursty has a much smaller workload of 400 clients, but burst index of 100. Such bursts are an important part of web application workload characterization (e.g., the "Slashdot" effect [5]).

When both SysSteady and SysBursty are in a steady state, Figure 3(a) shows that SysSteady Tomcat consumes about 70% of the CPU and SysBursty MySQL consumes a negligible amount. As SysBursty enters a workload burst (at 2, 5, 9, and 16 seconds), combined workload saturates the shared CPU, causing millibottlenecks and the associated queueing effect in SysSteady Tomcat. The queues that store the waiting requests consist of a server's thread pool (size

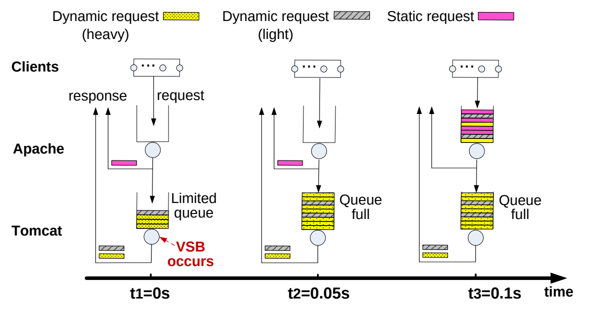


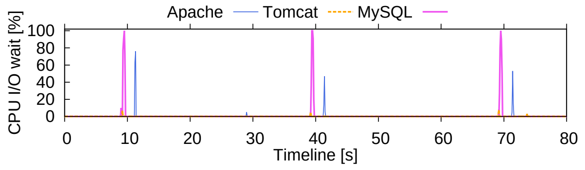
Fig. 4: Conceptual illustration of upstream CTQO in Apache-Tomcat. At t_1 time marker, a millibottleneck starts in Tomcat, filling up its queues. At t_2, MaxSysQDepth(Tomcat) is reached, forcing all types of requests to block in Apache. By t_3, even static requests which are only served in Apache are queued.

determined by its configuration parameter) and TCP buffer (TCP backlog). The total number (sum) of queue-able requests in all those queues is denoted by *MaxSysQDepth*. The queuing of requests in SysSteady Tomcat resulting from the millibottlenecks quickly exceeds MaxSysQDepth(Tomcat), blocking new comming requests. The blocking of Tomcat queues leads to even longer queues in SysSteady Apache because all types of requests start to queue there (Figure 3(b)). This "push-back" behavior is called upstream CTQO, as illustrated conceptually in Figure 4. For typical millibottlenecks, the growth of Apache queues in exceeds MaxSysQDepth(Apache) (278=150+128, indicated by the dashed line in Figure 3(b)), leading to dropped packets. The dropped TCP packets are retransmitted after 3 seconds in Redhat kernel 2.6.32, creating the VLRT requests shown in Figure 3(c).

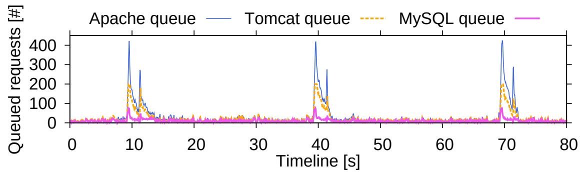
In addition to the first level Apache queue overflow, at MaxSysQDepth(Apache)$\approx$ 278 at 2, 5, 9, 15 seconds, there is a second level queue overflow at about 17 seconds in Figure 3(b), where MaxSysQDepth(Apache)$\approx$ 428. The increased MaxSysQDepth is from a new, second Apache process with an additional thread pool (150), which is in response to the full consumption of all the threads in the first process. However, incoming requests still get dropped when the thread pool in the second process is exhausted.

B. *I/O Millibottleneck: Log Flushing*

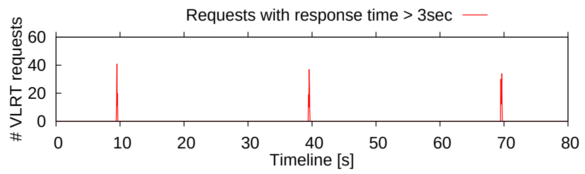
The second illustrative example of upstream CTQO consists of the flushing of logs by the `collectl` [1] performance monitoring tool used in our experiments. At fine granularity (every 50ms), `collectl` records system resource utilization, including CPU, memory, process runtime state, network and disk I/O. To remove the interference from CPU millibottlenecks in Tomcat,



(a) Log is flushed by `collect` every 30 seconds, causing CPU I/O wait millibottlenecks in MySQL.



(b) Millibottlenecks in MySQL create upstream CTQO to Tomcat, and finally to Apache. When MaxSysQDepth(Apache) is exceeded, new requests are dropped, becoming VLRT requests (see (c)).



(c) VLRT requests occur when MaxSysQDepth(Apache) is reached (see figure (b) above).

Fig. 5: Upstream CTQO due to I/O millibottlenecks in MySQL.

we scale up the Tomcat VM in these experiments from one CPU core to four cores. This upgrade increases the number of requests processed by Tomcat, shifting the millibottlenecks to MySQL when its `collectl` flushes measurement data log from memory to disk at 30 seconds intervals. The millibottleneck from log flush reliably produces upstream CTQO.

Figure 5(a) shows millibottlenecks in MySQL in pink high peaks at 10, 40, and 70 seconds (30-second intervals) as `collectl` flushes its log, resulting in 100% I/O wait. These millibottlenecks cause the other MySQL threads blocking for CPU. When queued requests exceed MaxSysQDepth(MySQL), the Tomcat starts to see growing queues due to upstream CTQO (Figure 5(b)). When queued requests exceed MaxSysQDepth(Tomcat), further upstream CTQO propagates to Apache. When queued requests in Apache exceed MaxSysQDepth(Apache), packets are dropped, creating VLRT requests (Figure 5(c)).

V. Evaluation of Asynchronous N-Tier Systems

A. *Evaluation Method*

The experiments in Section IV showed two cases of upstream CTQO, one due to CPU millibottlenecks in Tomcat, and the other due to I/O millibottlenecks

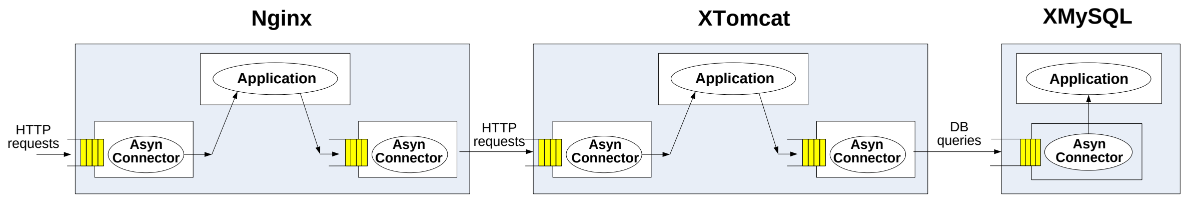


Fig. 6: Architecture of the asynchronous 3-tier system. Nginx, XTomcat, and XMySQL use asynchronous connectors to communicate with other servers.

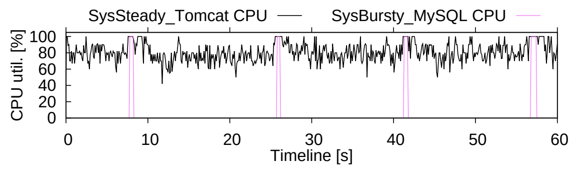
in MySQL. Since the dependencies due to the RPC-style communications are a necessary condition for upstream CTQO, a general question is whether we can reduce or prevent upstream CTQO by replacing the synchronous RPC calls with asynchronous event-based communications. We acknowledge that since its introduction [8], RPC has been considered *the right way* to build distributed systems, with n-tier applications as a classic example. Nevertheless, our experimental data will show that system designers should consider asynchronous communications when long-tail latency becomes a real problem due to dropped packets.

Starting from the same 3-tier application benchmark of Section IV, we will replace the three RPC-based servers (Apache, Tomcat, and MySQL) with their asynchronous counterparts one-by-one. All the experiments use the same workload to produce the same millibottlenecks, so we can study and compare the impact of asynchronous messages on the creation of CTQO. Sections V-B, V-C, and V-D describe the experimental results with an increasing number of asynchronous servers (NX=1, 2, and 3). The final configuration with three asynchronous servers is shown in Figure 6, with Nginx, XTomcat, and XMySQL. The limitations of some alternative design choices (e.g., increasing the thread pool size) are discussed in Section V-E. Due to space constraints, the implementation details of asynchronous servers and benchmark applications are included in Appendix A.

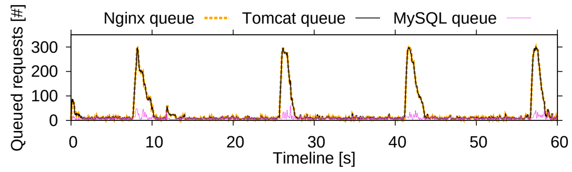
B. NX=1, Replacing Apache with Nginx

Since Apache is the source of VLRT requests in both cases described in Sections IV-A and IV-B, the first question to ask is whether replacing just the synchronous Apache with an asynchronous web server (Nginx [2], in wide production use for several years) will solve the upstream CTQO problem and remove VLRT requests. The answer turns out to be both yes and no. The Nginx indeed will not drop packets. However, the problem goes downstream to Tomcat and MySQL, which become new sources of VLRT requests.

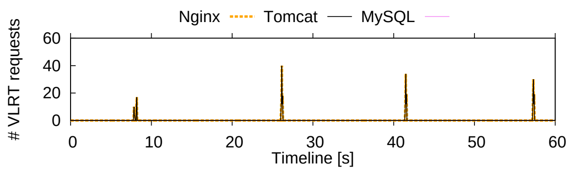
We make two technical observations in the following experiments. First, to better control our VM



(a) SysBursty-MySQL requires 100% of CPU during bursts (pink line), saturating the physical node and pushing SysSteady-Tomcat into millibottleneck (black line reaching 100%) at time markers 7, 26, 42, and 57 seconds.



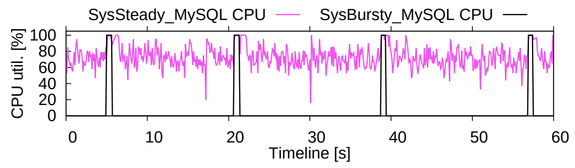
(b) Millibottlenecks in SysSteady Tomcat cause queued requests to reach MaxSysQDepth(Tomcat)=165+128=293; then packets start to drop, creating VLRT requests (see (c) below).



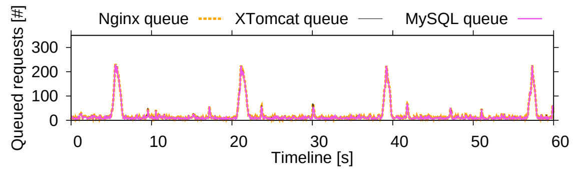
(c) VLRT requests observed in Tomcat during millibottlenecks in Tomcat itself.

Fig. 7: NX=1, Nginx-Tomcat-MySQL with millibottlenecks in Tomcat (a). No upstream CTQO in Nginx (b), but downstream CTQO happens when more packets than MaxSysQDepth(Tomcat) arrive during the millibottlenecks (c).

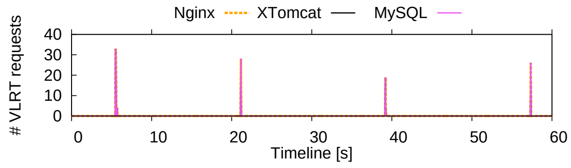
consolidation experiments (Section IV-A), we modified SysBursty to generate specific bursts of requests at specified times. For example, a batch of 400 ViewStory requests arriving every 15 seconds will create reproducible CPU millibottlenecks that last for approximately 300ms. Second, from the Cross-Tier Dependency sequence point of view, replacing the synchronous Apache with the asynchronous Nginx removes the limitations of MaxSysQDepth(Apache). This happens because the maximum number of requests processed by Nginx is no longer bound by MaxSysQDepth(Nginx), but a



(a) Millibottlenecks in SysSteady MySQL (pink line) caused by interference of the collocated SysBursty MySQL (black line) at time markers 6, 21, 39, and 57 seconds.

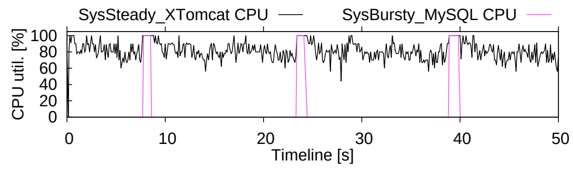


(b) Millibottlenecks in SysSteady MySQL cause queued requests to reach MaxSysQDepth(MySQL)=228=(100+128), then packets start to drop, creating VLRT requests (see (c)).

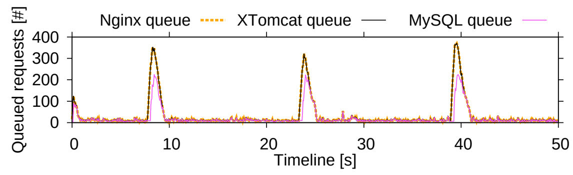


(c) VLRT requests observed in MySQL due to downstream CTQO and dropped packets.

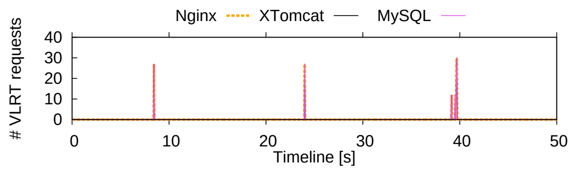
Fig. 8: NX=2, Nginx-XTomcat-MySQL with millibottlenecks in MySQL (a). No upstream CTQO from MySQL to XTomcat and Nginx (b). Downstream CTQO in MySQL (b) when packets arriving during millibottleneck exceed MaxSysQDepth(MySQL) (c).



(a) Millibottlenecks in SysSteady XTomcat (black line) caused by interference of the collocated SysBursty MySQL (pink line) at time markers 8, 24, and 39.



(b) Millibottlenecks in SysSteady XTomcat cause many queued requests in XTomcat (up to LiteQDepth) to be sent to MySQL in a batch, exceeding MaxSysQDepth(MySQL)=228=(100+128); excess packets are dropped.



(c) VLRT requests in MySQL due to downstream CTQO.

Fig. 9: NX=2, Nginx-XTomcat-MySQL with millibottlenecks in XTomcat. Millibottlenecks in XTomcat (a) cause batched requests (b) to be sent to MySQL, causing downstream CTQO and VRLT requests (c).

lightweight queue in Nginx of size LiteQDepth (e.g., all available TCP port numbers 65535). Nginx can route all the incoming requests to Tomcat, which shifts the problem downstream. Under the assumption of no resource bottlenecks in the web server, there are two potential sources of millibottlenecks downstream: Tomcat and MySQL, which are still synchronous.

The experimental results in Figure 7 show that millibottlenecks in SysSteady Tomcat can lead to dropped packets. Figure 7(a) shows the CPU utilization of SysSteady Tomcat and the co-located SysBursty MySQL (same configuration as Figure 2). The millibottlenecks in SysSteady Tomcat (at time mark 7, 26, 42, and 57) cause requests to queue in Tomcat (Figure 7(b)). Although the upstream CTQO between Tomcat and Apache has been removed by Nginx, Nginx will send a large number of requests (up to LiteQDepth(Nginx)) to Tomcat, which has MaxSysQDepth(Tomcat)=293 (thread pool size 165 plus the TCP buffer size 128). As the excess requests (LiteQDepth(Nginx) $>$ MaxSysQDepth(Tomcat)) overflow the queues in Tomcat (we call *downstream CTQO*), packets are dropped by Tomcat becoming VLRT requests (Figure 7(c)).

Millibottlenecks in SysSteady MySQL can be produced by bursts from a co-located SysBursty MySQL. These MySQL millibottlenecks can cause SysSteady Tomcat to drop packets due to upstream CTQO. Concretely, MySQL millibottlenecks cause queued requests in MySQL, which has MaxSysQDepth of about 50 (Tomcat DB connection pool size). When the requests exceed 50, Tomcat starts to fill up its own queues due to upstream CTQO. When the arriving requests exceed MaxSysQDepth(Tomcat), Tomcat starts to drop packets, causing VLRT requests. This upstream CTQO between MySQL & Tomcat is similar to the upstream CTQO between Apache & Tomcat (Figure 3). We omit the graphs in this case due to space constrains.

In summary, the Nginx-Tomcat-MySQL configurations show that the replacement of the synchronous Apache with the asynchronous Nginx removed the web server from the Cross-Tier Dependency sequence and prevented the upstream CTQO between Nginx and Tomcat. However, two problems may arise downstream. First, when millibottlenecks happen in Tomcat, the asynchronous Nginx can still send a large number of requests to Tomcat, causing downstream CTQO and dropped packets. Second, when millibottlenecks happen in MySQL, it may cause upstream CTQO in Tomcat.

C. NX=2, Replacing Tomcat with XTomcat

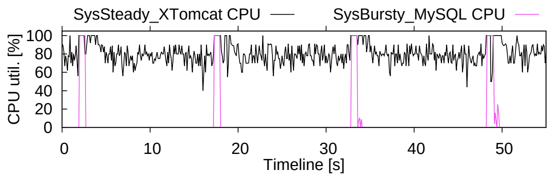
After the replacement of Apache with Nginx, the second step is to replace Tomcat with an event-based counterpart. Without a production-use asynchronous application server, we modified the Tomcat to handle requests asynchronously, which we call XTomcat. Since our focus is on the evaluation, the transformation of Tomcat into XTomcat is postponed to Appendix A.

After the yes-and-no answer to the first question (Section V-B), the second question to ask is whether replacing both the Apache and Tomcat with their asynchronous versions (Nginx and XTomcat) will solve the CTQO problem and remove VLRT requests. The answer again turns out to be both yes and no. The experiments show that upstream and downstream CTQO indeed cease to appear in both Nginx and XTomcat. However, downstream CTQO can occur in MySQL in two ways.

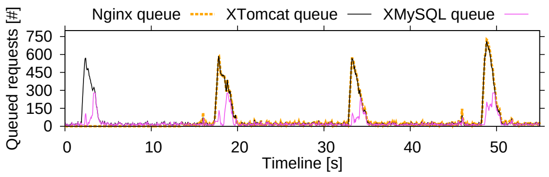
The first case of downstream CTQO happens with millibottlenecks in SysSteady MySQL, when co-located with SysBursty MySQL. Figure 8(a) shows the CPU utilization of SysSteady MySQL and SysBursty MySQL. The millibottlenecks in SysSteady MySQL appear at time marks 6, 21, 39, and 57, which create the queued requests in MySQL (Figure 8(b)). Since MaxSysQDepth(MySQL) 228 = thread pool size (100) + TCP buffer size (128), the continuous inflow of requests from Nginx and XTomcat cause dropped packets when more than 228 requests arrive during the MySQL millibottleneck, causing VLRT requests (Figure 8(c)).

The second case of downstream CTQO happens with millibottlenecks in SysSteady XTomcat, when co-located with SysBursty MySQL (Figure 9). The CPU usage of SysSteady XTomcat (Figure 9(a)) shows millibottlenecks in XTomcat at time marks 8, 24, and 39, causing queued requests (Figure 9(b)). When a XTomcat millibottleneck ends, XTomcat quickly sends all the queued requests to SysSteady MySQL (Figure 9(b)), up to LiteQDepth(XTomcat). The requests beyond the capability of MySQL plus MaxSysQDepth(MySQL) are dropped, becoming VLRT requests.

Although surprising at first glance, the downstream CTQO caused by XTomcat in MySQL can happen in realistic settings. Consider our illustrative example of millibottleneck with 0.4 sec duration and average request arrival rate of 1000 req/s. Since LiteQDepth(XTomcat) is large (e.g., 65535), XTomcat will store all 400 requests during its millibottleneck, and send all of them to MySQL after the millibottleneck ends. This will cause downstream CTQO and dropped packets when MaxSysQDepth(MySQL) of 228 is exceeded (Figure 9(c)).



(a) CPU millibottlenecks in SysSteady XTomcat caused by interference of the collocated SysBursty MySQL.



(b) Millibottlenecks in SysSteady XTomcat cause queued requests in XTomcat; Nginx queues and XTomcat queues indicate no CTQO in Nginx and XTomcat. Nginx and XTomcat also do not drop any requests since LiteQDepth is large (e.g., 65535).

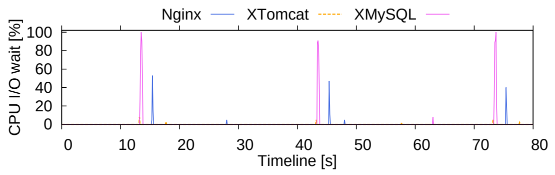
Fig. 10: NX=3, the Nginx-XTomcat-XMySQL case with millibottlenecks in XTomcat. CTQO disappears in the asynchronous 3-tier system.

D. NX=3, Replacing MySQL with XMySQL

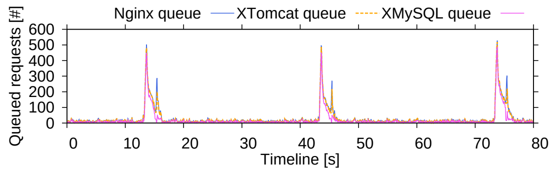
The third step of our evaluation is to replace MySQL with an asynchronous database server, XMySQL. The third question is whether replacing the entire n-tier system components with Nginx, XTomcat, and XMySQL will solve the CTQO problem and remove VLRT requests. The experiments will show that the entirely asynchronous n-tier system can prevent CTQO and avoid VLRT requests despite millibottlenecks in any of the servers. For XMySQL, we chose the InnoDB storage engine of MySQL that provides a feature to store the waiting requests in a lightweight queue when the threads are fully occupied. We configured the InnoDB in XMySQL to run with 8 threads to handle active queries, and an additional queue to store up to 2000 queries, which is sufficiently large for LiteQDepth(XMySQL).

To evaluate the entirely asynchronous n-tier system due to CPU millibottlenecks, we run the same VM consolidation experiments as described in Section IV-A and Figure 2). The component servers are changed from the synchronous version (Apache-Tomcat-MySQL) to their asynchronous counterparts (Nginx-XTomcat-XMySQL) in SysSteady, and the RUBBoS benchmark application changed to use asynchronous messages.

First, let us consider the millibottlenecks in XTomcat. Figure 10(a) shows CPU millibottlenecks in SysSteady XTomcat at time marks 4, 13, and 35, causing queued requests in XTomcat (Figure 10(b)). The graph also shows that the queues in Nginx have very similar depths of XTomcat queue during the millibottlenecks, suggesting no upstream CTQO between XTomcat and Nginx. Since Nginx, XTomcat, and XMySQL all have



(a) Millibottlenecks in MySQL caused by background process (collectl) log flushing every 30 seconds.



(b) All three asynchronous servers storing all requests in lightweight queues during millibottlenecks in MySQL.

Fig. 11: NX=3, the Nginx-XTomcat-XMySQL case with millibottlenecks in XMySQL. No CTQO or dropped packets in servers during I/O millibottlenecks.

very large LiteQDepth (e.g., 65535 and 2000), they can avoid downstream CTQO. Without either the upstream or downstream CTQO, there is no dropped packets.

For the case of millibottlenecks in XMySQL, we run the same Log Flushing experiments as described in Section IV-B, changing only the servers and the RUBBoS benchmark application from the synchronous version (Apache-Tomcat-MySQL) to their asynchronous counterparts (Nginx-XTomcat-XMySQL).

Figure 11(a) shows the CPU I/O wait of all the three tiers, with millibottlenecks in XMySQL at 30-second time intervals (time marks 13, 43, 73). Figure 11(b) shows similar request queue depths for XMySQL, XTomcat and Nginx, suggesting no upstream CTQO between all tiers. Similar to the VM consolidation experiments, large LiteQDepth(Nginx), LiteQDepth(XTomcat), and LiteQDepth(XMySQL) prevent downstream CTQO, resulting in no dropped packets.

E. Discussion of Alternative Designs

From a "RPC purist" point of view, an easy design alternative for the synchronous servers is to simply increase the MaxSysQDepth, e.g., by increasing the number of threads. In principle, this increase might be able to postpone or prevent the CTQO problems described in Section IV. Unfortunately, increasing threads number to thousands (comparable to LiteQDepth) introduces many problems discussed extensively in previous research [18], [25], [33]. Specifically, overhead due to high concurrency can come from various system layers including last level cache miss, high context switches, and scheduling overhead; for Java based servers, high workload concurrency also leads to non-linearly increased JVM garbage collection time due to more memory used by high number of threads.

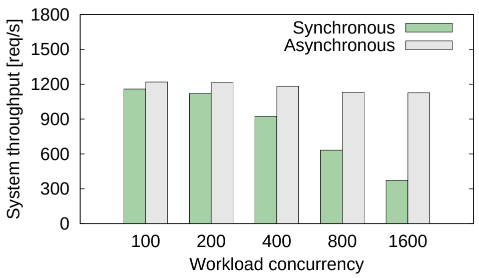


Fig. 12: Throughput comparison of the 3-tier system with the two different architectures. The asynchronous 3-tier system (Nginx-XTomcat-XMySQL) outperforms the synchronous one (Apache-Tomcat-MySQL) in system throughput at high concurrency.

As an example, Figure 12 shows that the throughput of the 2000-thread system (2000 thread pool size in Apache, Tomcat, and MySQL) decreases significantly as we increase the number of concurrent requests from 100 to 1600. Specifically, the throughput decreases from 1159 req/s at 100 concurrent requests to 374 req/s at 1600 concurrent requests. The decreased throughput is primarily due to the Tomcat CPU saturation, where the overhead of thread management increases with the threads number.

The second component of MaxSysQDepth is the TCP buffer (128 in Linux kernel). The TCP buffer size has been considered fixed by the networking community, since enlarging network buffer sizes has been shown to cause problems such as bufferbloat [11], leading to long delivery latency.

VI. Conclusion

In this paper, we described a methodical experimental study of one class of long tail latency problems that arise in distributed systems with tightly-coupled servers using RPC-style request-response communications: Cross-Tier Queue Overflow (CTQO). We have found two main cases of CTQO in our study: *upstream* CTQO, when an upstream server is dropping packets because of a downstream server suffering from millibottlenecks; and *downstream* CTQO, when a downstream server is dropping packets because of upstream (or interacting) server millibottlenecks. In both cases, CTQO can be avoided by replacing the server dropping packets with an asynchronous server. Furthermore, by replacing all servers with their asynchronous versions we can remove both upstream & downstream CTQO despite workload bursts in moderate average resource utilization. This study suggests that long-tail latency (due to CTQO) may be reduced by the adoption of asynchronous inter-tier communications for the entire n-tier system.

Software Stack

PRC Web Server	Apache 2.2.22 + tomcat-connectors-1.2.28
Asyn Web Server	Nginx-1.6.2
RPC App Server	Tomcat 7.0.57 + BIO connector + mysql-connector-java-5.1.19
Asyn App Server	Tomcat 7.0.57 + NIO connector + async-mysql-connector-1.0
Database server	MySQL 5.5.19
Operating system	RHEL 6.3 (kernel 2.6.32)
Hypervisor	VMware ESXi v5.0

(a) Software setup

ESXi Host Configuration

Model	Dell Power Edge T410
CPU	2* Intel Xeon E5607, 2.26GHz Quad-Core
Memory	16GB
Storage	7200rpm SATA local disk

VM Configuration

Type	# vCPU	CPU limit	CPU shares	vRAM	vDisk
Small (S)	1	2.26GHz	Normal	2GB	20GB

(b) ESXi host and VM setup

Fig. 13: Details of the experimental setup.

VII. Acknowledgement

This research has been partially funded by National Science Foundation by CISE's CNS (1566443, 1421561), SAVI/RCN (1402266, 1550379), CRISP (1541074), SaTC (1564097) programs, an REU supplement (1545173), Louisiana Board of Regents under grant LEQSF(2015-18)-RD-A-11, and gifts, grants, or contracts from Fujitsu, HP, Intel, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

Appendix A
Experimental Setup and Benchmark Applications

We run the RUBBoS benchmark [4] on our virtualized testbed. Figure 13 outlines the software components, ESXi host and virtual machine (VM) configuration. Our RUBBoS adopts the typical 1/1/1 topology consisting of one Apache server, one Tomcat application server, and one MySQL database server. Each server runs on top of one VM deployed on one dedicated ESXi host.

In our implementation of an asynchronous 3-tier system as shown in Figure 6, we replace the previous thread-based servers with the corresponding asynchronous servers except the database tier. Concretely, we use Nginx to replace Apache as an asynchronous web server. XTomcat is the latest version of Tomcat (version 7 at the time) which embeds an asynchronous connector (Tomcat NIO Connector) for upstream communication. We also modified an open source asynchronous JDBC driver [3] for XTomcat to support asynchronous communication with MySQL. For MySQL, we use the InnoDB storage engine of MySQL that supports a lightweight queue (with capacity over 2000) when a few allocated threads are fully occupied, achieving the same effects of an asynchronous XMySQL.

```
function doGet(request1) {
    ... pre-processing request1 ...
    ... form query1 ...
    result1=SyncDBQuery1(query1);
    ... think about result1 ...
    ... form query2 ...
    result2=SyncDBQuery2(query2);
    ... post-processing result2 ...
    ... form response ...
    return response;
}
```

(a) A simple synchronous Java Servlet

```
function doGet(request1) {
     ... pre-processing request1 ...
     ... form query1 ...
     AsynDBQuery1(query1, eventHandler1);
}
function eventHandler1(result1) {
    ...think about result1 ...
    ...form query2 ...
    AsynDBQuery2(query2, eventHandler2);
}
function eventHandler2(result2) {
     ...post-processing result2...
     ...form response ...
     return response;
}
```

(b) An asynchronous event-driven Java servlet

Fig. 14: Simple RPC transformed into a set of asynchronous calls.

Asynchronous benchmark applications: Most existing n-tier benchmark applications assume synchronous calls for inter-tier communication (e.g., RPC-style database queries). To utilize the asynchronous connectors of each server for inter-tier communication, we need to re-implement the original benchmark application using the event-driven programming model.

Event-driven programming implements the processing of each request as a set of event handlers in which transitions between handlers are triggered by events. For instance, calling an asynchronous database query does not block a XTomcat processing thread from continuing to execute the program; the "return" of the asynchronous database query will be processed in a different module or handler (i.e., a callback function). Figure 14 illustrates how a simple synchronous Java servlet can be transformed into a functionally equivalent event-driven version. Each synchronous database query splits into two halves. The first half is the asynchronous version of $SyncDBQueryN$, which we refer to as $AsynDBQueryN$, and the second half, $eventHandler_n$, is a callback function triggered only by the returned result of $AsynDBQueryN$. Applying this methodology to every synchronous function call, we obtain the asynchronous version shown in Figure 14.

Synchronous inter-tier communications involved in other common control flows such as if-statement or for-loop can also be transformed into their corresponding asynchronous version. Thibaud Schneider [28] presents a set of transformation rules that allows us to take a program that uses synchronous function calls in arbitrary sequential control flow, and transform it into a program that calls the functions asynchronously. Using Schneider's rules, we transformed the RUBBoS application into the functionally equivalent asynchronous version.

References

[1] Collectl. "http://collectl.sourceforge.net/".

[2] NGINX. "http://nginx.org/".

[3] Non-Blocking (asynchronous) MySQL Connector for Java. "https://code.google.com/archive/p/async-mysql-connector/".

[4] RUBBoS: Bulletin board benchmark. "http://jmob.ow2.org/rubbos.html".

[5] S. Adler. The slashdot effect: an analysis of three internet publications. *Linux Gazette*, 38:2, 1999.

[6] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *NSDI'12*.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP'03*.

[8] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, Feb. 1984.

[9] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[10] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14:1–14:26, Nov. 2012.

[11] J. Gettys and K. Nichols. Bufferbloat: dark buffers in the internet. *Communications of the ACM*, 55(1):57–65, 2012.

[12] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *SoCC'11)*.

[13] M. Jeon, Y. He, H. Kim, S. Elnikety, S. Rixner, and A. L. Cox. Tpc: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *ASPLOS'16*.

[14] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: Taming tail latencies in web search. In *SIGIR '14*.

[15] Y. Kanemasa, Q. Wang, J. Li, M. Matsubara, and C. Pu. Revisiting performance interference among consolidated n-tier applications: Sharing is better than isolation. In *SCC'13*.

[16] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *SoCC'12)*.

[17] R. Kohavi and R. Longbotham. Online experiments: Lessons learned. *Computer*, 40(9):103–105, 2007.

[18] M. N. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *USENIX ATC'07*.

[19] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *EuroSys'14*.

[20] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *SOCC '14*.

[21] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *ICAC'10*.

[22] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Improving resource efficiency at scale with heracles. *ACM Transactions on Computer Systems (TOCS)*, 34:6:1–6:33, 2016.

[23] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting realistic burstiness to a traditional client-server benchmark. In *ICAC'09*.

[24] J. C. Mogul. Emergent (mis) behavior vs. complex software systems. *ACM SIGOPS Operating Systems Review*, 40(4):293–304, 2006.

[25] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton. Comparing the performance of web server architectures. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 231–243, 2007.

[26] B. Snyder. Server virtualization has stalled, despite the hype. *InfoWorld*, December 2010.

[27] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *NSDI'15*.

[28] Thibaud Lopez Schneider. Writing Effective Asynchronous XmlHttpRequests. "http://www.thibaudlopez.net/xhr/Writing%20effective%20asynchronous%20XmlHttpRequests.pdf".

[29] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS IX*, 2003.

[30] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling high-level slos on shared storage systems. In *SoCC'12*.

[31] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *ICDCS'13*.

[32] Q. Wang, Y. Kanemasa, J. Li, C.-A. Lai, C.-A. Cho, Y. Nomura, and C. Pu. Lightning in the cloud: A study of very short bottlenecks on n-tier web application performance. In *TRIOS'14*.

[33] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP'01*.

[34] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *NSDI'13*.

[35] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: Reducing the flow completion time tail in datacenter networks. *SIGCOMM Comput. Commun. Rev.*, 42(4):139–150, Aug. 2012.

[36] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *SOCC'14*.