A Study of Fine-Grained Bottlenecks in a Distributed System

Revised Project Proposal by Raghav Bhat

Spring 2020, Intro to Enterprise Computing

Overview & Motivation

For many web servers, the vast majority of requests take 10s of milliseconds [1]. However, when response times are plotted, one will notice that a small subset of requests take significantly longer—on the order of seconds [1]. As it turns out, the distribution of number of requests vs. response time has a long-tail [e.g. Figure 1]—a very high concentration of requests is returned very quickly, but a few requests take slightly longer, and far fewer take substantially longer (i.e. they rapidly taper off). Note that the longer requests are not inherently slower (e.g. more complex queries)—the response time is an artifact of the system, even at low resource utilization rates. Although it may come as no surprise that some requests take longer than others, what is, perhaps, surprising in this case is that the difference in response time can be several orders of magnitude.

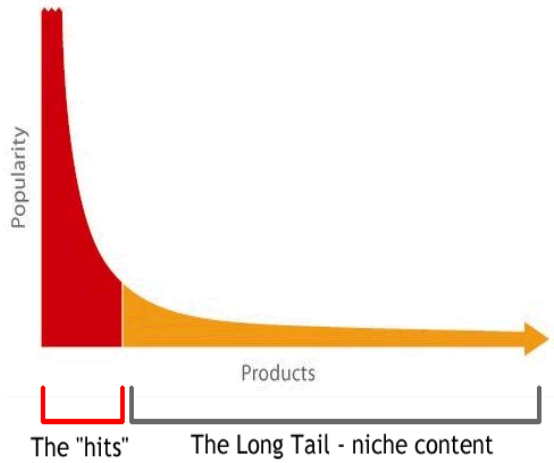

Figure 1

Due to the *very* long nature of the tail, it is problematic. According to an Amazon study, a 1% loss in sales was tied to every 100 milliseconds increase in page load time [1]. Google found a similar result, noting that if search took 500 milliseconds longer, the company stood to lose up to 20% of its revenue [1]. Thus, there is a strong financial incentive to tackle the latency long tail problem, and yet, it is not on the vast majority of developers' minds. This is likely because we know little about what causes these latency long tails and how to solve them.

According to Pu et. al, the long tail is caused by "millibottlenecks" [1]. The idea is that a milliseconds-long bottleneck in one part of an n-tier system will propagate through the entire system. Thus, this split-second bottleneck's effects are enlarged, and can cause queues in other systems to overflow, for example [1]. This theory has been substantiated by running a standard benchmark (RUBBoS) and logging response times, as well as resource utilizations, at the millisecond level [1]. After correlating different events, the seminal paper [1] shows that the JVM garbage collector can cause millibottlenecks.

However, there are several challenges in detecting and triaging these millibottlenecks. Firstly, resource monitoring must be very fine-grained, capable of detecting millibottlenecks of just 100 milliseconds. Most resource monitoring tools work at a far coarser granularity (seconds) and thus won't help detect and correlate millibottlenecks due to the sampling theorem. Secondly, the monitoring tools need to have very low overhead, so as to not introduce noise. This is especially true since we might need to simultaneously monitor several resources (e.g. CPU, memory, queue sizes). Finally, finding resource

millibottlenecks simply illustrates their effect; we must log the system events that are their cause, with the hope of correlating cause and effect.

One way to correlate, and therefore better understand, millibottlenecks with their cause is to run system experiments. System experiments will allow us to observe several millibottlenecks and gather the data to determine why they are happening. Eventually, this knowledge can be helpful in beginning to solve them. An experimental, as opposed to theoretical, analysis approach is best suited since we are only beginning to understand the effects of millibottlenecks and hardly have a grasp of the dependencies that might cause them.

Toolkit & Benchmark

We now detail the experimental framework. Specifically, we discuss the toolkit and the benchmark. The experiments are designed to be portable, so we can test the effects of different hardware and software.

Elba Toolkit

There are many challenges related to studying millibottlenecks. Firstly, the scientific process requires that the experiments are reproducible. Since millibottlenecks are system-level, all the dependencies the system has *must* be captured—for example, the specific hardware configuration and cache sizes must be known [5]. Secondly, there is the issue of data granularity. Most system monitoring tools monitor at the second or minute level. As discussed in the previous section, this is not sufficient for us. Finally, we must be able to reconstruct the execution path of requests in a distributed environment. This means we must have precise logging of events on every node in our system [5].

In order to meet these requirements, researchers at Georgia Tech have developed the Elba toolkit. The resource and event monitoring component of Elba is called milliScope. The resource monitoring component of milliScope uses open source tools such as *sar*, *Collectl*, and *iostat* to monitor the CPU, memory, network, disk, etc. at a fine granularity, with low overhead [3]. On the other hand, the event monitoring component of milliScope specializes the software components, retooling the native log components to log the execution boundaries of events (i.e. when the response goes in and out of the component).

In order to meet dependency requirements, Elba uses a workflow language called WED-Make to captures explicit and implicit dependencies [5]. With WED-Make, one can specify a complex benchmark, as is illustrated in paper [3].

WISE Benchmark

In order to conduct experiments, we must have a workload to run. The second-generation Elba toolkit, code-named WISE, includes a twitter-like web application that we use. Simulated tasks include creating and endorsing posts, subscribing to other users, viewing posts, and viewing a user's own inbox [6]. Following current industry trends in architecture, the application is microservices-based. As we can see in Figure 2, which details the architecture, the client sends an HTTP request to an Apache server, which then relays requests to the appropriate Thrift microservice [6].

Each of the five microservices connected to the Apache server handles a different aspect of the workload. Auth handles user registration and authentication, and sub manages user subscriptions; microblog provides core functions, such as creating posts, while queue is used to offload tasks that can be handled asynchronously; inbox allows the user to push and pull to his or her inbox. Stateful data is

stored in a PostgreSQL database, and there is a worker node that asynchronously processes requests relegated to the queue. [6]

The workload itself, which consists of one or more clients simulating user requests, is highly configurable. Configuration parameters include how many clients are sending requests and for how long [6].

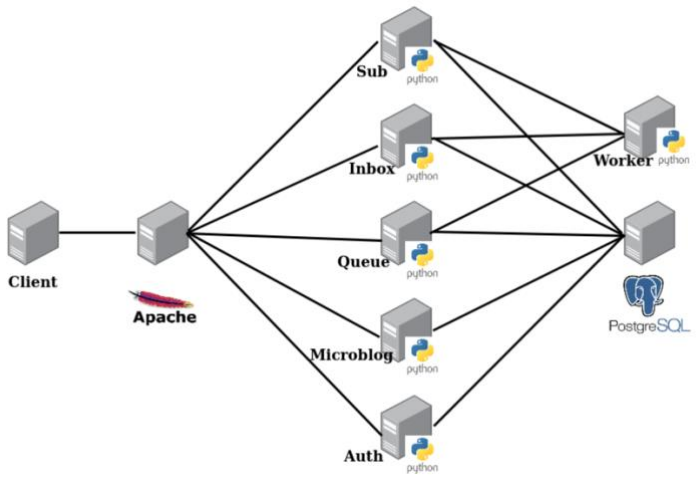

Figure 2 [6]

Goals, Implementation, & Evaluation Plan

We now discuss the experimental workflow that we will use in the two phases of our experiments, described below, to produce results. Figure 3 depicts the overall workflow of running the experiments, which is largely automated.

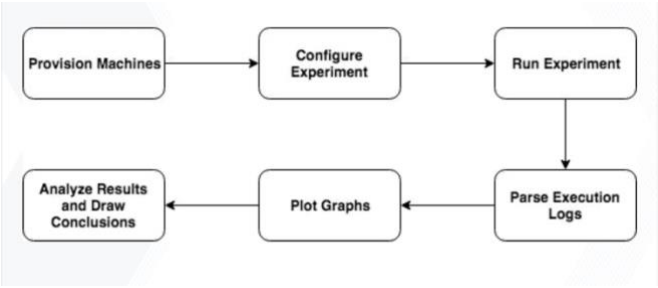

Figure 3 [5]

Provisioning Machines for Experiments, (Both Phases)

For both phases of the experiment, CloudLab [4] will be used to provision the necessary machines (step 1 in Figure 3). CloudLab is a public cloud for researchers and scientists, allowing us to provision to geographically dispersed machines with several different kinds of hardware and software.

Configuring Experiments, Phase One

Workload

In phase one, we will experimentally reproduce the results of paper [1] to verify the millibottleneck theory. This will help us understand the overall process of running and analyzing the experiments, thus setting us up for the more original work done in phase two.

For phase one, we will follow the tutorial in [6], which provides a guide to running each aspect of the experiment, as seen in Figure 3. Each service in Figure 2 will get its own host machine, and one host will be used to simulate the client for a total of 10 hosts.

Configuring Experiments

Configuration is the second step in Figure 3. For phase one, this process is largely automated—the experiments are contained in a WED-Makefile, a file that contains all the dependency and configuration parameters in a bash-like language. See Figure 4 for a sample WED-Makefile. By running the WED-Makefile, the event-monitoring and resource monitoring systems will both be installed and configured, after which the application configurations are set (e.g. thread count on Apache server), and the workload is defined [5]. We will use the default configurations for this phase.

Initial State: Experiment Configuration (Apache Server)

```
# Apache HTTP server configuration
# Reference: http://httpd.apache.org/docs/2.2/mod/core.html
# Reference: http://httpd.apache.org/docs/2.2/mod/mpm_common.html
readonly WEB_HTTPD_VERSION="2.2.22"
readonly WEB_HTTPD_TIMEOUT="5"
readonly WEB_HTTPD_KEEPALIVE="off"
readonly WEB_HTTPD_MAXKEEPALIVEREQUESTS="-"
readonly WEB_HTTPD_KEEPALIVETIMEOUT="-"
readonly WEB_HTTPD_MULTIPROCESSINGMODE="worker"
readonly WEB_HTTPD_SERVERLIMIT="200"
readonly WEB_HTTPD_THREADLIMIT="300"
readonly WEB_HTTPD_STARTSERVERS="1"
readonly WEB_HTTPD_MAXCLIENTS="300"
readonly WEB_HTTPD_MINSPARETHREADS="5"
readonly WEB_HTTPD_MAXSPARETHREADS="50"
readonly WEB_HTTPD_THREADSPERCHILD="150"
readonly WEB_HTTPD_MAXREQUESTSPERCHILD="0"
readonly WEB_HTTPD_LOGRESPONSETIME="on"
```

Figure 4 [5]

Configuring Experiments, Phase Two

Workload

In phase two, we will design experiments to observe the noisy-neighbor phenomenon when running the WISE benchmark on different system topologies. The noisy-neighbor phenomenon occurs when different virtual machines on the same host interfere with each other—for example, if another virtual machine on the same host suddenly has a burst in workload, the performance of your virtual machine will be impacted [7]. The noisy-neighbor problem was first published in [7] using an older version of the Elba toolkit and a different benchmark, RUBBoS[1]. Since we are running a different, microservice-based benchmark on a more mature version of Elba, our experiments will be materially different from those already published. Thus, our experiments serve the purpose of validating the noisy neighbor phenomenon on a new workload.

Configuring Experiments

In order to observe the noisy neighbor phenomenon, we will run multiple microservices within our WISE application on the same host machine. It is important to note that each service will be running in its own virtual machine, but multiple virtual machines will be configured to share the same host. For example, we might choose to locate the virtual machines running the inbox and microblog services on the same host. While we collocate different services on one host, we will also emulate noisy neighbors by running a stress-testing tool on that same host. The stress-testing tool could be, say, a CPU-intensive workload that helps show how high CPU usage by other tenants of a host impacts our service.

We will play around with different combinations of services and stress-testing tools. Our goal is to observe the noisy neighbor phenomenon by determining that the root cause of our system's very long response times is two or more virtual machines on the same host interfering with each other.

To start out with, we will run the following two experiments:

1. Baseline—follow Appendix A in [6] to place all the microservices across 3 different host machines using the default mapping of service to node. No benchmarking tools will be used on any of the nodes.
2. Disk-Heavy Neighbor to Database—Use the baseline configuration, adding a virtual machine with a disk-intensive stress-testing workload to the host running the database.

[1] The RUBBoS application benchmark is a four-tier system that simulates user requests to a Reddit-like website [3].

As we begin to run more experiments, we may also vary the experimental configurations of each service. Figure 4 depicts some of the configuration parameters that must be set for each service in our system. They include parameters such as the number of server processes, threads, and connection pool sizes. This can add another layer of complexity to our experiments.

OpenStack on CloudLab

In order to run multiple virtual machines on the same host, we will use OpenStack [8] to emulate a cloud-like environment. OpenStack can manage the creation of virtual machines and will therefore ease our experimental setup. Additionally, CloudLab can automatically installs and configures OpenStack on its physical hosts [6].

Stress-Testing Tool

We will create different stress-testing workloads using the Linux utility stress-ng [9]. This tool allows us to write scripts that hog one or more machine resources (e.g. memory, CPU, network I/O). After writing the stress-testing workload, we will run them on their own virtual machine.

Running, Processing, & Plotting Experiment Results, (Both Phases)

After an experiment is setup, we can run the experiments a number of times and save the results. Once the experiments are complete, we must parse and plot the experimental logs. This functionality is also provided by the toolkit.

Analyzing Results, (Both Phases)

Finally, we must analyze the results. Essentially, what we are looking for is a correlation between system resources and very long response times. For example, we might plot a response time vs. time graph, alongside a queue size vs. time graph. If we see a correlation (e.g. a spike in both at the same time), we can deduce that the long response time was likely due to flooding in the queue (and hence dropped packets). We might then examine another graph, such as a CPU utilization vs time graph, to try and determine what the system was doing to cause a bottleneck that eventually resulted in the queue flooding (e.g. garbage collection) [1].

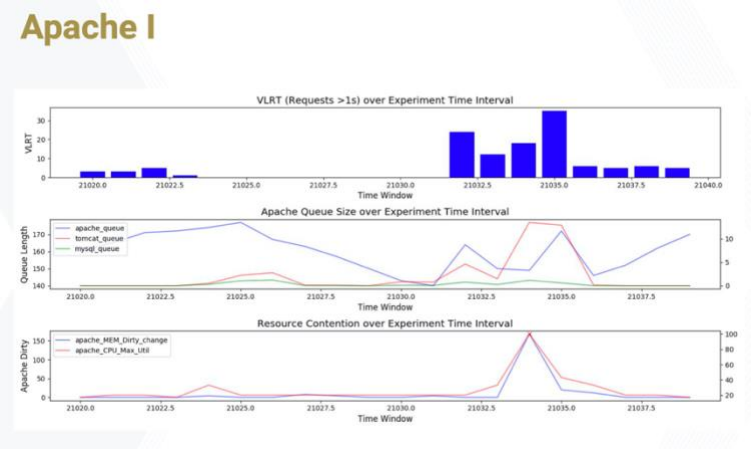

Figure 5 [5]

Figure 5 depicts a real-world analysis; we see that the number of very-long-response-time (VLRT) requests is correlated with a spike in queue size in the Apache server. This spike in queue size is a result of resource contention [5].

The analysis will require a lot of trial-and-error, as well as a deep understanding of the system and its various components. By reproducing the JVM bug [1] from phase one, I will gain an understanding of how this analysis is done. Then, in phase 2, I will seek to do this analysis on my own in an effort to determine a novel millibottlenecks.

Biweekly Milestones

For now, only the first and second milestone are clearly defined—in this phase, I will follow the tutorial by Rodrigo to reproduce the results of [1] and verify its conclusion. At the same time, I will work with Rodrigo to better define the novel experiments I will run for the 3rd and 4th milestones.

1st (due 02/28): Set up a CloudLab account and learn how to run a simple experiment following the tutorial.

2nd (due 03/13): Use the toolkit parsers to process system log data (performance and event) from the experiments, plot graphs, and analyze the results

3rd (due 03/27): Run a specific set of experiments to be defined with TA.

4th (due 04/10): Process and analyze the data resulting from this set of experiments.

References

[1] The Millibottleneck Theory of Performance Bugs, and Its Experimental Verification (Calton Pu et al.)
[2] The Tail at Scale (Jeffrey Dean and Luiz André Barroso)
[3] Systematic Construction, Execution, and Reproduction of Complex Performance Benchmarks (Rodrigo A. Lima et al.)
[4] CloudLab, https://www.cloudlab.us/.
[5] Elba Std Project Slides, by Rodrigo A. Lima. Presented in class Tuesday, February 11, 2020.
[6] Elba Project Tutorial 2, by Rodrigo A. Lima. https://www.cc.gatech.edu/~ral3/tutorial2.html
[7] Q. Wang, Y. Kanemasa, J. Li, C. Lai, C. Cho, Y. Nomura, and C. Pu, "Lightning in the cloud: A study of transient bottlenecks on n-tier web application performance," in *2014 Conference on Timely Results in Operating Systems, TRIOS '14, Broomfield, CO, USA*, 2014.
[8] OpenStack, https://www.openstack.org.
[9] stress-ng, https://wiki.ubuntu.com/Kernel/Reference/stress-ng.