Elba standard project

Alan Hesu and Ashwin Bhide

Submitted to:

Calton Pu

CS 4220 / CS 6235

30 November 2018

Motivation:

The performance of a lot of web applications is often degraded due to the presence of bottlenecks in the system [1]. N-tier web applications suffer from something known as the response time long-tail problem. The idea is that large web applications have a majority of their requests serviced with fast response times. However, a small portion of the requests made to the application have much longer response times. These longer response times occur even at low CPU utilization levels (~40%) when none of the hardware is near saturation. These performance bottlenecks can propagate through the system, magnifying their effects at each component. Even though the long-tail response times are in the order of seconds, these can have a tremendous impact on companies such as Amazon and Alibaba who lose millions of dollars for every 100 millisecond increase in response time [2]. Finding these performance bugs which lead to millibottlencks can help us understand the basic principles of system designs and implementation at a deeper level.

To find a bottleneck, we must study the events occurring before and during a millibottleneck. The system we plan to use, Elba [3], uses a fine-grain monitoring instrument called SysViz which runs on a Cisco router and timestamps messages at a microsecond resolution. All the nodes in the topology are connected to SysViz so that all messages are timestamped and we can perform a timing analysis of events associated with each millibottlenck.

The use of the tools in Elba tackle some of the challenges associated with monitoring and tracing millibottlenecks [1]. First, a very fine-grained temporal resolution is needed for the data, as millibottleneck events occur on the order of milliseconds. This means conventional resource monitoring tools, which may have a temporal resolutions on the order of seconds, will fail to capture the events of interest. The sampling theorem, which dictates that a discrete signal must be sampled with at least twice the frequency of the signal of interest, further increases the time resolution requirement of any data logging system. Second, the data monitoring must have a low enough overhead as to not impact the resource utilization data that it is trying to measure.

Methods:

In order to find the millibottlenecks using Project Elba, we had to create an experiment on Emulab with our configuration of w-x-y-z (where 'w' is the number of Apache HTTP servers, 'x' is the number of Tomcat application servers, 'y' is the number of CJDBC middleware servers, and 'z' is the number of MySQL DB servers). Once we created the experiment, we run the experiment on a certain workload by modifying the xml file corresponding to our experiment with an indication of workloads to run. Once the experiment was completed, it would dump the results as a tar file.

The next step was to unzip the file and parse the results to get the csv files for our analysis. We built the parsers and ran the parsers on the necessary files to get the corresponding csv files. Once we got these csv files, we plotted them using Matlab in order to generate graphs to perform further analysis. The Matlab script initially displays the point-in-time response time graph for the entire three minute duration of a chosen experiment. The user can then select a starting point of interest. Then, the script automatically reads in other relevant files to plot response time, queue length, and CPU and disk utilization for a 10 second window starting at the specified point. Using the point-in-time graphs, we identified potential millibottlenecks where there were sudden spikes in response time of the point-in-time graphs. We then looked for corresponding increases in queue length for other tiers such as Tomcat and Apache in the same time period.

We also created a peak detector that detected if the there was an anomaly in the sliding window. To do this, we created a sliding window and calculated the average, and then checked if the next point was significantly greater than the average of the sliding window, this was useful in helping us narrow down the time intervals at which a millibottleneck might appear. However, as data became noisier at higher workloads, this functionality became less accurate.

As there were a few bugs introduced in an earlier patch in Elba, we used data from past semesters and performed an analysis after parsing these files. Namely we used the 3-3-1-1 configuration data from a past semester.

Results and analysis:

Data was gathered for a series of experiments run on a 3-3-1-1 topology with workloads ranging from 1000 to 30000. For each workload, the point-in-time response time graphs were generated using the read-write (RW) data. Subsequent sets of graphs were generated for suspected millibottleneck causes. These largely arose from saturated resource utilization for either the CPU or disk at the CJDBC or MySQL tier. We suspect that these began the push-back wave that would propagate upwards and eventually affect the Apache nodes and ultimately the request response time.

5000 RW workload

One such instance of a millibottleneck at a workload of 5000 can be seen in in Figures 1-3. As shown in Figure 1, the response time jumps above several hundred milliseconds for a short duration. This event coincides in time with the momentary increase in queue length at all four tiers of the system, as seen in Figure 2. With the Apache tier, however, the queue length does not exceed the queue size upper limit as specified by Wang et al [4]. This indicates that no packets were dropped, though a slow down in response time still evidently occurs. The cause may be seen in Figure 3, which shows the disk utilization at the MySQL tier. During the short period of interest, the disk utilization suddenly reaches 100%. This suggests that the MySQL node was performing a disk i/o operation that slowed down the processing of requests. This delay propagated through the other three tiers. A second, smaller peak in the response time can be also be seen, though the magnitude is so low that this is most likely not attributed to a significant millibottleneck.

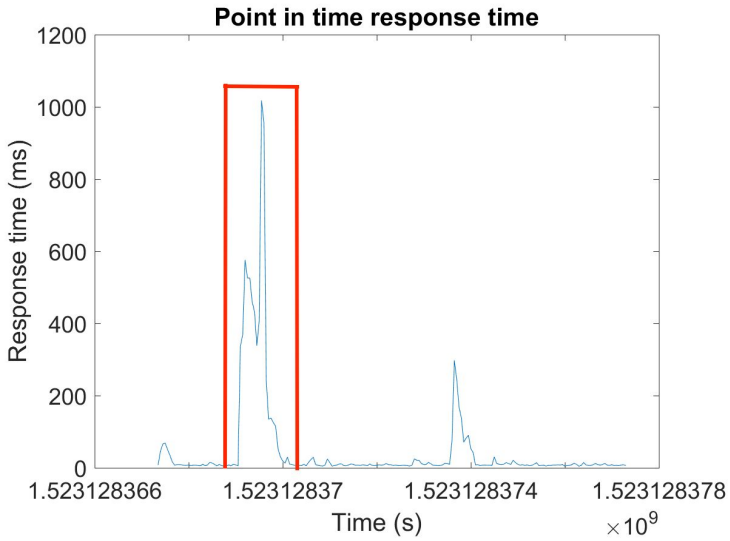

Figure 1: Response time

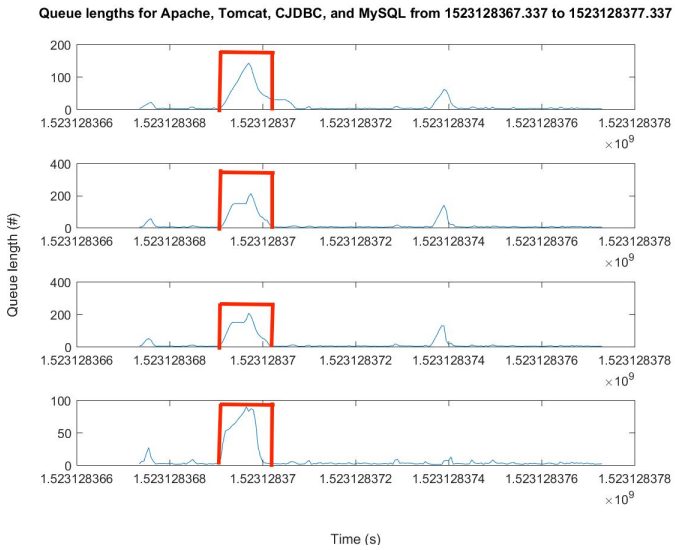

Figure 2: Queue lengths for system components

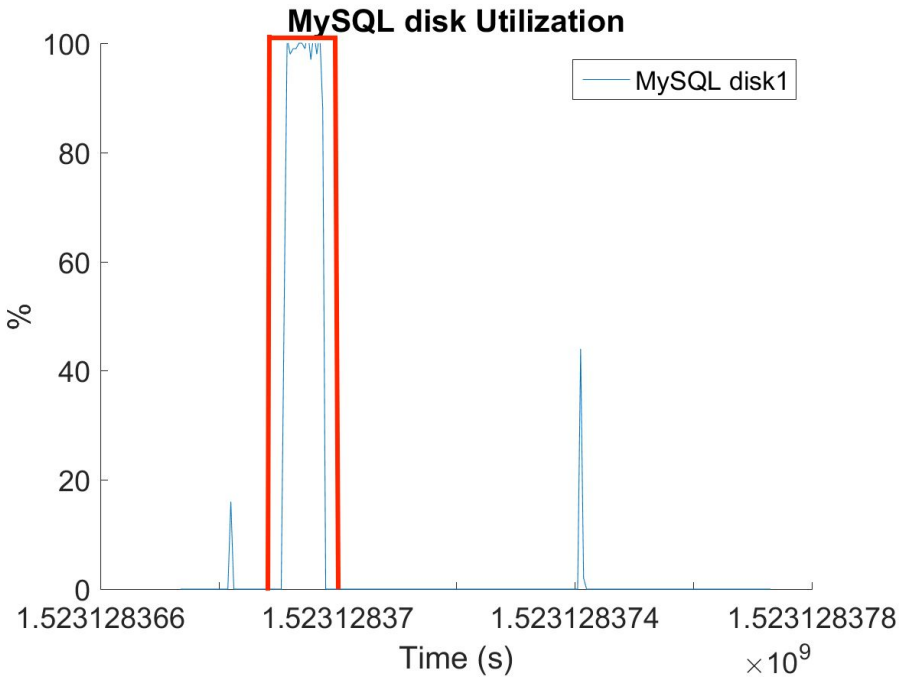

Figure 3: Average MySQL disk utilization

10000 RW workload

Another instance of a millibottleneck can be seen in Figures 4-8, which represent an experiment at a workload of 10000. Unlike with a workload of 5000, the response time here consistently remains higher, though there still exist short windows where the response time peaks at a higher than average value. The overall slower performance may be due to the very high queue length at the Apache tier, which can be seen in Figure 5. Here, the queue length often rises higher than the upper limit of 278, which indicates that packets are being dropped at a relatively high rate. Another issue with the Apache tier can be seen in Figure 6, where the CPU utilization remains close to saturation nearly the entire time for just one CPU. The other two CPUs see very little to no load, which suggests the experiment may have been poorly conditioned or that the other two nodes were somehow not processing as many requests. The two points in time where the CPU utilization drops correspond to points where the response time increases, which may indicate that the Apache node itself is waiting idle on downstream requests to finish processing. Figures 7-8 may corroborate this theory, as the disk utilization in the CJDBC and MySQL nodes experience a sudden spike that coincides with the previously discussed two points in time.

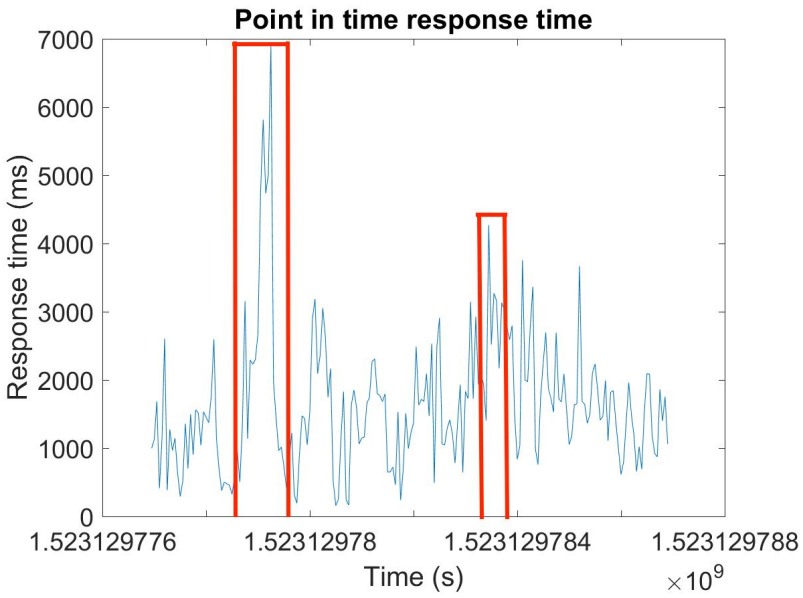

Figure 4: Response time (10000 wl)

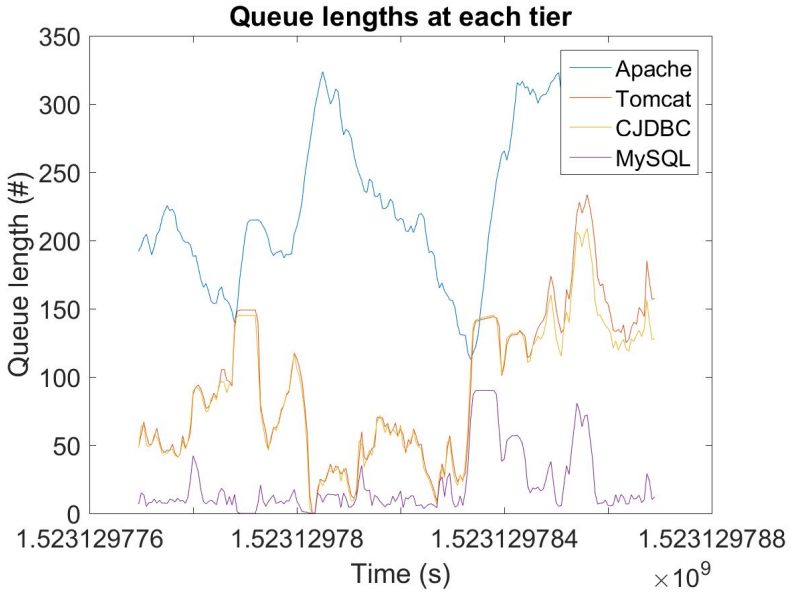

Figure 5: Queue length (10000 wl)

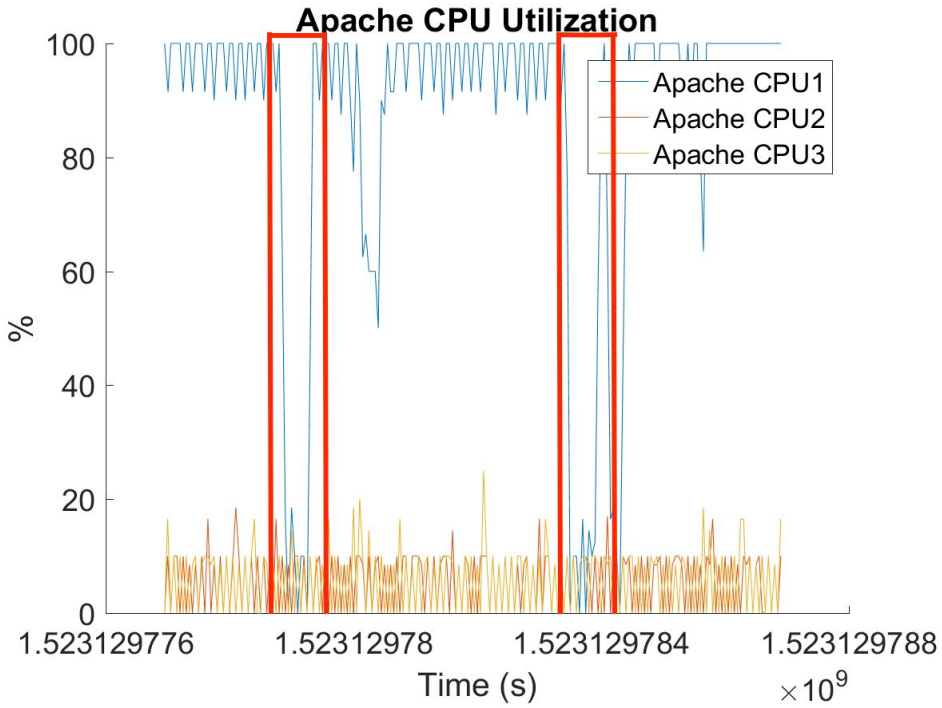

Figure 6: Apache CPU utilization (10000 wl)

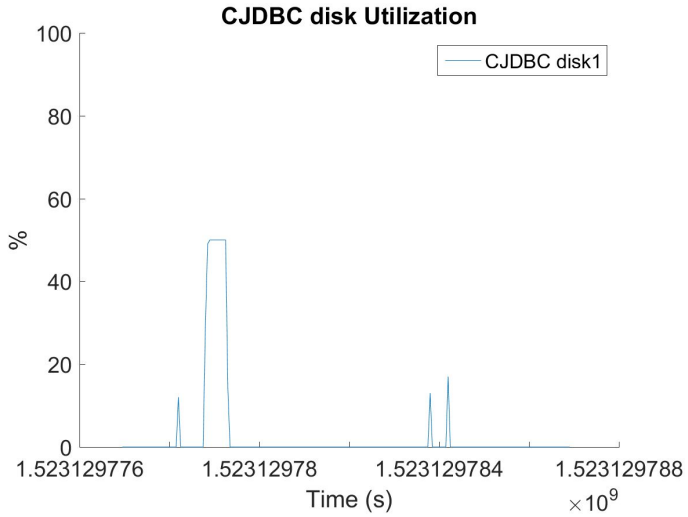

Figure 7: CJDBC disk utilization (10000 wl)

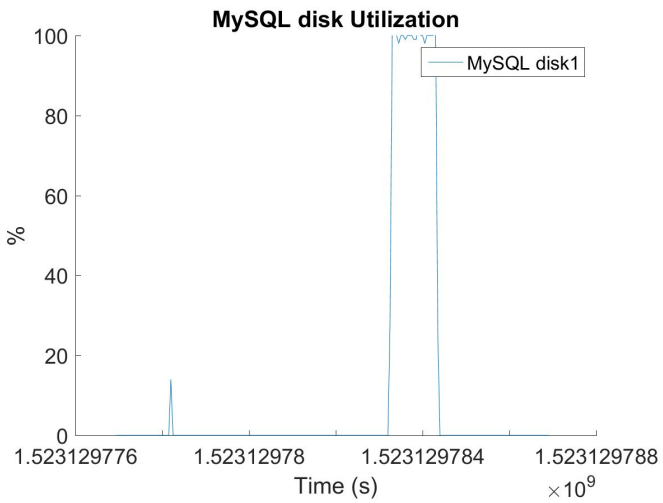

Figure 8: MySQL disk utilization (10000 wl)

5000 RW workload

As you can see, in figure 9, the point-in-time response graph shows peaks in 2 time intervals. These are good places to start looking for millibottlenecks. When we look at the corresponding queue length graphs for Apache, Tomcat, CJDBC and MySQL in figure 10, we can see that the peaks appear in these graphs as well. However, the peak on the left stops at the CJDBC tier, while there is no peak for the MySQL layer. This indicates that there is something going on in the CJDBC tier which causes the millibottleneck. Our hypothesis based on the figures 9, 10 and 11 are that the CJDBC tier is responsible for the millibottleneck on the left while the MySQL tier is responsible for the millibottleneck on the right. This claim is supported by the peaks (absence of peak in case of the peak on the left in the MySQL tier) in the queue length charts for the system components.

We can see from figure 11, that the CPU utilization for the CSJDB node reaches saturation when the millibottleneck is formed. This also backs up our hypothesis of the CJDBC tier being the cause of the millibottleneck.

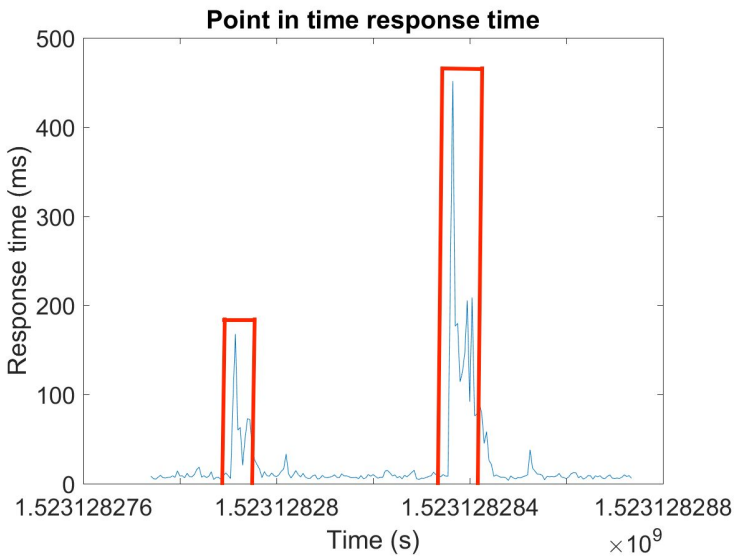

Figure 9: Response time (5000 wl)

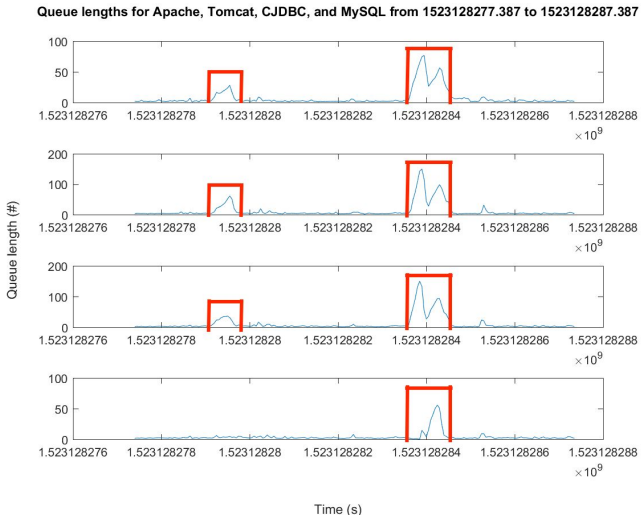

Figure 10: Queue lengths for system components (5000 wl)

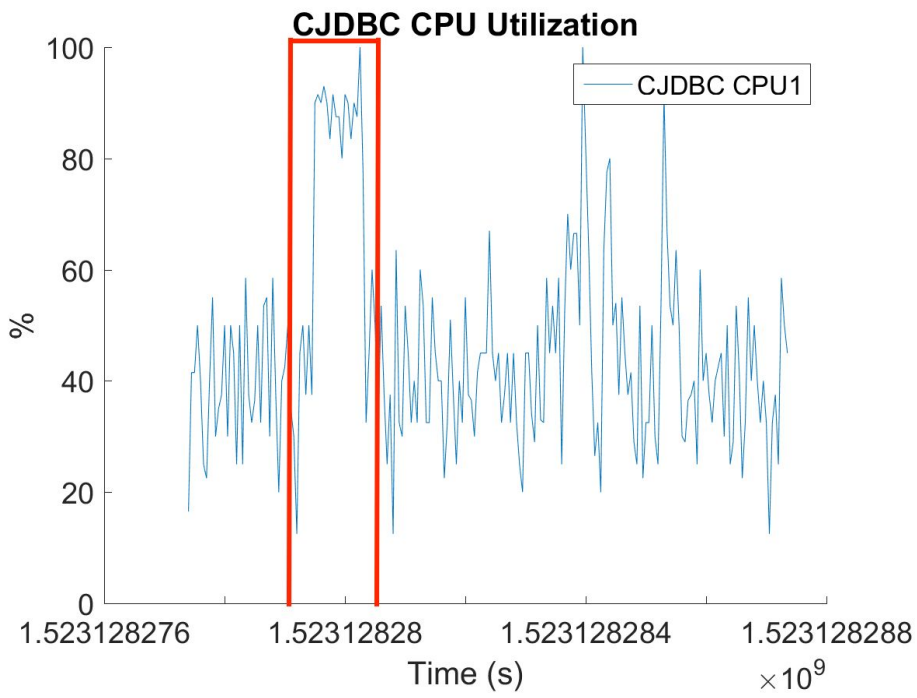

Figure 11: CJDBC CPU utilization (5000 wl)

1000 RW workload

At smaller workloads, shown in Figures 12-14, the response time may still experience momentary peaks, as shown in Figure 12. However, the magnitude of this peak is less than 150 ms, which is comparatively much smaller. Additionally, the duration of the peak is very small, which suggests that this phenomenon may not represent a significant millibottleneck event. The queue lengths, as shown in Figure 13, also experience a peak at the same time, but they do not approach saturation at all. Interestingly, the CJDBC disk utilization graph in Figure 14 shows a peak at the same point in time as well and the MySQL queue length does not rise, so this may indicate another instance in which the delay began at the CJDBC tier.

From figure 12, we can see that there is a peak. However, this is not necessarily a millibottleneck. This peak only has a response time of just under 150 milliseconds. The corresponding queue length charts (figure 13) show us that the queue length is indeed showing a spike, but the spikes all have a queue length of less than 20. On first glance, it looks like there may be a millibottleneck, however, on closer inspection that is shown as incorrect due to the short queue length.

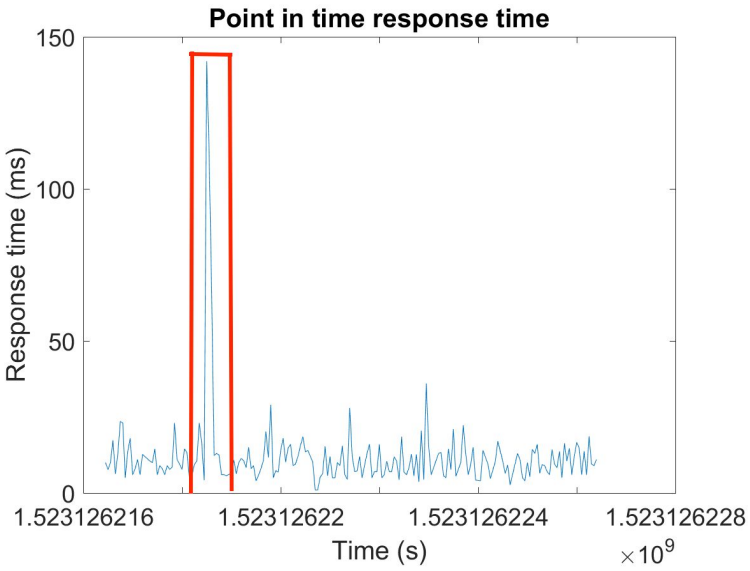

Figure 12: Response time (1000 wl)

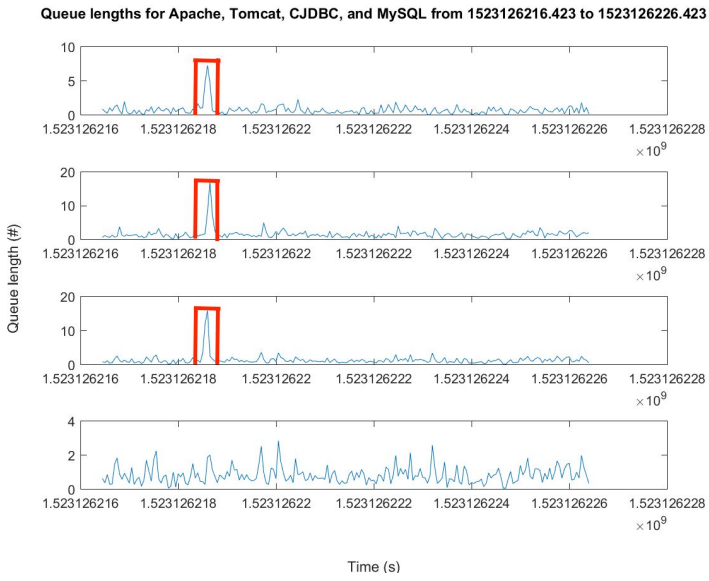

Figure 13: Queue length graphs for system components (1000 wl)

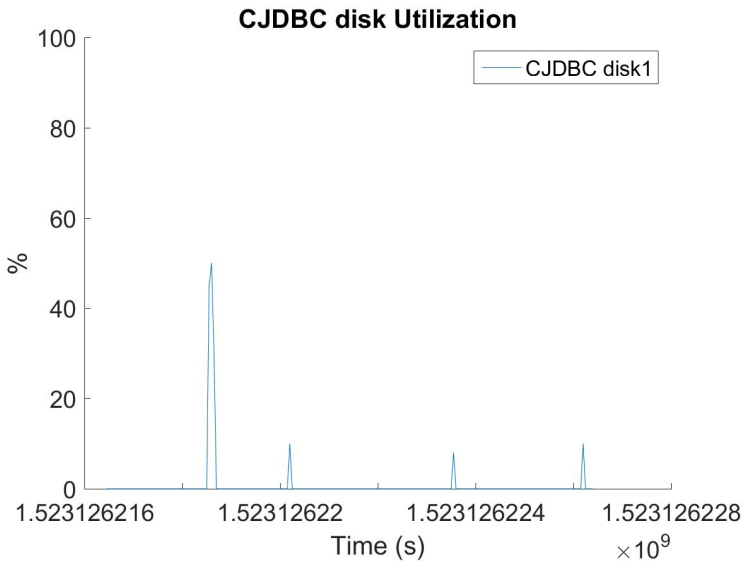

Figure 14: CJDBC disk utilization (1000 wl)

20000 RW workload

Sometimes, an increase in queue length or resource utilization does not necessarily correspond to a noticeable increase in response time. Figures 15-21 represent a workload of 20000. In Figure 21, the MySQL disk utilization saturates momentarily, which would normally suggest a millibottleneck that propagates through the rest of the system. Furthermore, the CPU utilization at each tier, as shown in Figures 16-19, drops at the same instance in time, suggesting that each tier is waiting idle for MySQL to finish the disk i/o operation and finish processing requests. The queue lengths, shown in Figure 20, also increase accordingly as the delay continues. However, this point in time in the response time graph, as shown in Figure 15, does not experience a marked increase or peak that has been seen in the previous examples. This may be caused by the fact that the Apache queue length does not experience the same rate of increase, nor does it rise above the queue size limit.

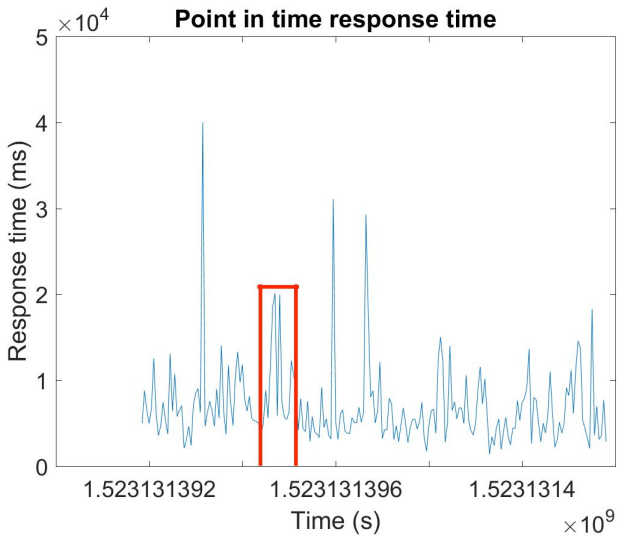

Figure 15: Response time (20000 wl)

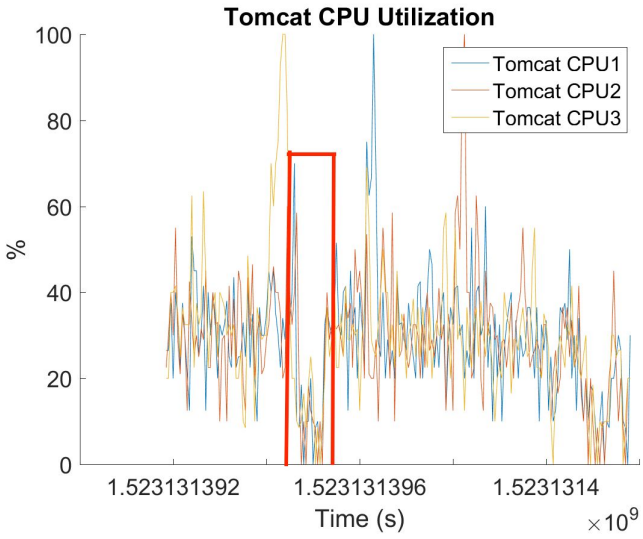

Figure 16: Tomcat CPU utilization (20000 wl)

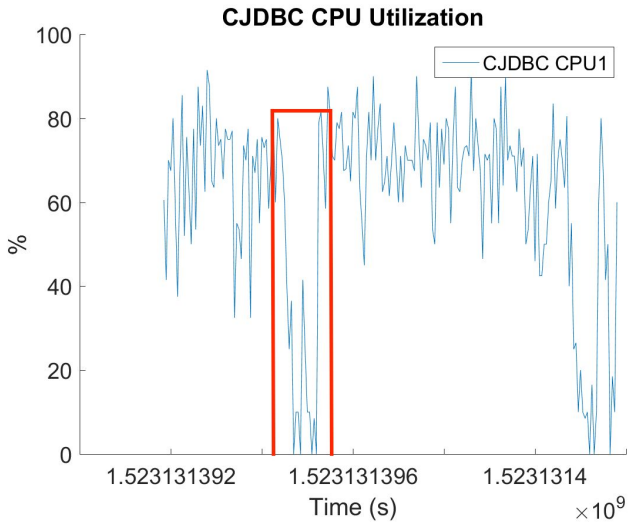

Figure 17: CJDBC CPU utilization (20000 wl)

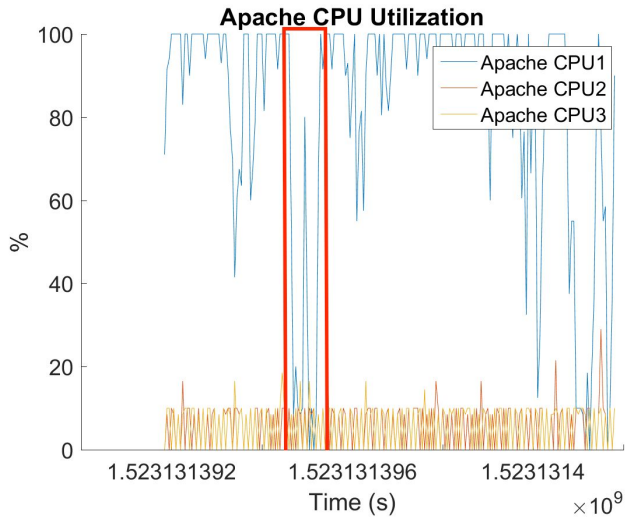

Figure 18: Apache CPU utilization (20000 wl)

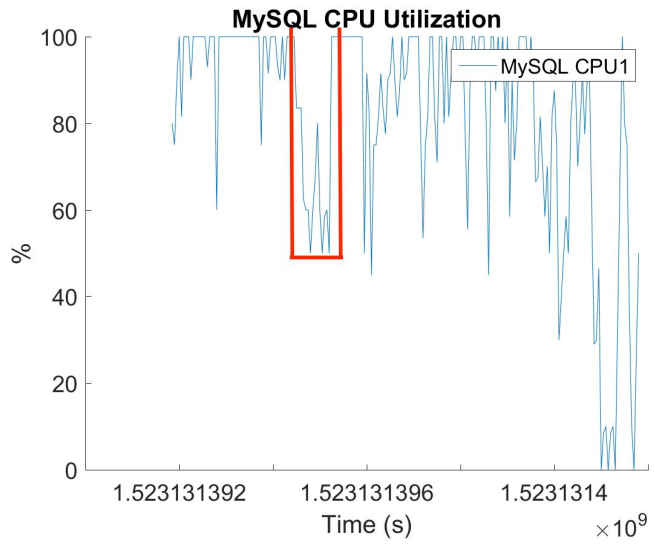

Figure 19: MySQL CPU utilization (20000 wl)

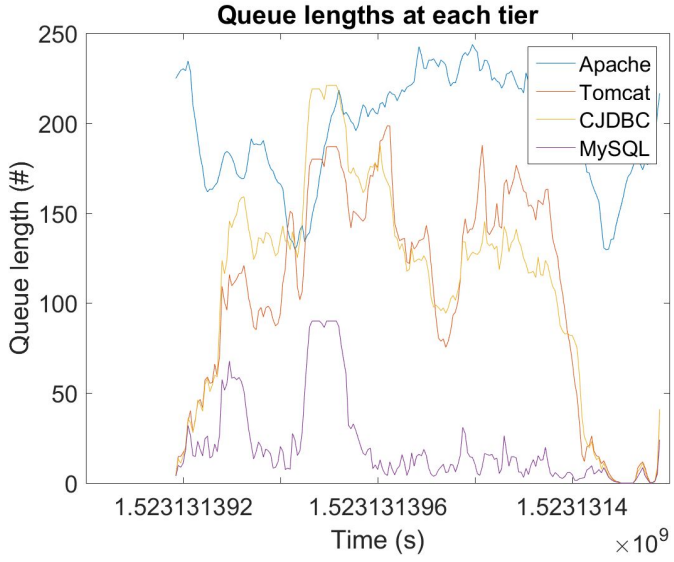

Figure 20: Queue lengths for system components (20000 wl)

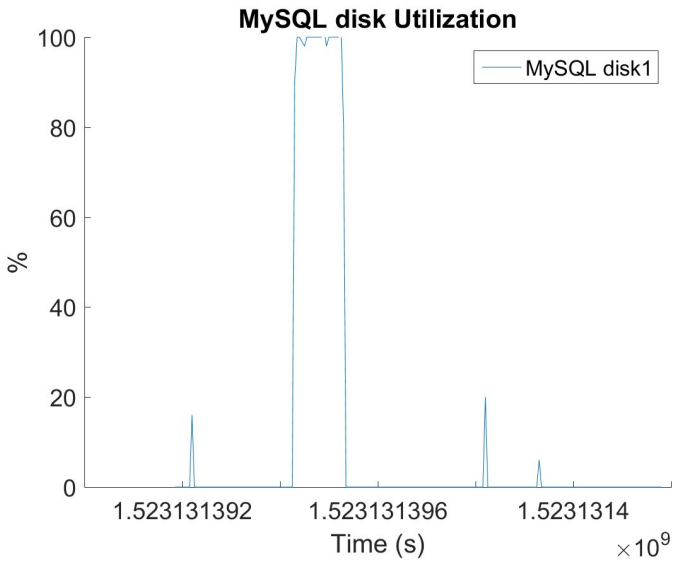

Figure 21: MySQL Disk utilization (20000 wl)

Issues:

While trying to run experiments and detect millibottlenecks we encountered a few problems with Elba and the parsers. We found that on running the experiments, at times, we would not have all the files necessary to graph the outputs. Rather, some of the files were missing. This was due to a bug introduced by an earlier code push.

Another issue we ran into was a number of 0's filling up the csv files which were resulting in a graph providing no real data. There are two sections in the rubbos.properties files that should not have been present, someone introduced this bug during a code push as well.

It took us some time to understand how to find a millibottleneck and how to identify the cause of the millibottleneck, but after understanding the chart and overlapping charts to find a peak in a point-in-time graph corresponding to an increase in queue size graph we started to understand how to find millibottlenecks.

There was a few issues in the parser as well, where we had to change a file path, or give it an absolute file path in order to get it to work.

Conclusion:

Very short millibottlenecks result in significant performance bugs. These millibottlenecks result in the introduction of long response time problems due to which the response time gets amplified through each of the system component that amplify the effects of the millibottlenecks

Due to the short durations of millibottlenecks, we need instruments capable of monitoring resources at a fine-time granularity. This is important as events may occur less than milliseconds from each other which results in millibottlenecks, so it is important that the resources be monitored at a very fine granularity in order to get the entire picture of what is happening in the system around the time of a millibottleneck. This project aimed to gather data from a non-trivial topology and plot graphs that would help analyze and detect the source of millibottlenecks. Although the experiments were not successful, the plots generated from older data may suggest possible causes. In the future, further automation of parsing the results and analyzing the data may greatly improve the number of experiments that can be run and the potential causes that can merit investigation.

Sources:

[1] C. Pu *et al*., "The Millibottleneck Theory of Performance Bugs, and Its Experimental Verification," *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, Atlanta, GA, 2017, pp. 1919-1926.

[2] R. Kohavi and R. Longbotham. “Online experiments: Lessons learned,” *IEEE Computer Society*., vol 40, pp. 103-105, Sept. 2007

[3] GitHub. (2018). Tutorial: Bootstrap & Experiment Execution (alpha). [online] Available at: https://github.com/coc-gatech-newelba/coc-gatech-newelba.github.io/wiki/Tutorial:-Bootstrap-&-Experiment-Execution-(alpha) [Accessed 21 Sep. 2018].

[4] Q. Wang *et al.*, “Lightning in the Cloud: A Study of Very Short Bottlenecks on n-Tier Web Application Performance, “ *2014 Conference on Timely Results in Operating Systems*, Broomfield, CO, 2014.