Visualizing Load in Submitty

Evan Maicus, Timothy Cyrus

May 1, 2018

1 Introduction

The Submitty Homework Submission Server is the assignment submission and automated grading platform used at Rensselaer Polytechnic Institute. Designed to be highly configurable, Submitty services 11 courses and over 1000 students at RPI. As Submitty has expanded, both in terms of adoption rate and feature set, diagnosing system problems has become increasingly complex. For our final visualization project, we proposed an extension to Submitty to provide a graphic representation of system load which we define as the number of student assignments being graded at a given timestep.

Our visualization displays submission load across all courses, providing a holistic view of system throughput and submission density at peak hours. We anticipate that this visualization will be useful to Submitty administrators, who currently must crawl through log files in order to gain insight into these important statistics. We anticipate that this feature will be useful in identifying high volume courses and hours in Submitty, and will therefore serve to facilitate any future work in balancing load on the submission server.

We further hypothesize that these additional statistics will be particularly relevant in the light of a recent extension to the homework submission server which has rendered it capable of running jobs on foreign machines (workers). This functionality allows professors to donate specialized machines to Submitty, thereby facilitating the grading of more complex assignments. With this in mind, our visualization has been made capable of displaying load both on main Submitty and on any worker machines registered to it.

Upon beginning our project, we hypothesize that load would peak before assignment due dates, and particularly on Thursday evenings, when Data Structures assignments are typically due. Furthermore, we hypothesize that submissions for each class will cluster close to the deadline, rather than being evenly distributed across the assignment period. Finally, we hypothesize that assignments allocated to a worker machine will take longer to grade than assignments graded locally, but will negligibly affect the response time of primary Submitty. Unfortunately, at this time, we have not yet integrated our visualization into the RPIs active Submitty installation. That being said, we maintain our hypotheses, and fully intent to explore them in future work (see Section 5 for details).

2 Background Material

2.1 Project Proposal Feedback

After proposing our project on the Submitty forum, we received feedback from two of our fellow students. These classmates expressed some concern about how frequently our algorithm would be able to update given nature of the log files that it would need to read, and further expressed concerns that we could slow down Submitty as a whole by attempting to read these logs. Both individuals recommended that we consider using a Streamgraph to represent our data, citing their usefulness when showing part to whole relationships over time. It was one of these peers who first expressed an interest in how our visualization could be used to view load associated with individual courses rather than just worker/primary Submitty load, which had been our focus up to that point. As we moved forward with our project, we took our classmates worries to heart, working hard to make sure that our visualization interfaces with Submitty with as little impact as possible. In particularly, we integrated new log lines into Submitty to greatly simplify our parsing (see Section 3 for more details). We incorporated the idea of viewing load by class into our project, making it one of our core focuses. Finally, we considered using a Streamgraph, as our classmates proposed, but decided to use the related Stacked Area Graph instead (See Section 4 for details).

2.2 Summary of Related Works

In their paper Massive Distributed and Parallel Log Analysis, Shu, Yao, and Lin present an algorithm for searching a distributed log system for potential security risks [1]. In order to do this, they specify a master machine, and then multiple storage slave and computation slave machines [1]. This setup is not unlike Submittys primary/worker relationship. Their algorithm then calls for the master machine to specify what data each computation machine should process in a coordinated distributed search [1]. While simple, perhaps the most meaningful lesson that we learned while reading this paper was that the logs themselves do not necessarily have to be passed to primary Submitty; instead, a more compact meta-log can be computed by the worker machine itself, which holds load data for the machine [1]. As we worked on our project, we were eventually able to move away from distributed log analysis by adding additional log lines that prevented us from having to ship logs from worker machines to primary Submitty. Despite this, we still used compact metadata to feed our visualization, in much the same manner as Shu, Yao, and Lin.

In Spell: Streaming Parsing of System Event Logs, Du and Li present a longest common subsequence approach to parsing system event logs [2]. Of particular interest to us at the outset of our project was the fact that the authors system is able to take in an arbitrary set of log lines and break them into classifications, or message types, in near linear time [2]. To do this, Du and Li made the important observation that any given log file can be viewed as a tuple containing a message type and parameter values [2]. For example, the log lines System Error: temperature at 52 degrees and System Error: temperature at 102 degrees represent two logs with the same message type (a temperature error), but with different parameter values (52 and 102 degrees, respectively). After making this observation, the authors then discovered that, in their system at least, the vast majority of log lines were dominated in size by their message type rather than their parameter values [2]. With this important observation in mind, they determined that it should be possible to extract message types from an arbitrary set of system logs using an algorithm based on Longest Common Subsequence [2]. The LCS problem (as it is presented in this paper) asks for the longest subsequence (where two tokens in a subsequence need not be adjacent) that is common between two lists of tokens [2]. So, for example, the longest common subsequence between ABDEF and ADGLF is ADF. Du and Li reasoned that, because message types dominate log line size, and because two log lines of the same message type should differ only in their parameter values, that log lines can be easily classified using LCSs [2]. In order to quickly classify/assign a message type to arbitrary incoming log lines, the authors build a prefix tree, or trie, with tokens from previously encountered LCSs as its node. This means that, for a previously encountered log line, classifying it takes nearly $O(n)$ time, where n is the number of words in the log line [2]. While the contributions of this paper fell outside of the scope of our project, knowledge about how to efficiently determine message types and parameter values for an arbitrary set of log lines in an automated way could be very useful if further visualizing Submitty log statements in the future.

For a time when beginning our project, we thought that we would have to ship log files from worker machines to primary Submitty in order to get a complete record. During this period, we investigated the use of rsyslog, an open source implementation of the UNIX syslog protocol [3]. Rsyslog is primarily used for sending logging information quickly over a network [3]. This would have been useful in syncing up logs between the worker Submitty machines and main Submitty, but would have required an overhaul to the way Submitty currently handles its logging.

Early in our development, we experimented with multiple different packages, mostly built atop D3.js, to determine which provided the best feature set to meet our needs. During this time, we experimented with Perspective [4] and Epoch.js [5]. Perspective is a visualization tool built by J.P. Morgan for real-time customizable visualization [4]. It is especially well suited for allowing the user to apply filters and to modify their axes in much the same way that they might in Google Sheets or Excel [4]. Epoch.js is a relatively new package, which is built to created stacked area charts with streaming data [5]. While this seemed at first like the perfect fit, its poor documentation and near total lack of customizability made it all but impossible to implement any of our required interaction using this platform. In the end, we settled on using NVD3.js, a D3.js extension based on the work of Mike Bostock [6]. This project provided good starting point implementations for us to work with, while maintaining a high degree of customizability.

3 Data Collection

Our original log parsing program was taken from work done for the Submitty Docker poster for SIGCSE [7]. This program removed unnecessary and redundant content and created csv files based on the data in Submitty log files. This parser was originally built for reading logs from the main Submitty machine, as remote grading had not been implemented at the time that the poster was created. As a result, we modified our parser to support remote machines as well as main Submitty. To avoid the need for anonymized logs while testing, a random log generator was created, which initially output csv files where the first column represented a time, and the subsequent columns represented machine and course load at that time step. In order to make the process of determining load clearer, we added two additional log lines to the main Submitty server. These log lines represent a start and an end time for a particular job, which was specified by an untrusted user (which can be thought of as a unique grading thread) a course, and the machine to which it was bound. Encountering a start log line, then, meant that we could increment the amount of load for its associated machine and course. Similarly, encountering an end log line meant that we could decrement the amount of load for its associated machine and course. Later, WebSockets were used rather than csvs to reduce parsing for the client and load on the server. Using these WebSockets allowed us to bypass the use of csvs entirely, instead providing data directly to the server at a specified interval. Issues that may arise when using WebSockets involve potential complications when streaming to many clients simultaneously and making sure the initial data and future rows of data are in sync.

4 Evolution of Design

4.1 Planning

As we began putting the systems in place to collect, simulate, and parse our dataset, we began brainstorming the list of features that would be necessary to easily and intuitively present it. After some discussion, we decided on the guiding principle that our project must allow administrators to find critical moments in Submittys execution quickly, and came up with the following necessary features:

1. Our visualization must display aggregate load to its viewer, allowing them to get a holistic view of the state of Submitty at a moment in time.
2. However, our visualization must also be able to show load by individual course, so that the state of a courses submissions can be viewed at a time step and weighed against the machines total load.
3. Similarly, our visualization must display load by machine.
4. Our visualization must be able to display both the batch and interactive load associated with a given machine.

As we developed this list of core features, we took note of our datas part to whole relationships: total system load is made up of the sum of all machine or course loads, and each course or machine load is made up of its batch and interactive loads. Also of note was the fact that our data is inherently temporal. Per our classmates recommendations (see Section 2.1), we considered displaying our data as a Streamgraph. However, noting this graph types penchant for being difficult to read at a glance, we decided to use the related Stacked Area Graph, which satisfies all of our necessary features while simultaneously maintaining relatively high readability.

After we decided on the stacked area graph as the template for our visualization, we developed a list of the interactive features we would implement to make our visualization simple to read and make our data easy to explore and interact with:

1. Because of the stacked nature of our visualization, we knew that it would be difficult to tell the exact size of some areas at a glance. In order to rectify this issue, we decided to add focus+context functionality, allowing the user to toggle between viewing all stacked areas and only an area of interest, snapped to the x axis.

2. Despite our focus+context features, we still wanted to facilitate allowing a user to view all regions at the same time by giving them the data that they need at a glance. To do this, we conceived of bubbles, mouseover regions for each area at each timestep that give the exact value associated with that area at that time.

3. We knew that a user might want to switch between viewing load by course and load by machine at any time. To facilitate this, we aimed to add a toggle button to switch between the two with minimal loading time or other interference.

4. As a stretch goal, we also set for ourselves the objective of making our project update once per second with new data.

4.2 Initial Design

At the outset of this project, we intended to craft our visualization by modifying an existing program written in Perspective. This program was made for visualizing Submitty log files, and had been used previously in Submitty research to inspect the performance impact of running student submissions inside of Docker containers [7]. Our plan was to strip away unneeded information, including submission grades and usernames, and instead to focus only on assignment start and end times. However, we quickly found that Perspective was ill-suited for representing stacked areas, a core feature of our project. As a result, we shifted our focus, and created our initial mockup directly in D3.js.

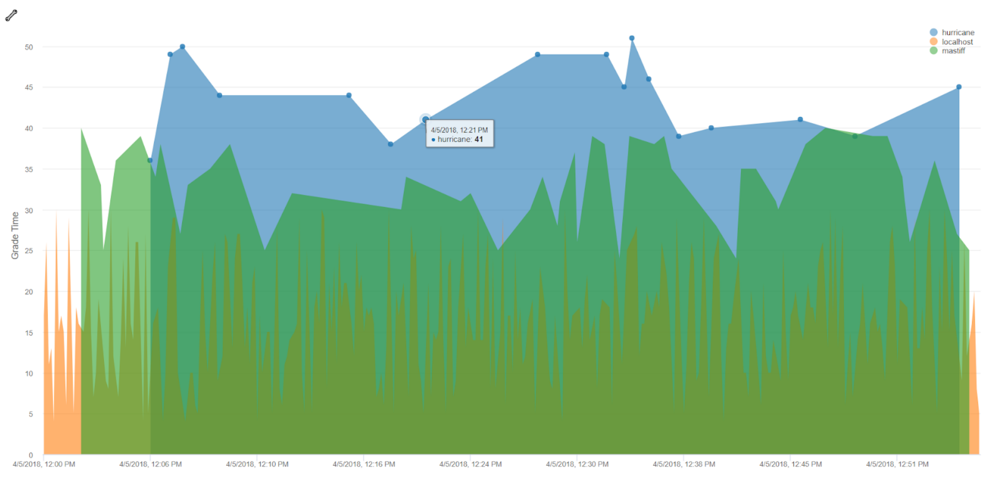

Figure 1: Our initial Perspective mockup, presented for peer review. This initial visualization was scrapped after not being able to show stacked area graphs

4.3 Peer Feedback

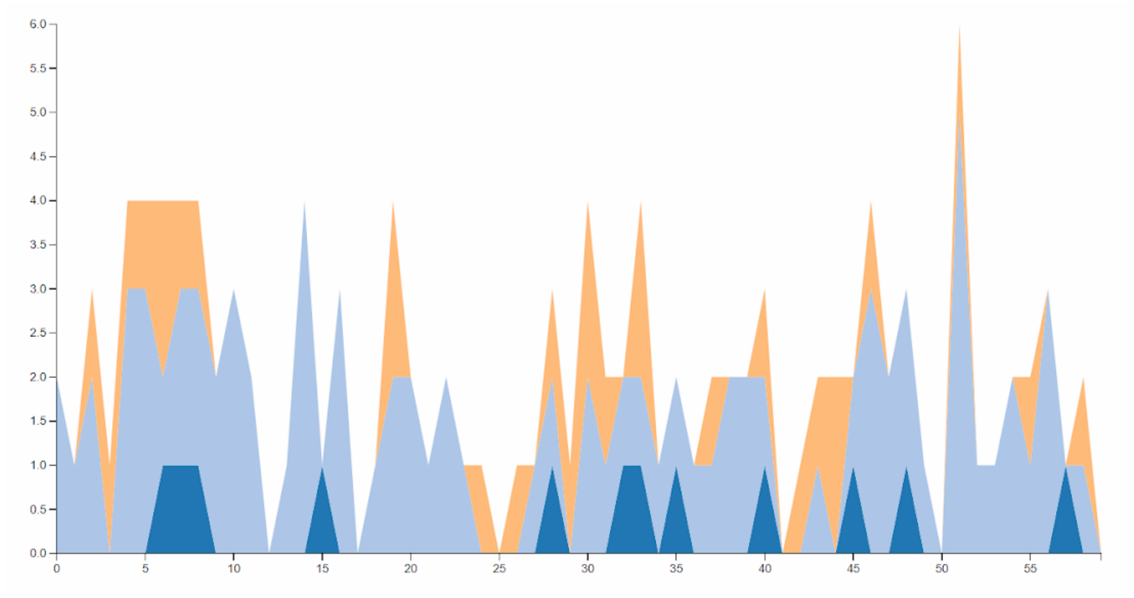

Figure 2: Our initial D3 mockup, presented for peer review. This initial draft was non-interactive, and merely represented machine load as stacked areas without differentiating batch and interactive load.

By peer feedback day, we had created the first draft of our log parser, and had integrated our new logs into Submitty. We created a D3.js mockup (Fig 2), and developed a list of question that we believed would provide valuable insight into how users would like to interface with our project and see our data represented. We were aware that our classmates didnt map exactly onto our intended set of users (Submitty instructors and administrators), and as such, we chose questions that we thought would be easily understood by non-expert users.

Before peer feedback day, we had been struggling with the correct way to visualize the interactive and batch load for a course or machine. The difficulty extended from the fact that these regions while distinct, are tied together for a given course or machine. In this way, they can be seen as sub-areas of an individual machine or courses load. We asked our classmates how they would like to see this relationship visualized, and they came through in spectacular fashion; some respondents recommended making batch and interactive submissions for a single machine or course have similar colors, which had been our plan up to this point. A few other respondents, however, recommended using a hatched sub-area within each of our larger areas to represent batch submissions. These subareas would be the same color as the interactive area, but would be textured to make them easy to distinguish. This feedback was invaluable, and was implemented into our final project.

Early on in our design process, we spoke at length over how we would define load as the number of assignments being graded at a time, or as a function of average wait time. On peer feedback day, we asked our classmates which of these definitions they preferred, and which they found yielded a more intuitive visualization. Three respondents preferred load as a function of how many assignments were currently being graded, stating that it was easy to understand and clearly conveyed the number of jobs at a timestep. One respondent preferred the second definition, stating that they preferred the way that it conveyed the difficulty of the jobs running. In the end, we moved forward with load as it is defined in this paper (total number of jobs being graded at a time).

4.4 Final Visualization

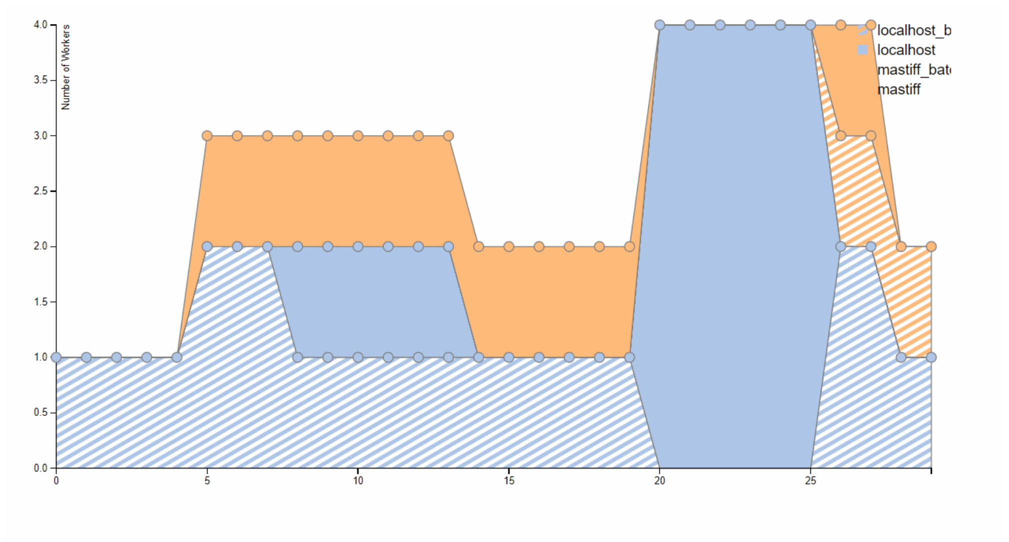

Figure 3: Our extended D3.js mockup. Note the hatched subareas for batch submissions, the mouseover Bubble system for viewing exact details at a timestep, and the generally improved aesthetics.

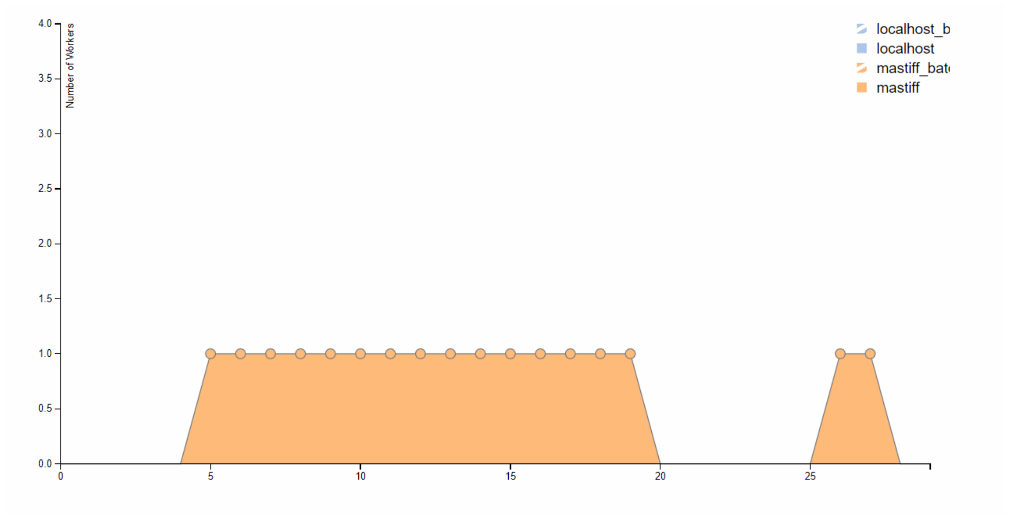

Figure 4: An example of our focus+context functionality after the mastiff region from Figure 3 has been clicked.

After peer feedback day, we continued our work in D3.js, extending our visualization to have many of the features that we initially determined to be necessary (Fig 3). This iteration contained many improvements over our initial mockup. First and foremost, it implemented our bubble system, allowing users to mouseover a region and see an exact value for the amount of load associated with it at a given timestep. Second, this version incorporated a focus+context extension, allowing users to click on an area of interest to snap it to the x axis and remove all other areas, making it much easier to track one region over time (Fig 4). This version also came with many aesthetic updates, including a legend, harder edged areas, a larger chart size, and hatched subareas for batch submissions as was recommended by our peers. This iteration contained some bugs, however; in particular, a race condition existed which would cause areas to bleed into one another and miscolor if clicked rapidly.

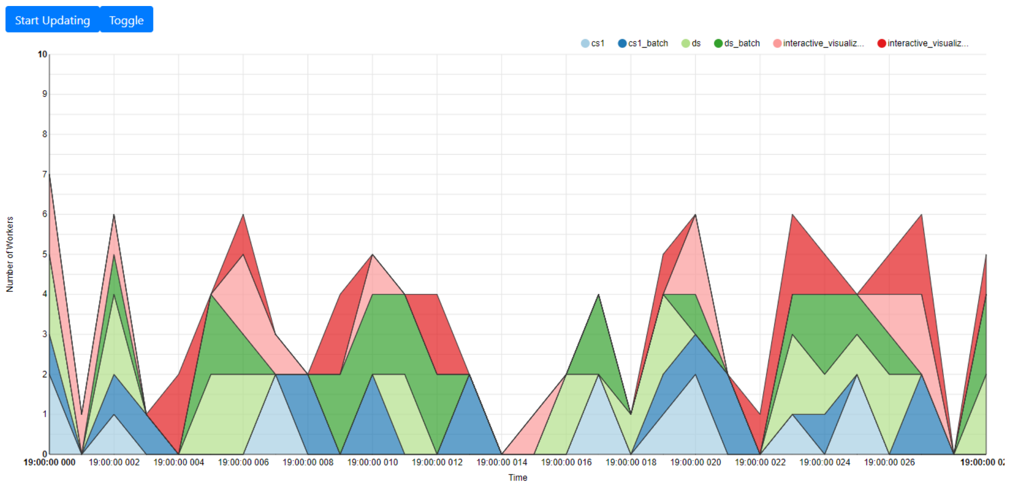

Figure 5: Our final visualization, made using NVD3.js. Note the new legend, which doubles as a selector for focus+context visualization. Further note the new toggle button in the upper left, which allows the user to switch between viewing load by course and viewing load by machine. Finally, note the start updating button, which allows the user to pause incoming updates as logs are written in.

Shortly after completing this D3.js mockup, we discovered the NVD3.js package, which came with many utility extensions we believed would benefit us greatly. The process of porting our visualization from one framework to another was tricky, and cost us a few days time as we worked to replicate our previous functionality. When we had, however, we saw the benefits almost immediately. First, and most significantly, NVD3.js provided similar focus+context interaction for free, but without any of the race conditions that had plagued us in our earlier iteration. Just as significant, with some customization we were able to add a hover guideline (Figure 6). This guideline allows users to view the total state of the system at a time step via a mouseover cut of the stacked areas. This functionality gives a far better picture of the system than our bubble extension, and is also more intuitive and less prone to errors.

Once we finished our port, we began adding additional features as our presentation fast approached. First, we configured our project to use NVD3s multi-region choosing. This can be viewed as an extension of our focus+context interaction, and allows a user to select a

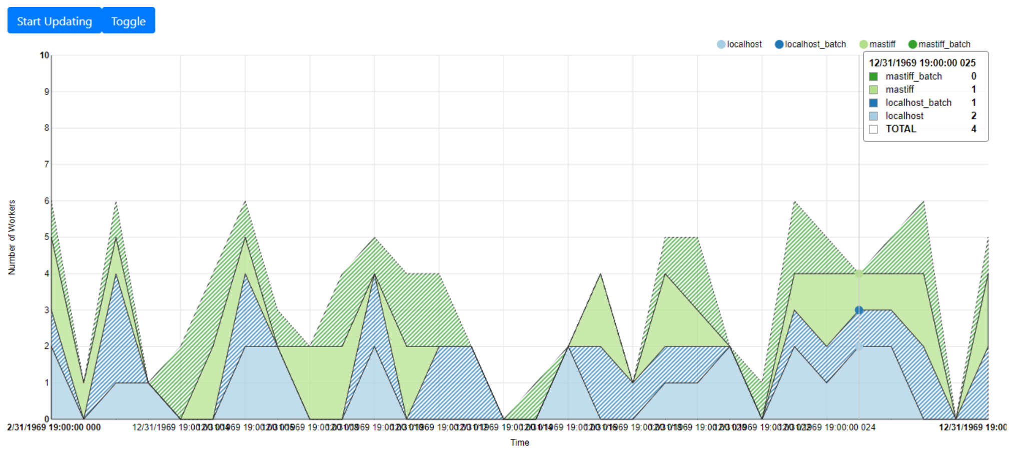

Figure 6: The same dataset as Figure 5, but displayed by machine instead of by course. Note the hover guideline in the upper right, which is showing the total load across all machines at that time step.

subset of the regions to view. Next, we added support to add new machines on the fly, and also for the y axis to scale as new max loads are encountered. This is all done in a dynamic way, without a reload of the visualization being required.

The final, and most difficult extension to our project was adding support for the automated streaming of new data. In order to facilitate this, we had to remove our projects reliance on csv files. We added support for streaming data from a WebSocket, which allowed us to avoid reloading and rereading a csv file at each server tick. This extension allows our parser script to sit atop a Submitty log, reading it as it is written and shipping off the data without writing any intermediate files. With this functionality in place, we were able to update our charts data as new information was retrieved from the simulated parser application running on the host. Finally, we added a button to our visualization which allows the user to pause incoming updates. This utility makes it easier to focus in on one timestep without it being shoved slowly out of view.

5 Future Work

While we have implemented the vast majority of our projects core functionality at the time of this writing, we find that there is still a sizeable list of utility features which we would like to add. First and foremost, our visualizations x-axis does not scale appropriately as additional data is loaded, a task which is made difficult by a limitation in NVD3.js. This functionality is under development even now. However, and we expect it to be added in the coming days.

Another large area that we want to explore is visualizing Submitty scheduling errors as their log lines occur. We have put a good amount of thought into this functionality, and have even sketched out a draft of what it will eventually look like. This feature should be added to the visualization this summer.

As we fix our x-axis scaling problem and become able to view even ever more data, utility features for navigating through time will be necessary. During our demos question and answer section, one student asked if we will provide jump to a timestep functionality. We fully expect to implement such features as we work to deploy our project on Submitty.

Finally, as was previously mentioned, we fully intend to deploy our application on Submitty for administrators to use. We earnestly believe that this functionality will be useful in the display of Submitty log files and in providing at-a-glance information about the state of the system. We fully intend to deploy integrate our visualization into Submitty for the Fall 18 semester.

6 Potential User Study

As we worked on our project, we were asked to formulate what a user study would look like if conducted on our visualization. If presented with such an opportunity, we would use it to test our visualizations effectiveness at allowing administrative users to quickly identify interesting timesteps within Submittys operation. Feedback gained from this study would play an integral role not only in verifying the utility of the system we have designed, but also in allowing us to identify areas for potential growth as we assess its usability.

6.1 Hypotheses

We hypothesize that, using our system:

- Users will be able to identify areas of peak load for a course.
- Users will be able to identify areas of peak load for a worker machine.
- Users will be able to distinguish between load decrease due to errors and load decrease due to job completion.
- Users will be able to identify when a professor begins a large batch regrade of an assignment.

6.2 User Study Setup and Protocol

If we were to perform a user study using our visualization, we would want each user to be given access to a workstation containing two 1080p monitors as well as a standard mouse and keyboard. We believe that such a setup would be inexpensive to create and easy to replicate given RPI resources. One monitor would display our visualization, which would be fed a task specific dataset. The second monitor would display a simple task prompt and two buttons labeled Done, and Unable to Finish. Responses would be timed. Participants would be given a detailed introduction to our system, and would then be presented with a series of tasks that we believe our visualization should be useful at solving. A control group would be presented with the same tasks, but rather than being given our visualization, would be given the raw system logs associated with it.

For each task, the participant would be asked to complete it and then click either done or unable to finish on their second monitor. If done is selected, the user would be prompted to enter the timestamp associated with the event specified by the task prompt. The user would then be asked for a justification of their response, and if any tweaks to the visualization could have made the task easier. If unable to finish is selected, the user would be asked to identify the source of their difficulty and how it could have been rectified. Example tests include tasks such as identifying an hour of peak load for a course and machine, discovering the hour at which a particular machine went down and was unable to grade, and discovering a time at which autograding worker threads began dying on a particular machine, resulting in less throughput.

When a user finished all test, they would be presented one at a time with any and all incorrect answers that they gave along with the correct solution. After seeing the correct solution, they would be asked for insight into the source of their confusion and into any changes that could take place in our system to make things more clear in the future. We believe that such a battery of tests would yield invaluable input that would allow us to fine-tune our feature-set and provide an even more intuitive and useful final visualization.

7 Conclusion

We set out to create a useful, intuitive solution to a pressing problem within the Submitty Homework Submission and Autograding System. As Submitty has grown and become more complex, easily identifying interesting periods of its execution has become more difficult. With the addition of specialized worker machines to Submitty, we anticipate that this difficulty will only grow. In order to combat this, we proposed and implemented a stacked area visualization for use in displaying system load. Our visualization is capable of depicting total load, load by machine, and load by course. It has been extended with many utility features that make interfacing with and exploring the underlying data much easier. At present, there is still future work to be done on this visualization. However, we believe that what we have created is not only an ideal visualization for depicting our data, but also a meaningfully interactive project which will make the lives of Submitty administrators far easier.

8 Who Did What

Prior to the start of our project, Tim created the Perspective visualization that we used as our initial template. Tim also wrote the parsing utilities used to provide our visualization with a feed of data. Evan created the D3 prototype of our project and also the extended version of it displayed in Figure 3. This included adding our bubbles system, hatched batch grading areas, and focus+context functionality. Tim began the initial port of this project into NVD3.js. Evan finished this port, added the ability for the visualization to auto-update, added the ability for the chart to switch between viewing by machine and worker based areas, and added many of the cosmetic updates viewed in the final project. Tim then added support for WebSockets, allowing true streaming data rather than csv updates.

References

[1] Shu, X., Smiy, J., Yao, D., & Lin, H. (2013). Massive distributed and parallel log analysis for organizational security. 2013 IEEE Globecom Workshops (GC Wkshps). doi:10.1109/glocomw.2013.6824985

[2] Du, M., & Li, F. (2016). Spell: Streaming Parsing of System Event Logs. 2016 IEEE 16th International Conference on Data Mining (ICDM). doi:10.1109/icdm.2016.0103

[3] RSYSLOG. (n.d.). Retrieved March 27, 2018, from `http://www.rsyslog.com/`

[4] Perspective. (n.d.). Retrieved March 27, 2018, from `https://jpmorganchase.github.io/perspective`

[5] Epoch. (n.d.). Retrieved March 27, 2018, from `https://epochjs.github.io/epoch/real-time/`

[6] nvd3.js. (n.d.). Retrieved April 30, 2018, from `http://nvd3.org/`

[7] Peveler, M. Breese, S., Maicus, E., Aikens, A., Cyrus, T., Dinella, E., Anderson, J., Barthelmess, J., Lee, M., Montealegre, L., Wang, J., Holzbauer, B., Cutler, B., Milanova, A. (2018). Analysis of Container Based vs. Jailed Sandbox Autograding Systems. Retrieved from `https://github.com/Submitty/publications/raw/master/2018_SIGCSE_poster_peveler_et_al/Poster.pdf`