Project Proposal

d3tiknom

Sam Olds, Jesus Zarate
10/26/17
CS 6630

Basic Info

Project Title	d3tiknom
Github Repository	github.com/jesuszarate/d3tiknom

Name	*E-mail Address*	*UID*
Sam Olds	samolds@yahoo.com	u0742533
Jesus Zarate	jesus.zarate@utah.edu	u0816141

Background and Motivation

Sam currently works for Vivint, building a new REST service backend for a mobile app. The system is in it's infancy and could use a good deal of profiling and monitoring to know where optimization efforts need to be focused. The system is already integrated with a monitoring library, called *monkit* (godoc.org/gopkg.in/spacemonkeygo/monkit.v2), that reports various statistics every two minutes. The data is easy to obtain and doesn't (shouldn't) require any preprocessing. In fact, it would be ideal if our project could receive a stream of data to show live statistics.

The type of of data reported by *monkit* is mostly information about the duration of any function execution, in failure and success scenarios. It collects different types of metrics about these runtimes like *average*, *min*, *max*, and *most recent*. It also collects information about the call stack of these functions, so it's possible to trace the calling path.

This all means that collecting the data will be trivial and shouldn't require any additional finagling. And a visual representation of the data would be invaluable information for the performance of this project. Sam's team currently uses *graphite* (graphiteapp.org) to represent this data. But it's very challenging to use and lacks desired functionality. There are a number of features that would make the data visualizations more valuable than what graphite currently supports. An additional benefit of this project is that it won't only benefit Sam's project; it would be useful for any project that uses *monkit*.

Project Objectives

The primary question to be answered is, "What parts of the system need to be sped up?" We would like to have a visual representation of the running time of a system as a whole, as well as all of its individual parts. This can lead to improvements in the codebase, help uncover bugs, and possibly provide indication of things like memory leaks. These sorts of things are invaluable for an industry product that is expected to have hundreds of thousands of users.

Data

The data is reported every two minutes to a central service. The data can be downloaded from this central service as text, a CSV, or as JSON. Ideally, we would change the location that the server is reporting data to, so that our project can report live data points. But, as an initial step, we will just use static data. The following data is an example of the text format:

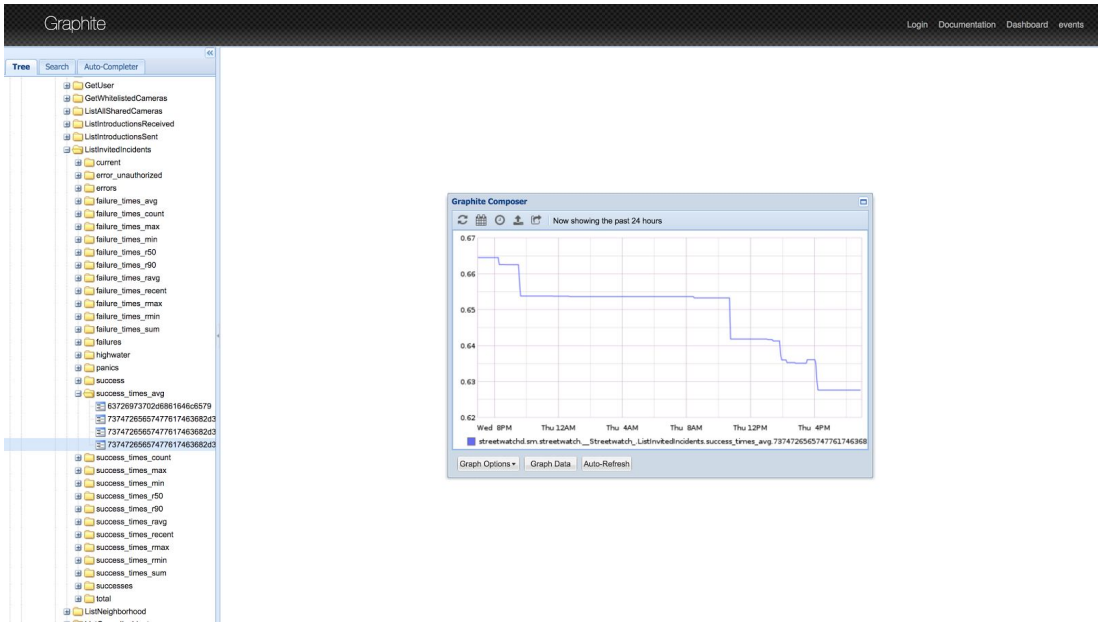
```
env.os.fds      120.000000
env.os.proc.stat.Minflt 81155.000000
env.os.proc.stat.Cminflt        11789.000000
env.os.proc.stat.Majflt 10.000000
env.os.proc.stat.Cmajflt        6.000000
...

env.process.control     1.000000
env.process.crc 3819014369.000000
env.process.uptime      163225.292925
env.runtime.goroutines  52.000000
env.runtime.memory.Alloc        2414080.000000
...

env.rusage.Maxrss       26372.000000
...

sm/flud/csl/client.(*CSLClient).Verify.current  0.000000
sm/flud/csl/client.(*CSLClient).Verify.success  788.000000
sm/flud/csl/client.(*CSLClient).Verify.error volume missing     91.000000
sm/flud/csl/client.(*CSLClient).Verify.error dial error 1.000000
sm/flud/csl/client.(*CSLClient).Verify.panics   0.000000
sm/flud/csl/client.(*CSLClient).Verify.success times min        0.102214
sm/flud/csl/client.(*CSLClient).Verify.success times avg        1.899133
sm/flud/csl/client.(*CSLClient).Verify.success times max        8.601230
sm/flud/csl/client.(*CSLClient).Verify.success times recent     2.673128
sm/flud/csl/client.(*CSLClient).Verify.failure times min        0.682881
sm/flud/csl/client.(*CSLClient).Verify.failure times avg        3.936571
sm/flud/csl/client.(*CSLClient).Verify.failure times max        6.102318
sm/flud/csl/client.(*CSLClient).Verify.failure times recent     2.208020
sm/flud/csl/server.store.avg    710800.000000
sm/flud/csl/server.store.count  271.000000
sm/flud/csl/server.store.max    3354194.000000
sm/flud/csl/server.store.min    467.000000
sm/flud/csl/server.store.recent 1661376.000000
sm/flud/csl/server.store.sum    192626890.000000
...
```

Data Processing

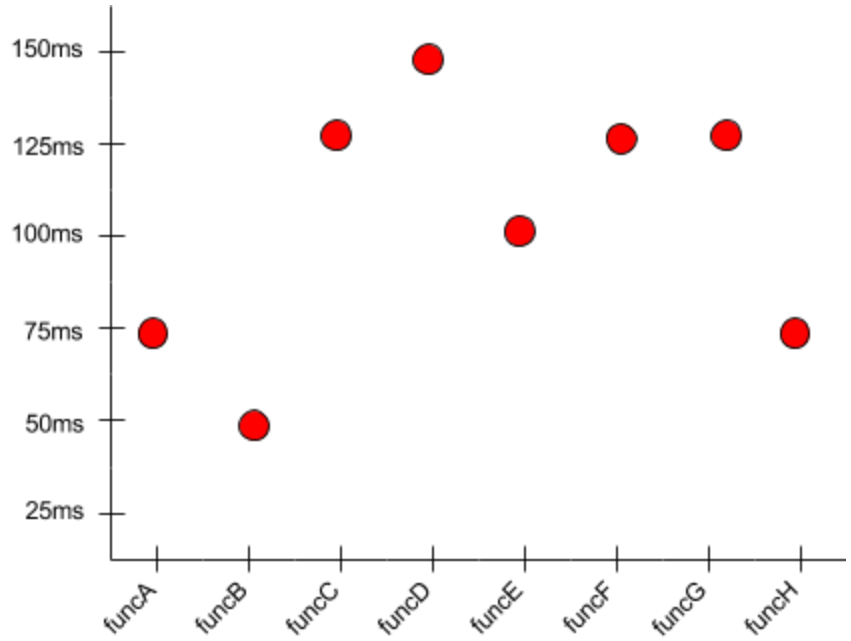
We anticipate to have no data processing. The data that we get from *monkit* comes in several different formats like we mentioned above. We plan on downloading the data in the JSON format, which will allow us to easily access the name of the functions, and the execution time of the functions.

Visualization Design

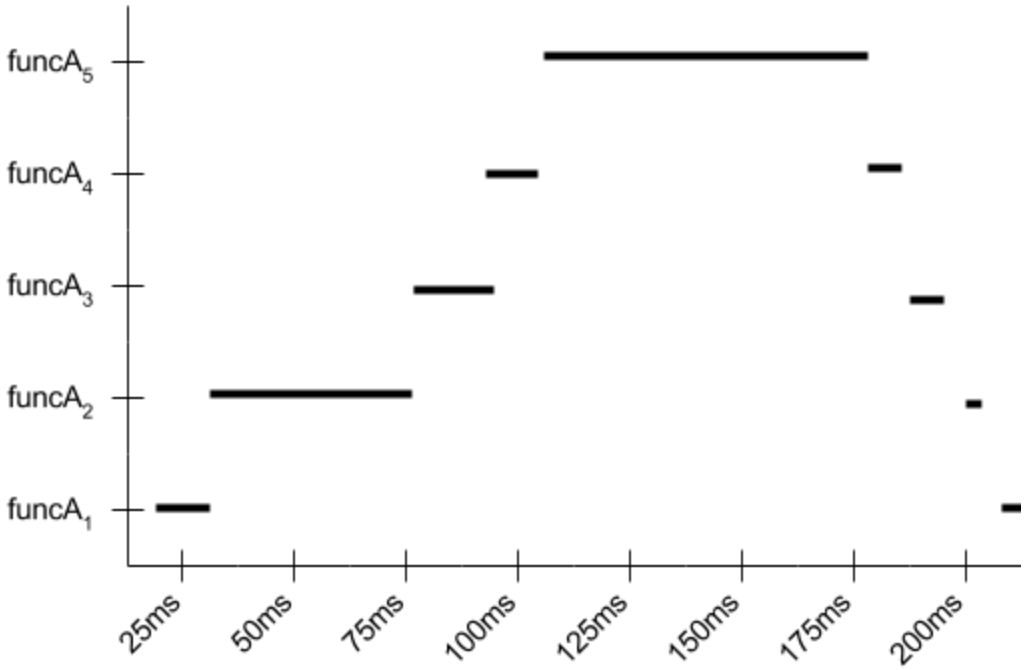
The current tool used to visualize the dataset, *graphite*, provides this UI. It's not great; it's hard to compare the runtime of various aspects of the system, and it tends to just show runtime over time, not the averages of multiple functions at the same time.



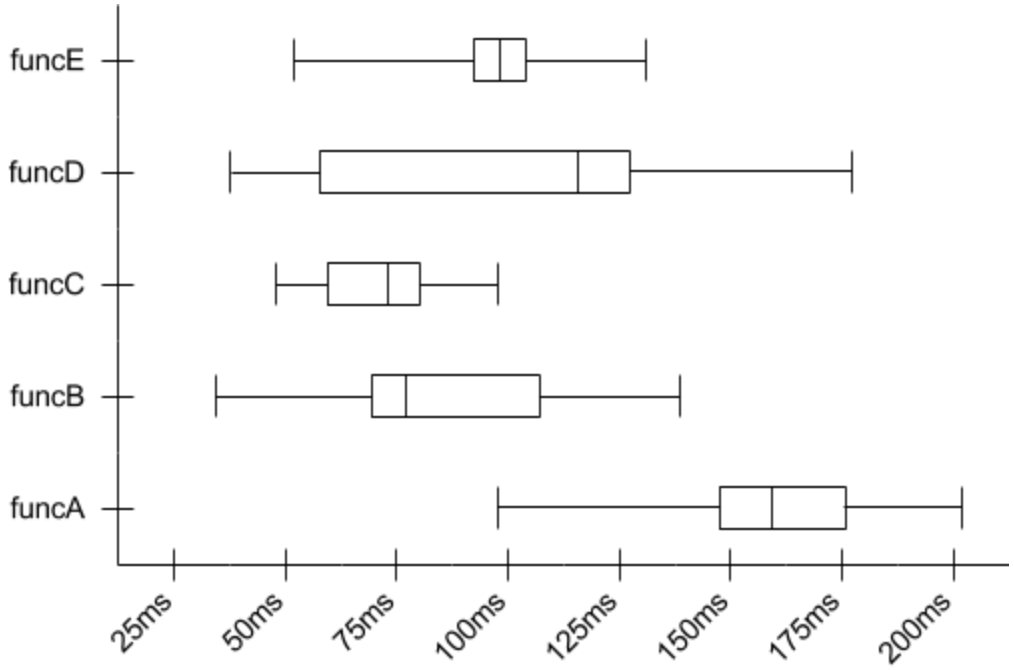
A possible comparison chart: This type of chart will allow the user to quickly spot the functions that are taking the most amount of time, as well as be able to spot trends at a specific point in time.



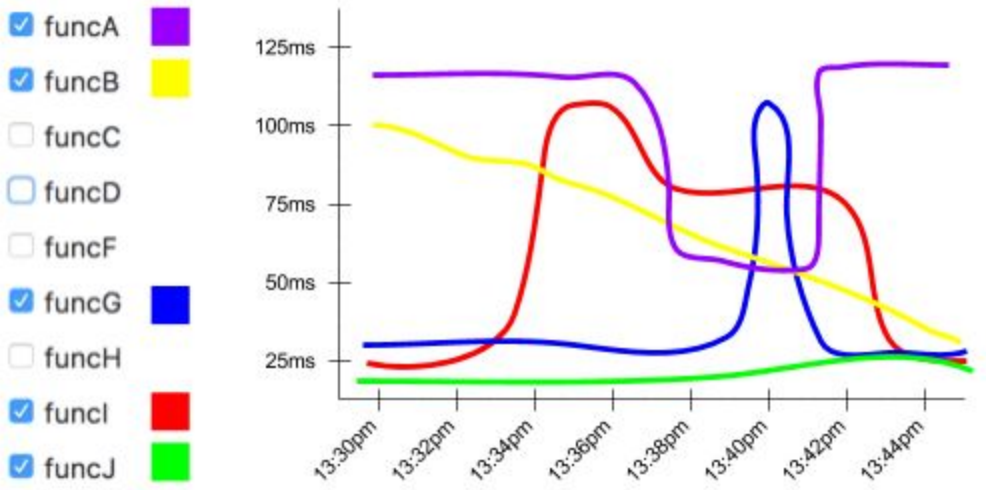
A possible call stack trace timing chart: This chart will not only allow us to see the stack trace, but at the same time it will allow the user to spot which point (function) in the process is taking the most amount of time. This would aid the user in determining the exact long running function.



A possible box plot comparison: The box plot will allow users to see all the time types, rather than only one, which will be beneficial in determining if it's worth digging in deeper into what a function.



A possible comparison of average run times over time: This line chart will show the average run times of various functions over time. This will be useful for showing which functions can perform well under heavy load, as well as when those peak usage times occur.



Must-Have Features

- **Runtime Type Selection** - Be able to switch between runtime types. I.e. compare the average time of the function or the min/max times.
- **Function Selection** - Selecting individual functions in order to compare them isolated from the rest.
- **Datapoint Comparison** - Have the ability to compare different datapoint using different visualizations.
- **Interactive Data Points** - Being able to click on a datapoint and expand on the information of that datapoint. I.e clicking on a datapoint will provide the stack trace of the function belonging to the point.
- **Runtime Metric Over Time** - It's useful to know how the average/min/max runtimes are changing over time, so a plot that displays this data from a selection of functions would be important.

Optional Features

- **Stream live data** - Since we have access to live data we would like to have a feature that will allow us to see the data in real time.
- **Global Visualisation** - Having a single visualisation that will show all of the data that we have available. One option would be to use a heat map that will have a color scale to encode the total runtime of every function.

Project Schedule

Week1 (October 29 - November 4)
● Gather data required ● Clean and structure the data if necessary
Week2 Milestone Due (November 5 - November 11)

● Structure the project
● Implement simple visualizations:
  ● Scatter plot
  ● Line graph
  ● boxplot

Week3 (November 12 - November 18)
● Organize and unify the visualizations, and pick the visualizations we will pick for the final version
Week4 (November 19 - November 25)
● Implement the stack trace visualization ● Start implementation of program to get the live data feed
Week5 Final Due (November 26 - December 1)
● Incorporate live data stream with existing visualization