Gorouter Throughput Performance

This document summarizes results of a performance test run against Cloud Foundry's L7 HTTP router, known as Gorouter. It may also contain a comparison of two sets of results.

The test results in this report can be reproduced, and this report regenerated by following instructions in the README for Routing Performance Release (https://github.com/cloudfoundry-incubator/routing-perf-release).

Click the button 'Show Code' below to toggle the display of the code used to generate the graphs below. To compare two sets of results, set `compareDatasets = True`.

Test Description

This report provides results of a long-running "ramp up" load test in which a fixed number of requests is sent to a static app through Gorouter from one client thread. The number of concurrent threads are then incrementally scaled and at each step the same number of requests are sent across all concurrent threads. As concurrency increases, the time required to send the same number of requests is reduced.

The tests were run against a standalone Gorouter and a static app running on a raw VM, not against a full Cloud Foundry deployment. The test environment was comprised of two deployments on AWS. We elected to do our testing on Amazon Web Services (AWS) for its network stability.

- We use a modified cf-release manifest to deploy Gorouter and NATS only. Gorouter is run on our default VM type of `c3.large`.
- Load was generated using the Routing Performance Release (https://github.com/cloudfoundry-incubator/routing-perf-release), which also includes the static backend and a job that registers a configurable number of routes for the backend.

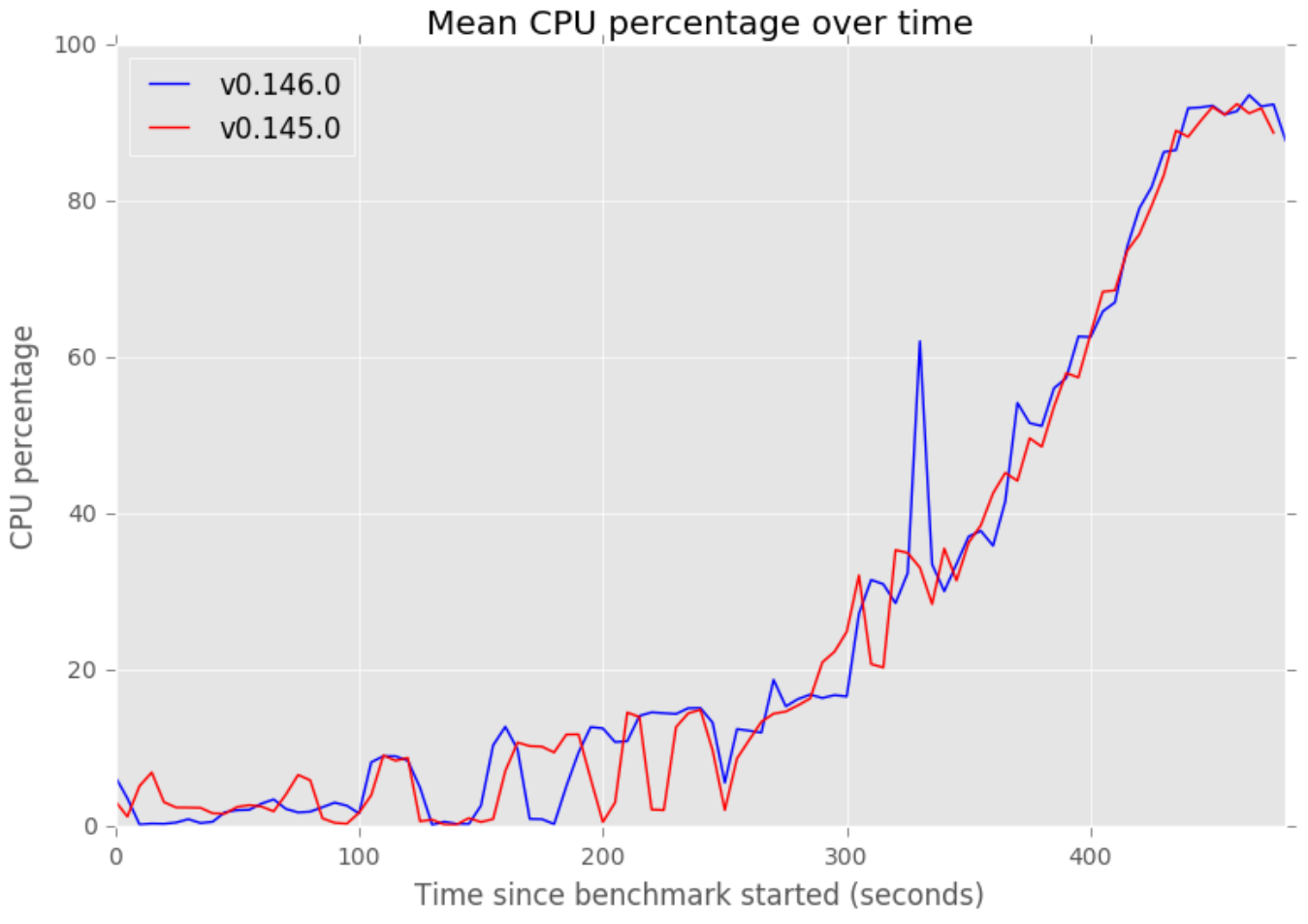
Gorouter CPU Load over Time

The following plot shows a summary of the CPU load over the duration of the test run. Here are some key points to note when looking at this data:

- Each CPU sample collected is highly variable; this is characteristic of CPU load on Gorouter when it is handling requests. For purpose of graphing, the test results have been downsampled in order to smooth out variations that happen over a short periods of time. The mean of each sample is represented in the plot to depict the average CPU load over time. As an example, while the plots may plateau before reaching 100% CPU load, actual CPU load during the sample may have fluctuated between 80% and 100% CPU load.
- When we test with a large routing table, as when `http_route_populator` is configured to register 100,000 routes, we see many tall spikes that show up periodically every 60 seconds. These spikes represent processing of the large number of route registration messages. When `http_route_populator` is configured to register only one route, fewer spikes are present.
- It's possible for the plot to be cut off before the edge of the graph. This indicates where the performance test run finished.
- There may be many jagged small spikes throughout the plot. This corresponds to the starting and stopping of the load tests each time concurrency is incremented.

To avoid degradation of service (significant increases in latency) Gorouter should scaled vertically and/or horizontally before CPU approaches maximum. The graphs below demonstrate that as CPU approaches the plateau, latency begins to increase to the point where throughput plateaus. Past this point throughput does not increase with added load; this only results in added latency. In general we recommend scaling Gorouter when CPU reaches 60-70%.

`Out[5]:` The following CPU data is sampled over 5 second windows.

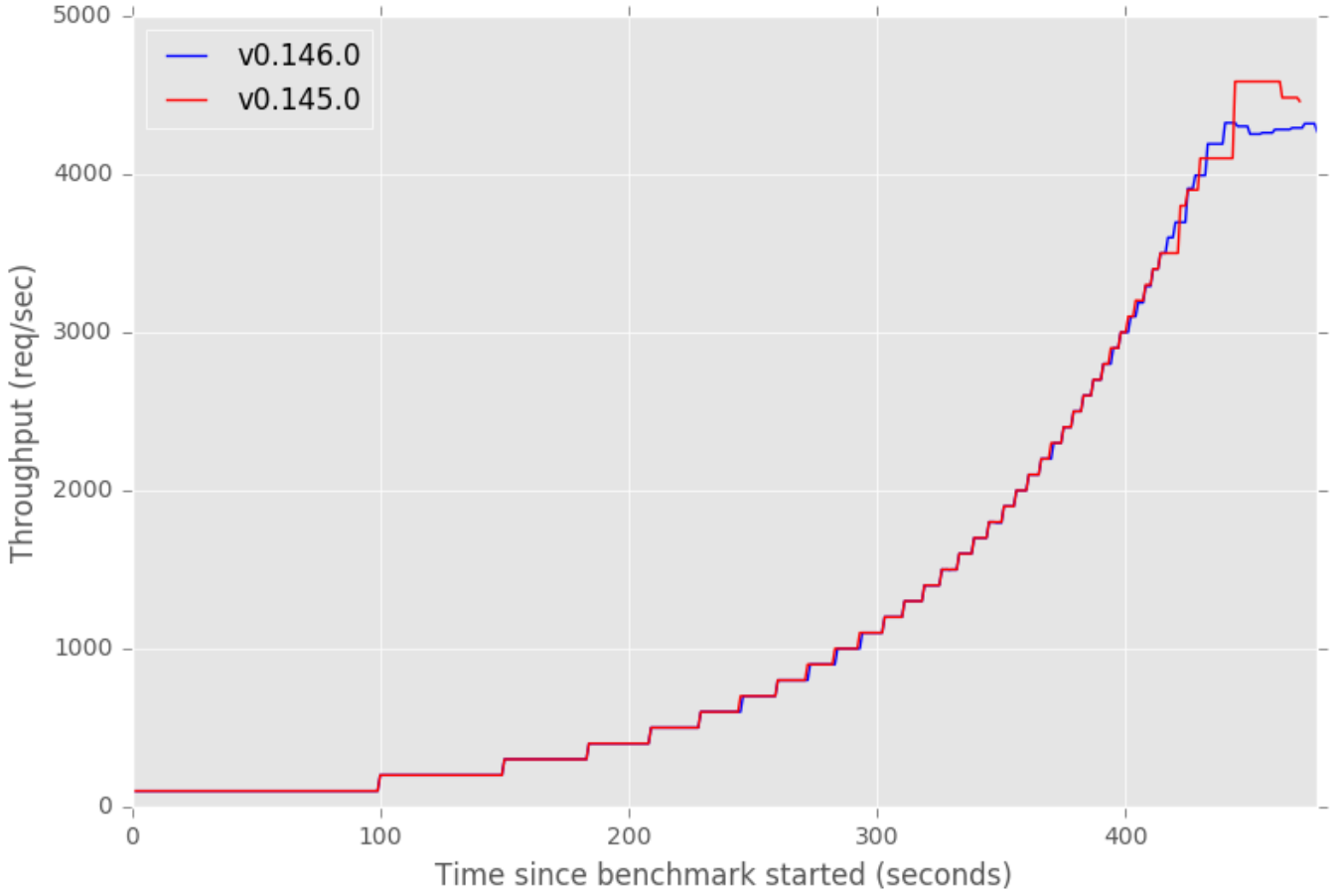


Gorouter Throughput over Time

The following plot shows a summary of throughput performance over the duration of the test run. Here are some key points to note when looking at this data:

- When we test with a large routing table, as when `http_route_populator` is configured to register 100,000 routes, we see many spikes and few troughs in this graph. These represent processing of the large number of route registration messages. When `http_route_populator` is configured to register only one route, far less variability is observed.
- It's possible for the plot to be cut off before the edge of the graph. This indicates where the performance test run finished.
- The throughput plot ramps up as concurrency is scaled and then plateaus. The point at which throughput plateaus should correspond with where CPU plateaus.

The throughput level at which the plot plateaus is the absolute maximum throughput that each Gorouter VM will be able to handle. As can be seen in the graph below "Headroom Plot", by the time throughput reaches this point latency will have spiked to unacceptable levels. See comments above for "CPU Over Time" regarding scaling Gorouter to prevent degradation of service.



Throughput Headroom Plot

This graph plots throughput versus latency to illustrate the cost of latency as throughput approaches a limit. This graph provides Gorouter's maximum throughput to remain within a given target latency. The CPU graph above shows the CPU load at this throughput. An operator can monitor CPU, as well as latency, and scale Gorouter vertically or horizontally to remain within latency targets as application request load increases. In order to target throughput for application request load, divide the targeted throughput number by the throughput at the target latency and round up to the nearest whole number. This is the minimum amount of Gorouter instances (`c3.1arge`) necessary to meet performance objectives.

For example, in order to achieve a targeted throughput of 10,000 requests per second at 20 milliseconds of average response time (latency), and if the point in the Headroom plot corresponds to 2,500 requests per second (throughput), then at least four Gorouter VMs are required to meet performance objectives.