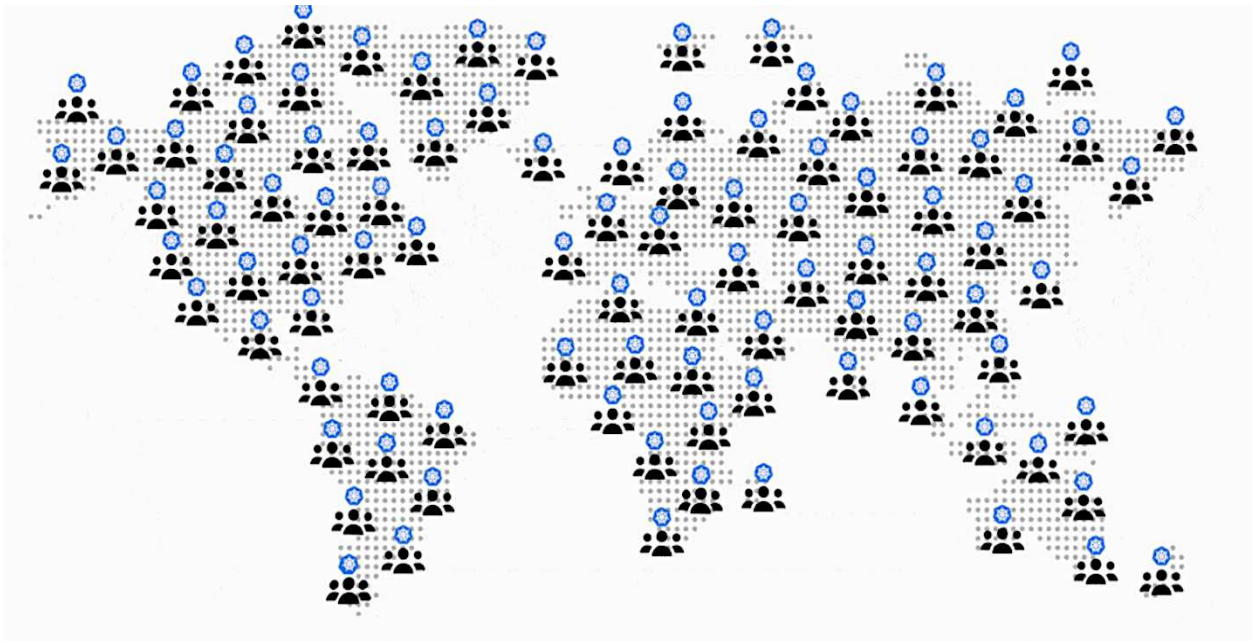
RAFAY



How Does Latency Affect Web Application Response Time?

By John Dilley
Chief Architect, Rafay Systems

Introduction

Response time matters to any organization that sells products or services over the Internet. Back in 2009, Akamai published a study that concluded consumers become impatient when web pages take longer than 2 seconds to load and that businesses lose sales when their e-commerce sites underperform. Today, web pages and applications are far more personalized, interactive and chatty than they were a decade ago. They are also bigger; the average web page size in 2018 was 1711.4kb compared to 467.7kb in 2010, according to data collected by the HTTP Archive. Now, more than ever, fast response times - or conversely, low latency - is of utmost importance.

Web page response time is the total time a browser takes to fetch and render a page. Before dissecting the components of that transaction, here is a quick review of some network queuing theory to formalize a few terms:

- Network Latency: The minimum (latent) time a packet takes to get from here to there, typically expressed in terms of round-trip time (RTT), which is the delay there and back.
- Queuing Delay: The time a packet spends waiting in a queue, perhaps in a router as it moves from network to network. Or delay waiting for access to resources on a server. Let's call it QD.
- Server Residence Time: How long the server takes to compute and start sending a response back. This includes generating dynamic content, looking up the response in a cache, or whatever. Let's call it CPU.

As a developer, the residence time is your code doing its thing. You work to make this fast, and this time dominates the overall response time in desk and local network testing. Only when pushed to the open Internet do you really face latency and queuing delay.

Looking in detail at what your browser had to do to fetch the page, and focusing in on the network queuing theory reviewed above, here are the steps for the first fetch of the page:

- Look up the server hostname to resolve its IP address – 1 or more RTT to DNS (more on this later).
- Open a TCP connection to the server – 1 RTT plus queuing delay (QD) if the network is congested.
- Establish a secure TLS session – 1 RTT (plus maybe QD) plus crypto work on the server (server residence time in the security library).
- Send the web request to the server – 1 RTT to send the request and receive an acknowledgment (TCP ACK) that it arrived.
- Compute the response – 0 RTT but server residence time for the web server to look up the page in memory cache or fetch it from disk or a remote server if it's not in cache.
- Send the response – multiple RTT, one for each TCP congestion window of content.
- Fetch embedded objects required to render the site correctly ... for example CSS stylesheet, JavaScript, site image content, ads, etc.

Each of these embedded objects further require:

- DNS lookup - if it's not the same hostname.
- Open a TCP Connection to the server – 1 RTT.
- Establish TLS connection – 1 RTT + crypto CPU.
- Send the request – 1 RTT.
- Compute the response – 0 RTT + CPU.
- Send the response – multiple RTT... just like the base page above.

Thus, to render the page you need many responses. Your browser will make some requests in parallel; each new connection requires the TCP and TLS handshake before they get any data.

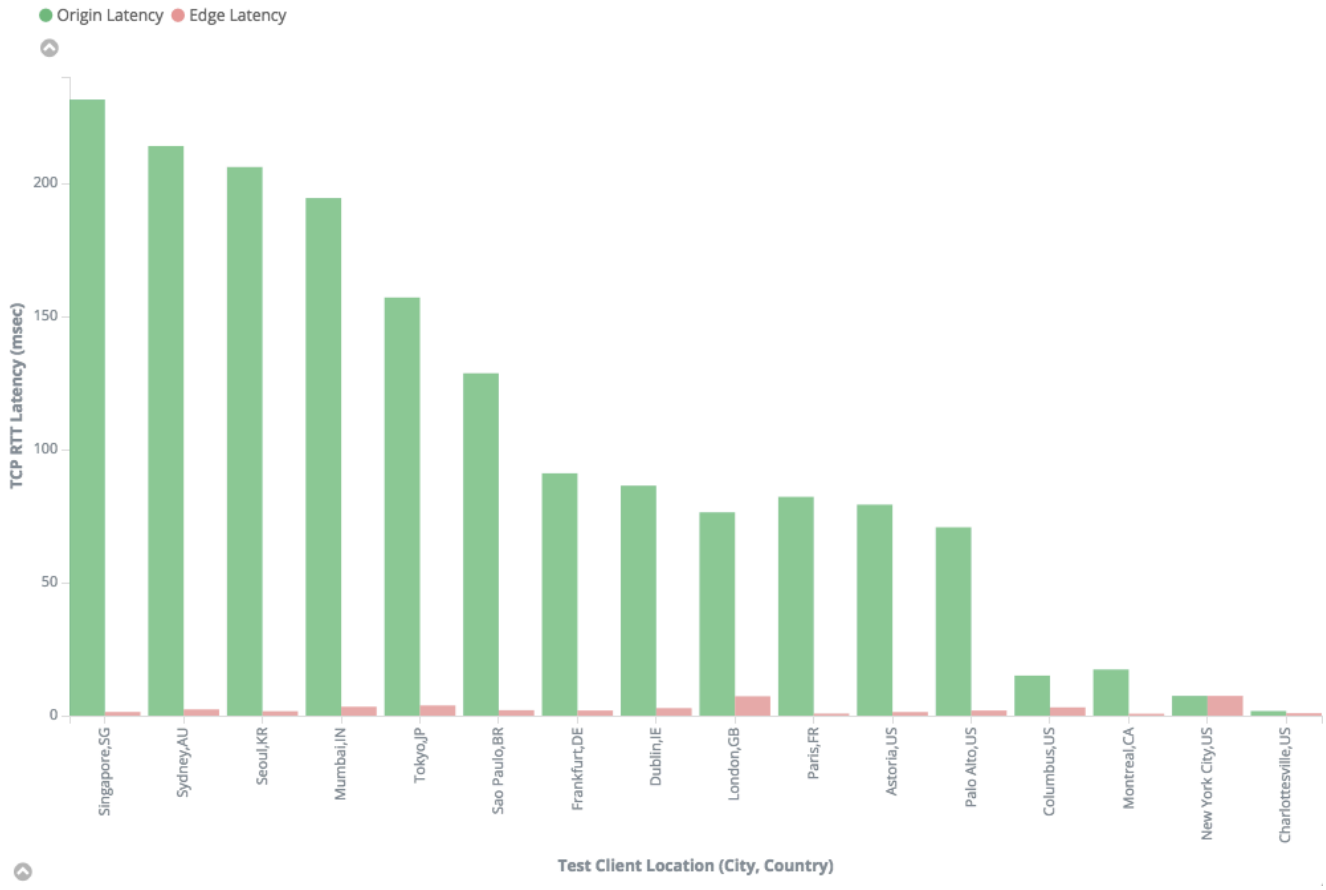
The "multiple RTT" on the response reflects that the TCP stack can only send one TCP window of data before it has to wait for acknowledgment that the client has received some or all of the data sent. You can't have more than one window of data in flight (per TCP connection). The window typically starts out small and grows to perhaps 64 KB. Servers and fancy clients can turn the window up further, but unless you break TCP you have to start smaller and grow. This is to prevent fast senders causing congestion, queuing delay and packet drops.

A hypothetical blog post takes 80-100 network requests including images, some JS and CSS, and four or so HTML objects across 15-20 domains. Some of the objects are over 100 KB. The total round trips for the page to load is well over 100. Multiply the RTT by 100 round trip object fetches and it really adds up!

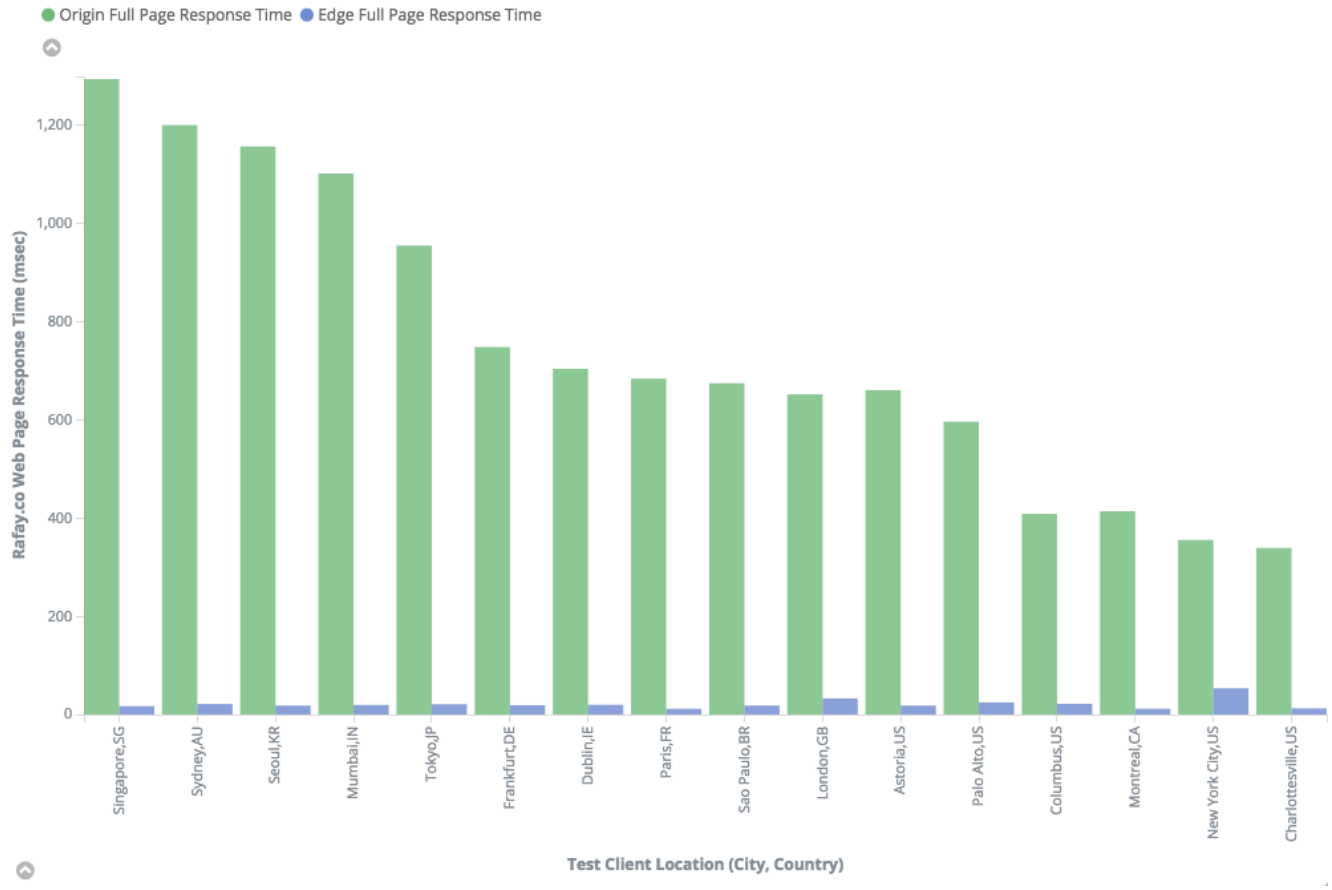
At 20 msec per RTT, it's two seconds. At 100 msec, the page would take ten seconds to load, well above most people's tolerance according to the Akamai study. Going from 20 to 100 msec delay can turn an acceptable page load into one that people may browse away from.

Network Latency Examples

The first graph below shows the network latency, client-to-edge and client-to-origin, for a web server tested from client locations listed on the horizontal (x) axis. The green bars show client-to-origin network latency, coral shows client-to-edge latency. From Asia and South America, we saw over 100 msec RTT to the origin server, down to about 10 msec in VA, where the server is evidently located.



The second graph below shows the full-page response time – and clear correlation between the latency in the first graph and the full-page response time, where the endpoints in Asia took a full second to load a simple (44 KB static) base web page from the origin. Note that the worst-case, full-page response time is well over a second -- even though network latency was at most 100 msec. From the edge, page response time was consistently under 100 msec world-wide.



A website that requires many round trips from the client to authenticate, retrieve dynamic personalized content, or provide other computation should see performance gains at least as strong as the case above.

Conclusion

Of all the components that comprise a network transaction, network latency has perhaps the greatest potential for significant negative impact. RTT can often make the difference between a great and lousy web experience for end users. This is where Rafay Systems can be of benefit.

Applications owners and DevOps teams now have the ability to run their applications closer to users and end points. Rafay Systems enables developers to automate the distribution, intent-based scaling, and operations and management of containerized applications across public and private clouds, and service provider networks. Rafay's platform intelligently, dynamically places and operates containers in locations based on a variety of policies including latency targets, geo-fencing, and time-of-day. The platform's innovative, compute-forward feature set is delivered as a service. It includes a full suite of developer-friendly tools to help run containerized apps globally.

With Rafay Systems, you can eliminate the need to build complex, costly infrastructure and expertise in-house. Application owners can use Rafay to drive additional revenue opportunities by instantly rolling out new services. Developers and DevOps teams can easily integrate Rafay with existing CI/CD pipelines to automatically push containers anywhere. Operations and Site Reliability Engineering (SRE) teams can leverage Rafay to automate the placement and operation of containerized microservices close to user populations.