2015 OpenSIPS summit – Austin, Texas

November 9th-10th, 2015

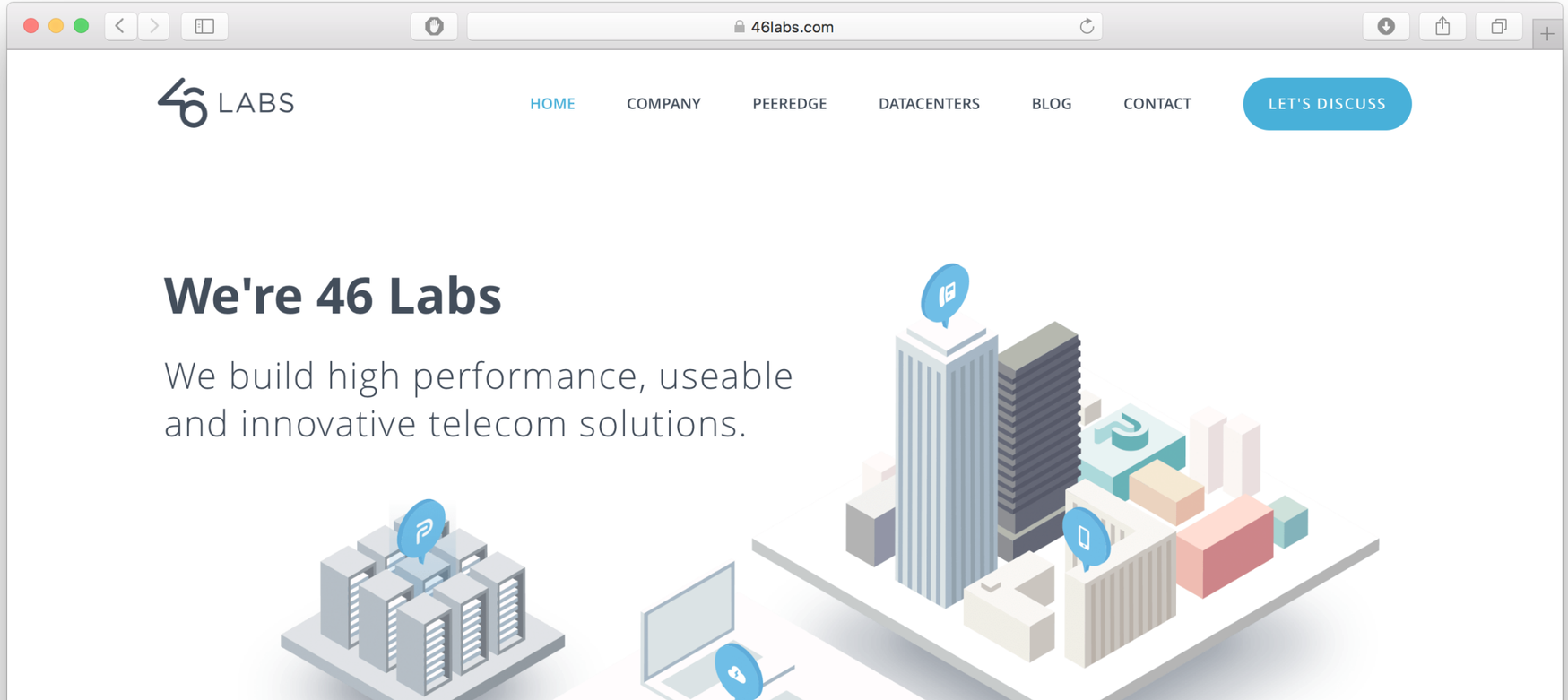
Trevor Francis

Founder and Chief Architect

46Labs.com

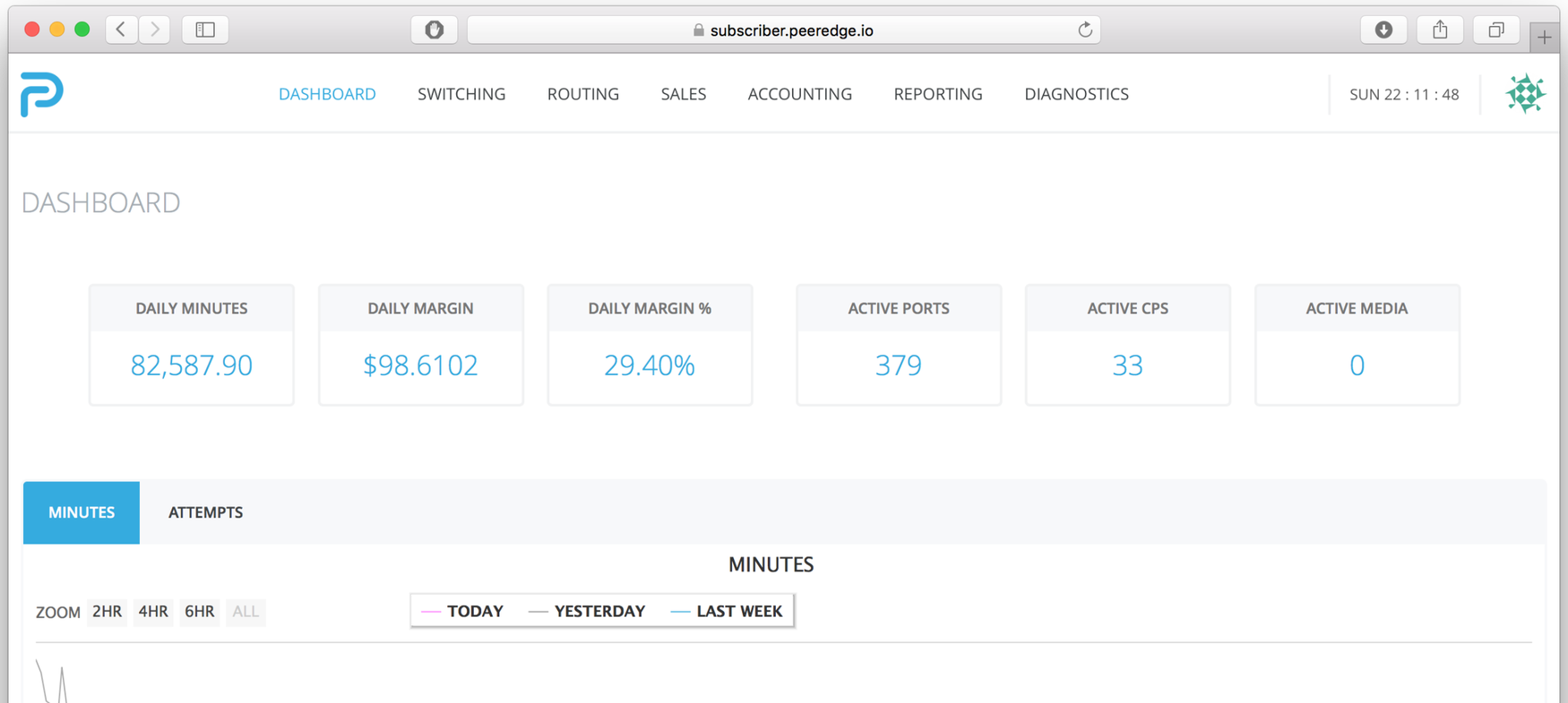
Who is 46 Labs?

We are a solutions and infrastructure-as-a-service(IAAS) provider to telecom carriers and large enterprises.



What is 46 Labs known for?

We built **Peeredge**, one of the world's largest fully-integrated wholesale telecom switching, routing, rating and analytics solutions that can be deployed standalone or used to augment and control existing infrastructure.



It can scale (and often does) to an unlimited amount of inbound and outbound traffic. Currently, in production, we are routing nearly 10,000 calls per second of inbound traffic and growing.

It is my intent to illustrate what we are doing right now on a few of our production tiers and then abstract out where Opensips fits into each of these.

Switching (Hybrid Proxy / SBC)

But that is only ½ the equation. These 10,000 CPS end up generating, on average, 3 outbound calls for every inbound call. This means that we are closer to 30,000 CPS in production. Each of these inbound calls may take a completely unique path out of our infrastructure, depending on the logic our customer has fed into the system. Furthermore, one of the benefits of our platform from a customer perspective is that it doesn't matter what inbound path a call takes to originate at our infrastructure, we are able to control and throttle both the ingress/egress CPS but also the dialog counts.

Consider that we have over 5k trunk groups in production throughout our platform. Each of those trunk groups chit-chat inbound/outbound dialog limits, inbound/outbound CPS limitations, blocklists, loop-detection, etc. . So 5k trunk groups x 6 variables = 30k variables considered x 30k CPS = 900 million go/no-go decisions per second. Our ratelimit threshold is per-second (due to the "Second" in Calls-Per-Second).

So, we broke every cachedb backend that Opensips had to handle distributed rate-limiting.

If you also consider that our routing engine typically considers over 125 terminating endpoints, each with at least ½ a million rates, a single inbound call will consider around 72-million potential terminating routes before making its decision on the most optimized path for a call (we make multi-variable routing decisions, not just based upon cost)

10k CPS x 72-million routes = 720 Billion route considerations/second.

Rating / Analytics

With every inbound call generating a single CDR for the A-leg, and 3 CDRs per second on the B-leg, we generate about 40,000 CDRs per second of exhaust data, that must be fully aggregated and data warehoused.

At this scale, post processing of CDRs and analytics data is not possible. So we stream the CDRs through our patented (USPTO US 8817652) analytics processing engine and both the statistics and raw CDR data is available in real-time to our customers.

Each CDR generates around 100 aggregate data points of information, so we are processing, on average, 4 million aggregate data points per second.

Yes, Redis, Couchbase, Memcached, Cassandra, Mysql, Postgres, Hbase, Mongo, Riak, DynamoDB, and nearly every new-fangled in-memory DB....you name it..is too slow, not redundant enough, not accurate enough or simply too lazy to complete the task. We know, we tried all of them and most of them together in some way.

We were forced to build our own solution to handle our scale, although as many of you know, we use Cassandra for our data warehousing needs. We are also big supporters of that project, especially considering Datastax was started by two guys here in Austin.

1. High-scale infrastructures are allergic to physics and time.

Considerations:

- If switching gets slow, re-transmissions happen. When re-tranmissions happen, CPS stack on top of each other and 10k CPS now looks like 30k CPS. The gears grid to a halt.
- If analytics gets slow, CDRs just sit and stack and stack and stack. At 40k per second, it takes 25 seconds to amass 1 million CDRs. If you don't have a way to buffer, then shared memory tanks, OpenSIPS crashes and all of those glorious billing records vanish. Poof, gone!....just like your customers.
- If routing lookups get slow, they stack until they starve Opensips children. Opensips stops responding to anything. Invites are ignored (then retransmitted), Byes are ignored...causing billing inaccuracies and, while Opensips is still alive...it mine as well not be. If you load balance to the servers doing the heavy work (like most horizontally scalable solutions do), those load balancers will cannibalize your backend infrastructure with re-transmissions and backend server by backend server will lay down.

Lesson:

Use physics and time to your advantage. Eschew obfuscation in your architectural approach and always ask yourself how can I break this.

2. If there is a flaw, we will expose it…..and usually in production.

Considerations:

In the last month, we were responsible for exposing significant shared memory and package memory leaks that took 2.1 to its knees. None of these issues were present in testing, at any scale. However, when you throw thousands of different UAS/UAC combinations at a solution, coupled with every type of broken signaling imaginable and blast it with thousands of calls-per-second, behind the curtain actors hop out and start playing a staring role.

I can, however, report that our roll-out of 2.1 is currently rock solid thanks to the efforts of Razvan, Bogdan and Vlad coming to our rescue at such short notice. If you are running 2.1, it is in your best interest to pull and deploy master because it has a ton of new important fixes.

Lesson:

If you run at scale, a “stable” version is never stable until you make it so. Expect very little and your expectations will be met. Our friends at Cassandra always recommend not rolling something into production until it has 5 subversions.

3. Hammer testing anything we build with SIPP is useless.

Considerations:

A distributed SIPP (because we break SIPP before it breaks us) testing scenario shows that we can handle over 1 million calls-per-second. Oh the beauties of synthetic traffic! It also tells us that we are handsome and very talented.

Lesson:

Synthetic benchmarks are just that and have no bearing on real-world performance.

4. When you have exhausted every method of testing at your disposal, you are 50% of the way there. We call this the 48-hour rule.

Considerations:

I cant count how many times we have deployed a solution that performs well through peak and then lays down in an off hour, or a patch that breaks something else.

Lesson:

Never deploy anything that cant be rolled back and never call anything “fixed” until 48-hours of production traffic has run over it, regardless of your level of confidence.

Where does Opensips 2.1 fit in Peeredge and why?

Switching

As I mentioned in the previous section, our scale broke any distributed database backend for rate limiting. We would simply overwhelm the backend, OpenSIPS would stall after all of the children were starved and everything would grind to a halt.

Having done considerably rate limiting inside of Opensips 1.X for the past three years, we knew that the issue was in the distributed exchange of information, not in OpenSIPS' ability to do high-scale rate limiting. We took that knowledge and commissioned the expansion of the ratelimit module to use the binary interface for chit-chatting. In this rev, the actual counting is done within Opensips and an aggregate value is exchanged between servers, over the binary interface. So, instead of every increment/decrement or set hitting an external DB for every call, that is done in memory. Combine that with 2.1's async abilities and we now have a solution that no longer hangs children and allows finite limiting.

While technically going into 2.2, it will compile fine into 2.1:

https://github.com/OpenSIPS/opensips/commit/842588f52010e686d63f12321477467ece9a3a52

Where does Opensips 2.1 fit in Peeredge and why?

Routing

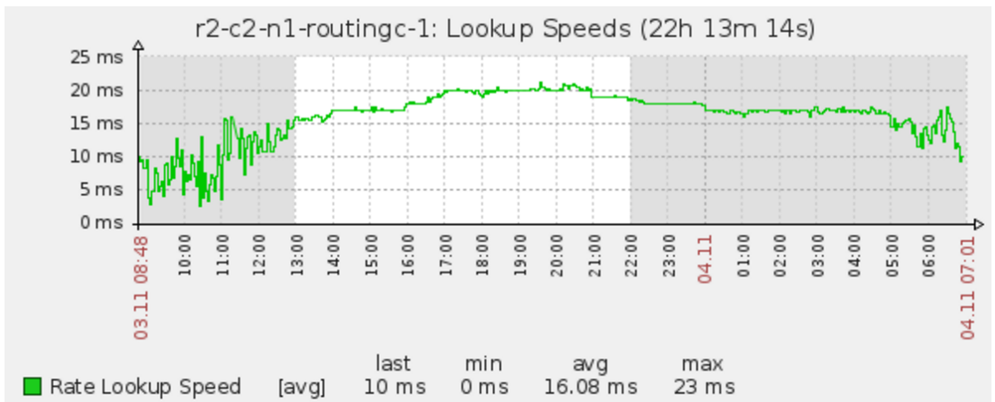
The biggest benefit we got out of Opensips 2.1 from a routing perspective was its async database operations. It is important to note that DBVIRTUAL is not supported in Async mode, but with our architecture that didnt pose us a problem.

Given our scale, any slowdown in database operations can grind our infrastructure to a halt, which is why we keep a close eye on our database routing speeds to make sure switching operations complete smoothly. Any spike in database speeds (even temporarily) can cause a significant impact on non-async database operations and hang OpenSIPS children. For instance, in 1.x if our DB operations went from 15ms to 30ms per lookup, it would hose an OpenSIPS box because there is only a finite amount of children allocated per physical server.

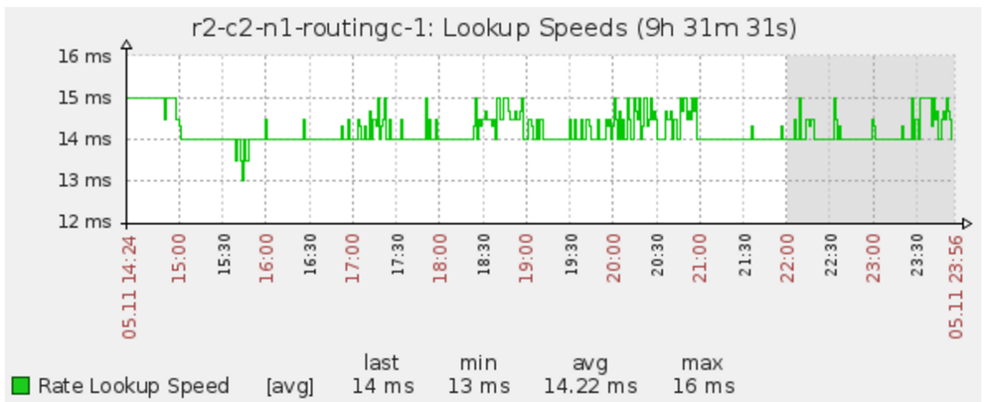
We also noticed that non-async operations reported increased lookup speeds under load, versus non-loading. This did not comport with our synthetic benchmarks of lookup speeds. This was because, at scale, we would max out the limitations of the OpenSIPS children and lookup speeds would slow by a few milliseconds. It also meant that OpenSIPS was the bottleneck and not our database servers.

Where does Opensips 2.1 fit in Peeredge and why?

See Graph



By enabling async operations inside of opensips, this all changed. Routing speeds matched synthetic benchmarks and temporary spikes in lookup speeds caused by changes in the rate structure were buffered by the asynchronous db connections.



Notice the change in the curvature of the graph and the outlier max lookup speeds:

Where does Opensips 2.1 fit in Peeredge and why?

Furthermore, in non-async mode, DB operations were limited to a single DB connection per child. Assuming you had 16 OpenSIPS children and your lookup speeds were 25ms, the maximum calls per second for that box (assuming it isn't actually handling signaling!) is 640. If you consider that the children have to do more than just perform lookups, you are looking at more like 320 calls per second.

With async database operations enabled, proper tuning of the OpenSIPS children and "db_max_async_connections," you can improve the density of OpenSIPS to match that of your database. This means that OpenSIPS is now a lot more tolerant to spiky database performance and allows you to load OpenSIPS up much heavier without over-allocating children, causing excess context switching.

Where does Opensips 2.1 fit in Peeredge and why?

Analytics

The exhaust data that we create in our platform must not only be never lost, but also be idempotent from a reporting perspective. Overcounting or undercounting isn't an option with what we do because CDRs serve as billing records for our customers. This means that resiliency and accuracy must be favored over speed. Since we have our own analytics backend, the native OpenSIPS db backends don't work for us. To get the exhaust data out of Opensips, we utilize the eventrabbit module and have since it was released.

The previous problems with the module, in its native form, is that it doesn't support failover, you cant define exchanges on the fly and if the single rabbitmq server fails...records are lost. This was not acceptable to us. To address this, we had the module expanded to allow multiple rabbitmq servers, failover, multiple exchanges (which can be defined on the fly), and dumping to a file in the event that all rabbits go missing.

While I don't see a public commit of these changes to the main source branch, we have nevertheless approved its re-release back into the community and it is working with 2.1 in production. If you want a copy of the module branch, message me and I will see that it finds its way to you.

We have experienced multiple occasions of hardware failures causing RabbitMQ servers to be unavailable. The changes in the module allow for load balancing/automatic failover, which allows us to reliably scale the linkage between our analytics solution and Opensips.

Questions?

www.46labs.com
trevor.francis@46labs.com
+1-512-831-3664