iVoyeur
ChatOps

DAVE JOSEPHSEN

Dave Josephsen is the sometime book-authoring developer evangelist at Librato.com. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

Once upon a time, not too long ago, I was locked in a small room with a horde of telephone salesmen. They were quite enamored of VOIP, and SIP phones, and MPLS, and together with one of the executives, they were convinced that telephones (yes, telephones) were the final and ultimate solution to every productivity-related problem in the company.

They had a grand vision to unite all workers of the company by attaching them (nearly symbiotically) to a device called a "SIP phone." The SIP phone would sit on everyone's desk and inform everyone of everyone else's status via the blue-hued LED magic of something called a "presence protocol."

Everyone, I was told, would log into their phones first thing every morning, and they would message each other via phone protocols, as well as transfer files to each other this way. Everyone would always know the moment everyone else logged in to work, and because everyone would do everything through their phone, pie charts (yes, pie charts) could be produced that detailed the work habits of the whole company (except of course the execs). The executives really liked pie charts, so they had locked the Ops guys (us) in a small room with the telephone salesmen for six hours (yes, six) to teach us about presence protocols and explain things like how the telephones would replace email and monitoring systems, and how the new system would pay for itself in months once the people were all wired up to the phones. We were evidently expected to do the wiring; honestly, I wasn't looking forward to it.

Meanwhile, on the West Coast, so many companies were in want of smart people that they had snatched up every smart person out there and were actively sending out spies to capture and import more. Some of them had even resorted to stealing smart people from each other, and paying college kids to drop out of school. Other, more respectable, West Coast companies realized that they might be able to use smart people from other parts of the world without shanghaiing them, if they could just figure out how to handle remote workers, so they started exploring the concept of "distributed teams" [1]. Oddly, they collectively came to a vastly different conclusion about the best way to manage their employees—one that did not involve anything like plugging everyone into a phone.

The emergent "distributed teams" phenomenon is based on asynchronous communication, like persistent chat systems, and flextime to maximize individual productivity. Persistent chat is a lot like Web-based IRC, except you get scroll-back of all the conversations that have been going on even while you were disconnected (hence, persistent). These tools integrate easily with other tools and services, especially operations-focused tools, and so they've become a popular solution to centralize operations undertakings, like troubleshooting, and software deployments via chatbot.

Collectively referred to as ChatOps [2], this loosely collaborative alternative to approaches like the phones described above is often referred to in a theoretical way on this blog or that [3, 4] these days, but how it works in practice is not well documented and is something you really

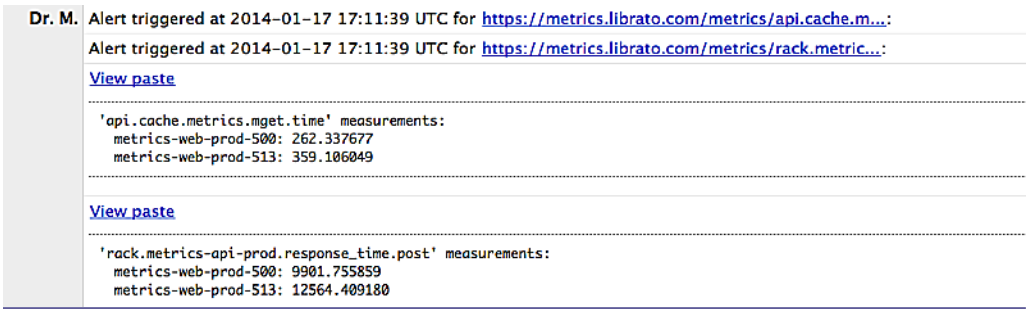

Figure 1: Initial notifications of API problems appear in our Ops channel.

Figure 2: Follow-up alerts from our log processing system appear in the Ops channel.

need to see in the wild before you comprehend just how nice it is. So, having just joined the distributed workforce, I thought it might be interesting to share a peek inside one such company—how we use ChatOps to communicate, troubleshoot, and monitor our own infrastructure.

My story begins a few days ago when our production API experienced a small glitch. Not the kind of thing that provokes a thorough postmortem, but just a momentary network issue of the sort that briefly degrades service.

Glitches like this are to be expected, but when you're running distributed applications, and especially multi-tenant SaaS, these little hiccups are sometimes the harbingers of disaster—they cannot be allowed to persist and should be detected and investigated as quickly as possible. Our glitch on this particular morning didn't grow into anything disastrous, but I thought it might be nice to share it with you, to give you a peek at what problem detection and diagnosis looks like at Librato.

A Culture of ChatOps

About half of our engineers are remote, so unsurprisingly we rely heavily on ChatOps for everything from diabolical plotting to sportsball banter. Because it's already where we tend to "be" as a company, we've put some work into integrating our persistent chat tool with many of the other tools we use. It should be no surprise, then, that our first hint something was wrong came by way of chat (Figure 1).

Dr. Manhattan (Dr. M), a special-purpose account used for tools integration, is the means by which our various third-party service providers feed us notifications and—like his namesake—can talk to all of our service providers at the same time. In this paste, he's letting us know that he's gotten two notifications from our own alerting feature. The first alert means that our API is taking longer than normal to look up metric names stored in memcached, while the second indicates that our metrics API is responding slowly in general to HTTP POSTs.

Thanks to our campfire [5] integration, Dr. M. is also able to tell us the names of the hosts that are breaking the threshold and their current values. This is alarming enough, but these alerts are quickly followed by more. First comes a notification from our log processing system (Figure 2).

We've configured rsyslog on our AWS hosts to send a subset of our ngnix logs to a third-party alert processing service (Papertrail). About the same time those metrics crossed threshold, Papertrail noticed some HTTP 502 errors in our logs and is sending them to Dr. M. who is listing them in channel. Some of these lines indicate that a small number of requests are failing to post. Not good.

More trouble follows, including several more alerts (Figure 3) relating to our API response time, as well as notifications from our alert escalation service [6], and our exception reporting service [7], the latter of which indicates that some of our users' sessions might be failing out with I/O errors.

Alert triggered at 2014-01-17 17:13:11 UTC for 'rack.metrics-api-prod.response_time.post' with value 12969.483398 from metrics-web-prod-513: https://metrics.librato.com/metrics/rack.metric...

Alert triggered at 2014-01-17 17:13:11 UTC for 'api.cache.metrics.mget.time' with value 789.981506 from metrics-web-prod-513: https://metrics.librato.com/metrics/api.cache.m...

⚡ [metrics-api] IOError - closed stream https://www.honeybadger.io/inspector/p/1494/330...

🔥 2014-01-17T17:13:13Z Amnis Production triggered http://librato.pagerduty.com/incidents/PWLF5R3/... 🔥

Figure 3: A third set of alerts from various service providers appear in channel.

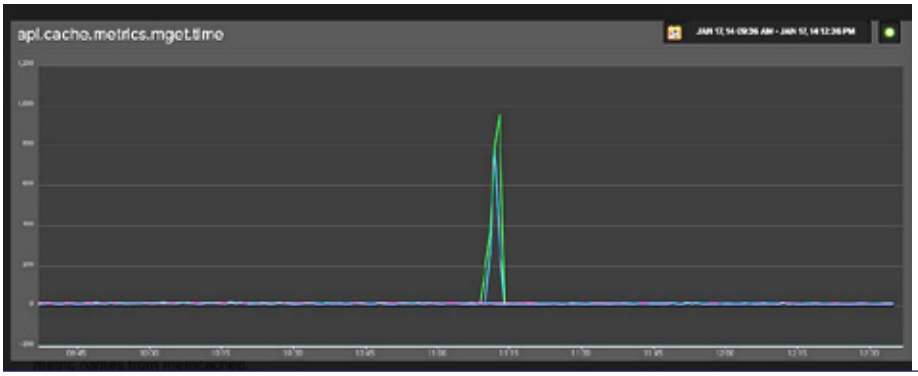

Figure 4: Our API latency is visualized from links in the chat-alerts.

Jared K. wow

Figure 5: Our engineers react to the influx of bad news.

Collin V.	so we saw a backup of the amnis 'measures' workload
	at the same time we saw a huge spike in api latency
Peter H.	Yeah.... looking to see if there is an obvious cause.
Collin V.	kafka a common possible theme in bolth cases
	9:30 AM
Jared K.	api->kafka latency doesn't appear to have changed during that time
Peter H.	Not feeling the kafka idea so far. Kafka is rarely the issue. I'm looking at ELBs, and/or maybe az level issues?
	A couple of metrics-web nodes saw their usage plummet for a minute there.
Jared K.	yup
	a cpu usage drop on two nodes afaict
Peter H.	May just be some kind of strange artifact 'cause web console view of same time period doesn't show that.
Collin V.	saw that as well
Peter H.	Oh yeah, the ELB definitely thought something was wrong for a second.

Figure 6: Our engineers troubleshoot the latency issue.

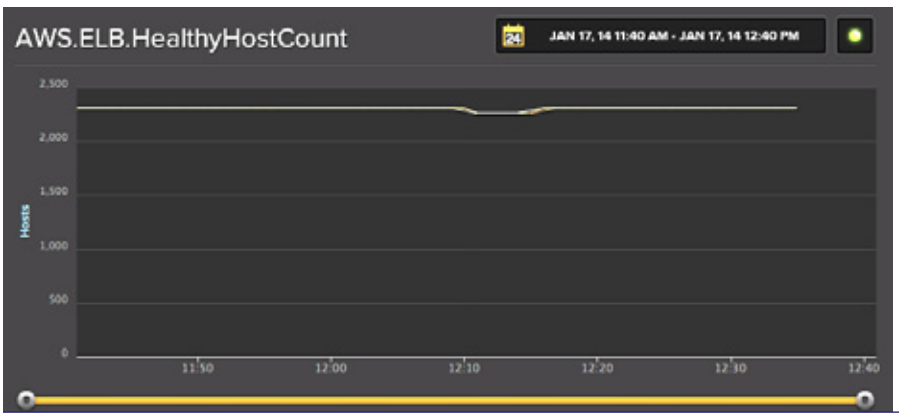

Figure 7: The data confirms a short-lived upstream network partition.

Peter H.

Figure 8: Thumbs up from our operations guys signals the all clear.

Dr. M. 2014-01-12T07:27:12Z Prod Metrics Alerts Email triggered http://librato.pagerduty.com/incidents/PP6QSEH/...

2014-01-12T07:27:47Z Prod Metrics Alerts Email acknowledged http://librato.pagerduty.com/incidents/PP6QSEH/...

11:30 PM

Peter H. has entered the room

Peter H. Hm.

Not sure what is going on but we started to see some 500s about 45 minutes ago, then had a big spike which is what triggered this alarm.

11:35 PM

Peter H. No spike in RDS, no obvious changes in metrics-web EC2 metrics, No big changes in raw-v3 or rollups-v1.

11:40 PM

Peter H. Hm, this seems not good:

View paste

```
Replication State: error
```

Replication Error: Apply Error 1594: Relay log read failure: Could not parse relay log event entry. The possible reasons are: the master's binary log is cor
check this by running 'mysqlbinlog' on the binary log), the slave's relay log is corrupted (you can check this by run...

That's on portal-production-rr16, only visible after loading up the RDS part of AWS web console and looking at status of RDS nodes.

11:45 PM

Peter H. Ah ok, you can see that something went awry on the Binary Disk Log Usage graph:

Dr. M. RDS Binary Log Disk Usage: https://metrics.librato.com/instruments/3142855...

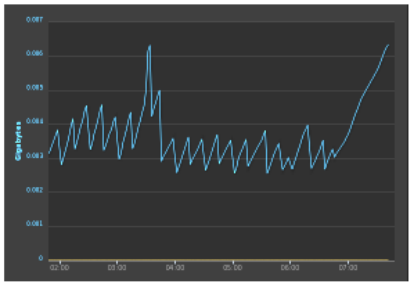

Figure 9: Our operations staff, talking to themselves.

In every case, the notifying entity provides us a link that we can use to get more info. For example, clicking the link in the first notification from our alerting feature yielded the graph (Figure 4) in our metrics UI.

Sure enough, there are two obvious outliers here, indicating that two machines (out of the dozen or so API hosts) were three orders of magnitude slower than their peers returning metric names from memcached.

Our engineers take a moment to assimilate the barrage of bad news that was just dumped into the channel (Figure 5), and then they dig in to figure out just what exactly is going on. Having machine notifications inline with human conversation is a huge win for us. The ability to react directly to what we are seeing in channel without having to context-switch between our phones and workstations makes us a more productive team—everyone is literally on the same page all the time (Figure 6). We win again when the troubleshooting we do as a team is automatically documented, and any newcomers to the channel who join mid-crisis get all the context in the scrollback when they join.

We initially suspected that our message queue might be the culprit, but we were able to quickly check the queue latency graph and eliminate that possibility without wasting time poking at the queue directly. Then we noticed some aberrant system-level stats on the two hosts that broke threshold in the initial alert.

Our metrics tool puts all of our production metrics in one place, so it's trivial to correlate metrics from one end of the stack to the other. Using a combination of application-layer and systems-level graphs, we were able to verify that the problem was in fact

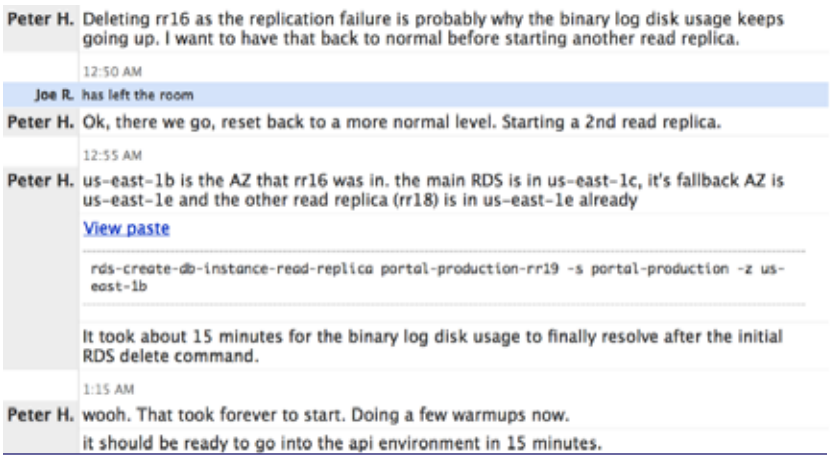

Figure 10: Our on-call engineer deletes a malfunctioning database replica.

Figure 11: Our on-call engineer brings up a new replica and monitors the result.

some sort of short-lived network partition that isolated those two particular AWS nodes. The data included the graph in Figure 7, which depicts our total number of healthy AWS nodes (this data sources from AWS CloudWatch [8], using our turnkey AWS integration feature).

The "dip" you see indicates that two of our nodes went dark for a few seconds (too short a length of time for them to have bounced). We noted their names and will keep them under observation for a few days, but at that point all we could do was glare in Amazon's general direction and call "all clear" (Figure 8).

Communicate to Document

We love ChatOps so much that sometimes—late at night, when nobody is around—you can catch our engineers talking to themselves in channel (Figure 9).

This happened not long ago, late at night on a Sunday. Our on-call Ops engineer was alerted by pager via our escalation service about a database problem. You can see both the escalation alert and our engineer's acknowledgment shortly before he joins the channel at the top. As he troubleshoots the issue, he narrates his discoveries and pastes interesting tidbits, including log snippets and graphs of interesting metrics. In this way, he documents the entire incident from detection to resolution for the other engineers who will see it when they join the channel the next morning. Using ChatOps to document incidents has become an invaluable practice for us. Important customer interactions, feature ideas, code deploys, and sales stats are also communicated asynchronously, company-wide, via ChatOps. It is our portal, wiki, and water cooler.

Another practice that we would find it hard to live without is that of sharing graphs back and forth in channel in the way Peter has done above. This is a handy way to both communicate what you're seeing to other engineers and simultaneously document it for later. We'd begun to rely so heavily on copy/pasting graphs to each other that we added a snapshot [9] feature to our metrics tool so our engineers can share the graph they're looking at in our UI directly to a named chatroom in our chat tool without copy/paste.

In this particular incident, Peter tracks the issue down to a read-replica failure on a particular database node, and decides to replace the node with a fresh instance. He first deletes the faulty host and waits for things to normalize (Figure 10), and then he brings up the new node (Figure 11) and monitors dashboards until he's convinced everything is copacetic.

Throughout his monologue, Peter is using the snapshots feature I mentioned earlier to share graphs by clicking on the graph in our UI, and then hitting the snapshot button to send it to Dr. M., who, in turn, pastes it into the channel for everyone to see. Snapshots are generally preferable to copy/pasting because they provide everyone in channel a static PNG of the graph as well as a link back to the live graph in our UI. Those of us who use the Propane [10] client for Campfire even get live graphs [11] in channel instead of boring PNGs.

ChatOps Works

ChatOps delivers on the promise of remote presence in a way the presence protocols never did. It's a natural estuary for stuff that's going on right now; information just can't help but find its way there, and having arrived, it is captured for posterity. ChatOps is asynchronous but timely, brain-dead simple yet infinitely flexible. It automatically documents our internal operations in a way that is transparent and repeatable, and somehow manages to make time and space irrelevant.

Furthermore, because our monitoring is woven into every layer of the stack and is heavily metrics-driven, data is always available to inform our decisions. We spend less time troubleshooting than we would if we chose to rely on more siloed, legacy techniques. Instrumentation helps us to quickly validate or disprove our hunches, focusing our attention always in the direction of root cause.

References

[1] An Ode to Distributed Teams: http://blog.idonethis.com/post/41946033190/an-ode-to-distributed-teams.

[2] ChatOps at GitHub (Rubyufza '13): http://www.youtube.com/watch?v=NST3u-GjjFw.

[3] Distributed teams at buffer: http://joel.is/post/59525266381/the-joys-and-benefits-of-working-as-a-distributed-team.

[4] Distributed teams at Zapier: https://zapier.com/blog/how-manage-remote-team/.

[5] Campfire team collaboration tool: https://campfirenow.com/.

[6] Pagerduty: http://www.pagerduty.com/.

[7] Honeybadger: https://www.honeybadger.io/.

[8] AWS CloudWatch: http://aws.amazon.com/cloudwatch/.

[9] Librato Snapshots: https://metrics.librato.com/product/features.

[10] Propane client for Campfire: http://propaneapp.com/.

[11] Librato-Propane library: https://github.com/librato/librato-propane.