Vladimir Kolobaev Follow
Oct 2 · 11 min read

Photo by Matthew Henry on Unsplash

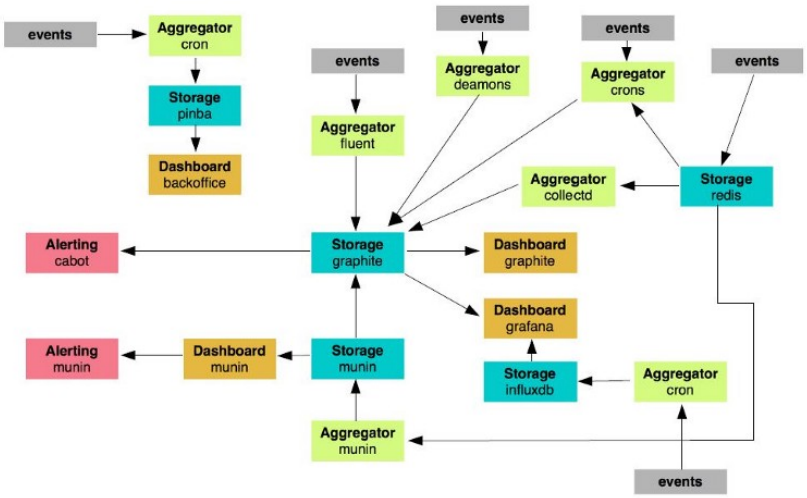
Monitoring as a Service: a Modular System for Microservice Architecture

In addition to the all-in-one code, our project is supported by dozens of microservices. Each of them needs to be monitored. Having all them monitored by DevOps engineers is hardly possible. We have developed a monitoring system operating as a service for developers. They can on their own configure metrics in the monitoring system, use them, build metrics-based dashboards, set up alerts triggered by thresholds. The only thing that DevOps engineers have to provide is the infrastructure and documentation.

This blog post is a transcript of my presentation at our RIT++ section. We have received multiple inquiries about a text version of the presentations made at RIT++. If you attended the conference or watched the video, you will hardly discover anything new. Otherwise, enjoy this blog post. I'll tell you how we arrived at this solution, how it works, and how we plan to update it.

The past: layouts and plans

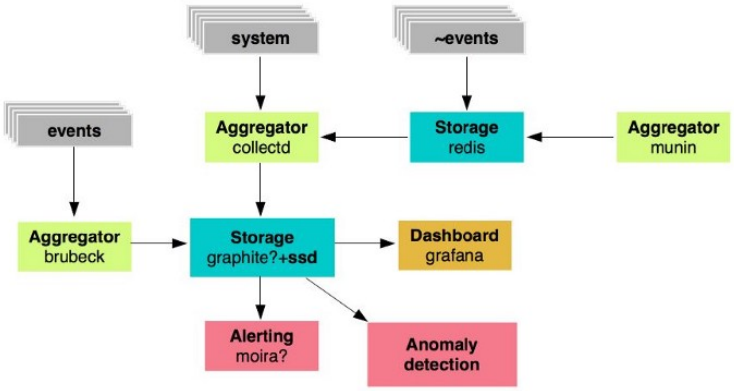
How did we arrive at the existing monitoring system? In order to answer this question, we need to go back to 2015. Here's how it looked back then:



We had 24 nodes responsible for monitoring. There is a whole bunch of crons, scripts, and daemons of all types that somehow monitor

something, send messages, perform other functions. We realized that the further down that road we went, the more unsustainable the system would grow. It did not make sense developing the system—it was simply too cumbersome.

We decided to select those monitoring elements that we will keep and develop, and those that will be dropped. 19 elements were selected to be kept. Those included only Graphites, aggregators, and Grafana as the dashboard. But what will the new system look like? Like this:



We have a metrics repository—Graphites on fast SSD disks and metrics aggregators. Also, Grafana for displaying the dashboards and Moira for the alerting function. We also wanted to develop a system for finding anomalies.

Standard: Monitoring 2.0

Here's what our plans looked like in 2015. But we had to develop not only the infrastructure and the service itself, but also its documentation. We developed a corporate standard and called it Monitoring 2.0. System requirements were like this:

- 24/7 availability,
- metrics storage interval = 10 seconds,
- structured storage of metrics and dashboards,

• SLA > 99.99%,

• collection of event metrics via UDP (!).

We needed UDP, because we have heavy traffic and multiple events generated by metrics. If they are all immediately stored in Graphite, the repository will crash. We also chose first level prefixes for all metrics.

Group	Stored metrics
network	network and network hardware
servers	server performance metrics only (cpu, ram, hdd, swap, eth, etc.)
containers	metrics of containers and hosts capable of changing their physical location
resources	metrics of shared resources, e.g. memcache, rabbitmq cluster, etc.
apps	metrics of services, standalone apps, daemons, cron-scripts, apps inside tarantula, etc.
products	product metrics
complex	complex metrics

Each of the prefixes has some property. We have metrics for servers, networks, containers, resources, apps, and so on. There is a clear, strict, typified filtering approach, where we keep the first level metrics and drop the others. That's how we saw the system in 2015. What does it look like today?

Today: interaction between the monitoring components

First of all, we monitor apps—our PHP code, apps, and microservices—in short, everything our developers code. All apps send metrics via UDP to Brubeck aggregator (statsd, rewritten in C). It proved to be the fastest in synthetic tests. Brubecks sends the aggregated metrics to Graphite via TCP.

It has a special type of metrics—timers. They are extremely convenient. For example, for every user connection to a service, you send the response time metric to Brubeck. Even with a million responses, the aggregator generates as few as 10 metrics. You have the number of visitors, the maximum, minimum, and average response time, the median value, and the 4 percentiles. Then the data is transferred to Graphite and we see it all live.

We also aggregate the hardware and software metrics, system metrics, and our legacy monitoring system Munin (we used it until 2015). We collect all this with the C daemon CollectD (it has a whole bundle of plugins embedded, can interrogate any resources of the host system it is installed onto, and you only need to specify in the configuration where the data should be written) and send the data to Graphite. It also supports python plugins and shell scripts, so you can develop custom solutions: CollectD will collect the data from a local or remote host (let's assume that there is a Curl) and send it to Graphite.

Then, all the collected metrics will be sent to Carbon-c-relay. It is the Carbon Relay solution by Graphite, modified in C. It is a router that collects all the metrics that we send from our aggregators and routes them to the nodes. When routing, it checks the validity of the metrics. First, they must match the prefix layout shown above and, second, they must be valid for Graphite. Otherwise, they are dropped.

Then Carbon-c-relay sends the metrics to the Graphite cluster. As the primary metrics repository, we use Carbon-cache modified in Go. Because of its multithreading capability, Go-carbon is much more powerful than Carbon-cache. It receives the data and writes it to disks using the whisper package (standard package, written in python). To read the data from our repositories, we use the Graphite API. It is much faster than the standard Graphite WEB. What happens to the data next?

The data is sent to Grafana. As the main data source, we use our Graphite clusters, and we have Grafana as a web interface for displaying metrics and building dashboards. For each of their services, developers build its own dashboard. Then they plot graphs showing metrics taken from their apps. In addition to Grafana, we also have SLAM. This is a python daemon for calculating SLA based on the data from Graphite. As I said, we have several dozen microservices, each of which has its specific requirements. Using SLAM, we check the documentation, compare it with Graphite's data, and assess whether the availability level of our services meets the specifications.

Alerting is the next step. It is built using a powerful system—Moira. It is autonomous because it has its own Graphite under the hood. It was developed by the SKB Kontur team, written in Python and Go, and is 100% open source. Moira receives the same stream that goes into

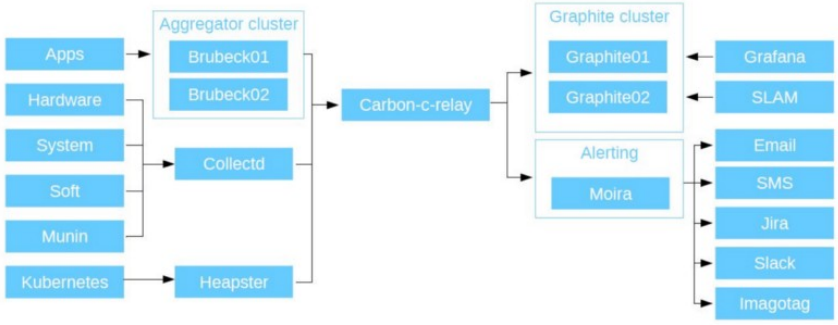
Graphites. If, for some reason, the repository is down, the alerting function will still work.

We deployed Moira in Kubernetes and as a primary database it uses a cluster of Redis servers. As a result, we have a fault-tolerant system. It compares the metrics stream with the list of triggers: if there are no mentions in it, it drops the metric. So it is capable of processing gigabytes of metrics per minute.

We also added a corporate LDAP, with the help of which any user of the corporate system can set up notifications for existing (or new) triggers. Since Moira contains Graphite, it supports all its functions. So, first you choose a line and copy it into Grafana. Check how the data is displayed in the graphs. And then you copy the same line to Moira. Set up limits and now you have an alert. To do all this, you do not need any special skills. Moira can send alerts by SMS, email, to Jira, Slack, etc. It also supports the execution of custom scripts. When it is triggered and is subscribed to a custom script or binary, it launched the binary and sends JSON to the binary's stdin. And your program has to parse it. It is up to you what to do with the JSON. Send it to Telegram, open Tasks in Jira, or do whatever you want.

For the alerting function, we also use our proprietary solution—Imagotag. We adapted to our needs the panel usually used for electronic price tags in stores. We use it to display Moira triggers. It indicates their status and time. Some of our developers have unsubscribed from notification by Slack and email in favor of this dashboard.

And since we are a future-oriented business, we also use this system to monitor Kubernetes. We added it to the system using Heapster, which we installed in the cluster to collect the data and send it to Graphite. The resulting layout looks like this:



Monitoring components

Here is a list of links to the components that we used to do this. They are all open source.

Graphite:

- go-carbon: github.com/lomik/go-carbon

- whisper: github.com/graphite-project/whisper
- graphite-api: github.com/brutasse/graphite-api

Carbon-c-relay:

github.com/grobian/carbon-c-relay

Brubeck:

github.com/github/brubeck

Collectd:

collectd.org

Moira:

github.com/moira-alert

Grafana:

grafana.com

Heapster:

github.com/kubernetes/heapster

Statistics

Here are some performance statistics of our system.

Aggregator (brubeck)

Number of metrics: ~ 300,000/sec

Interval for sending metrics to Graphite: 30 sec

Server resource usage: ~ 6% CPU (here we mean fully-featured servers); ~ 1 Gb RAM; ~ 3 Mbps LAN

Graphite (go-carbon)

Number of metrics: ~ 1,600,000/min

Metrics refresh interval: 30 sec

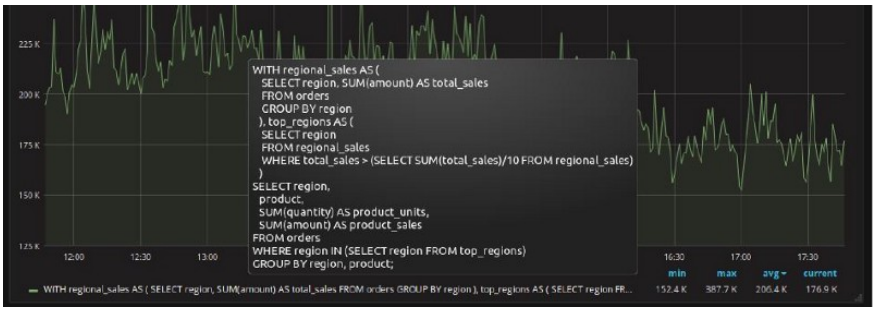
Metrics storage pattern: 30sec 35d, 5min 90d, 10min 365d (gives an idea of how the service performs over an extended period of time)

Server resource usage: ~ 10% CPU; ~ 20 Gb RAM; ~ 30 Mbps LAN

Flexibility

We at Avito greatly appreciate the flexibility of our monitoring service. Why is it actually so flexible? First, its components are interchangeable —both the components themselves and their versions. Second, it is highly supportable. Since the entire project is built on open source solutions, you can yourself edit the code, make changes, implement functions not available off the shelf. We use quite common stacks, mostly Go and Python, so it's quite easy to implement.

Here is an example of a real-life problem. A metric in Graphite is a file. It has a name. Filename = metric name. And it has a path. In Linux, filenames are limited to 255 characters. Here come some guys (our internal customers) from the database team. They say: "We want to monitor our SQL queries. And they are not 255 characters, but 8 MB each. We want them displayed in Grafana, see the query's parameters, or even better, see the top rating of the queries. Would be great if it is displayed in real time. And ideally, they should be integrated into the alerting function."



An example of a SQL query comes from postgrespro.ru

We set up a Redis server and, using our Collectd-plugins that connect to Postgres and get the data from there, send the metrics to Graphite. But we replace the name of the metric with hashes. The same hash is sent

to Redis as the key and the entire SQL query as the value. The only thing left is to enable Grafana to connect to Redis and get the data. We open the Graphite API, since it is the main interface for the interaction between all the monitoring components and Graphite, and enter a new function called aliasByHash ()—from Grafana, we get the name of the metric and enter it in the Redis query as the key, in return we get the value of the key, which is our "SQL query." Thus, we have displayed in Grafana a SQL query, which in theory cannot be displayed there, along with its statistics (calls, rows, total_time, …)

Conclusions

Availability. Our monitoring service is available 24/7 from any app and any code. If you have access to the repositories, you can write data to the service. Language doesn't matter, solutions don't matter. You only need to know how to open a socket, upload a metric, and then close the socket.

Reliability. All components are fault-tolerant and perform well under our loads.

Low entry barriers. In order to use this system, you do not need to know programming languages and queries in Grafana. You simply open your app, set up a socket that will send metrics to Graphite, close it, open Grafana, create dashboards, and monitor your metrics through notifications via Moira.

Independence. All this can be done independently, without the involvement of DevOps engineers. And this is a clear advantage, because you can start monitoring your project right away, without having to ask anyone for help—either to get started or to make changes.

What are we striving for?

All listed below is not just abstract thoughts, but actual goals, towards which first steps have been made.

1. Anomaly detector. We want to set up a service that will connect to our Graphite repositories and check every metric by different

algorithms. We have algorithms that we want to view, we have the data, we know how to handle the data.

2. Metadata. We have many services, and they change over time, and so do those who support and use them. Maintaining the documentation manually is not an option. Therefore metadata are now being built into our microservices. Metadata specify who developed the service, the languages it supports, SLA requirements, recipients of notifications and their addresses. When the service is deployed, all data entities are created independently. As a result, you have two links—one to the triggers, the other one to the dashboards in Grafana.

3. Monitoring for everyone. We believe that every developer should use this system. In that case, you can always understand where your traffic is, what happens to it, where problems and bottlenecks are. If, for example, something has crashed your service, you will discover that not when your customer service agent calls you, but from an alert and will be able to immediately open the logs and check what happened.

4. High performance. Our project is constantly growing, and today it processes nearly 2,000,000 metrics values per minute. A year ago, the figure was 500,000. Meanwhile, we are still growing, and it means that, after a while, Graphite (whisper) will start overloading the disk subsystem. As I said, the monitoring system is quite universal due to the interchangeability of its components. Some have chosen to support and expand their infrastructure specifically for Graphite, but we decided to go the other way—to use ClickHouse as a repository for our metrics. The transition is almost complete, and very soon I'll describe in more detail how that was done—what the challenges were and how we overcame them, how the migration process went; I will describe the harness components and their configurations.

Hope you enjoyed reading this. Your questions are welcome and I'll do my best to answer them here or in the upcoming posts. If anyone has experience building a similar monitoring system or switching to Clickhouse in a similar situation—share it in the comments.