A scalable architecture for automated monitoring of microservices

Radu BONCEA[1,2], Alin ZAMFIROIU[1,3], Ioan BACIVAROV[2]
[1]National Institute for Research and Development in Informatics, ICI Bucharest,
[2]University Politehnica of Bucharest, Faculty of Electronics Telecommunications and Information Technology
[3]Bucharest University of Economic Studies Bucharest, Romania
radu.boncea@ici.ro, zamfiroiu@ici.ro, bacivaro@euroqual.pub.ro

In this article we propose an architecture for monitoring microservices by logging key performance metrics at both system and application levels. By using advanced analytics algorithms, we can then classify the events in microservice's behavior and automate the decision processes, thus improving the overall reliability and security. We will be using Prometheus for storing short-live metrics, OpenTSDB for long term retention and RabbitMQ for passing structured messages between various IT components which orchestrate the collection of our microservices.

Keywords: microservice, distributed architecture, monitoring, complexity
JEL classification: C38, C8, O21

1 Introduction

Microservice architecture is a method of developing software applications as a suite of independently deployable, modular services where each such microservice represents an unique business process capable of communicating through a well-defined, lightweight mechanism to serve a business goal. It has its roots in service-oriented architecture and, initially, it was indistinguishable from SOA. In fact, Netflix used the term “fined grained SOA” [1].

Microservice oriented architecture greatly improve [2]:

- deployability: shorter build-test-deploy cycles, increased agility in rolling new versions and applying patches, greater flexibility in employing security, replication, persistence;
- reliability through better fault isolation;
- availability: rolling out new versions or applying patches require little downtime as only specific microservices are restarted;
- scalability: each microservice and be deployed in containers, thus greatly benefiting from the elasticity of the cloud;
- modifiability: more flexibility to use new frameworks, libraries, data sources, and other resources;
- management: the development effort is divided across teams that are smaller and work more independently.

Microservice architecture also has drawbacks when compared to monolith architecture[3]:

- it adds complexity to the project just by the fact that a microservices application is a distributed system. You need to choose and implement an inter-process communication mechanism based on either messaging or RPC and write code to handle partial failure and take into account other fallacies of distributed computing;
- it has the partitioned database architecture. Business transactions that update multiple business entities in a microservices-based application need to update multiple databases owned by different services. Using distributed transactions is usually not an option and you end up having to use an eventual consistency based approach, which is more challenging for developers;
- it makes testing much more complex then in case of monolithic web application. For a similar test for a service you would need to launch that service and any services that it depends upon (or at least configure stubs for those services);
- it is more difficult to implement changes

that span multiple services. In a monolithic application you could simply change the corresponding modules, integrate the changes, and deploy them in one go. In a Microservice architecture you need to carefully plan and coordinate the rollout of changes to each of the services;
- deploying a microservices-based application is also more complex. A monolithic application is simply deployed on a set of identical servers behind a load balancer. In contrast, a microservice application typically consists of a large number of services. Each service will have multiple runtime instances. And each instance need to be configured, deployed, scaled, and monitored. In addition, you will also need to implement a service discovery mechanism. Manual approaches to operations cannot scale to this level of complexity and successful deployment a microservices application requires a high level of automation.

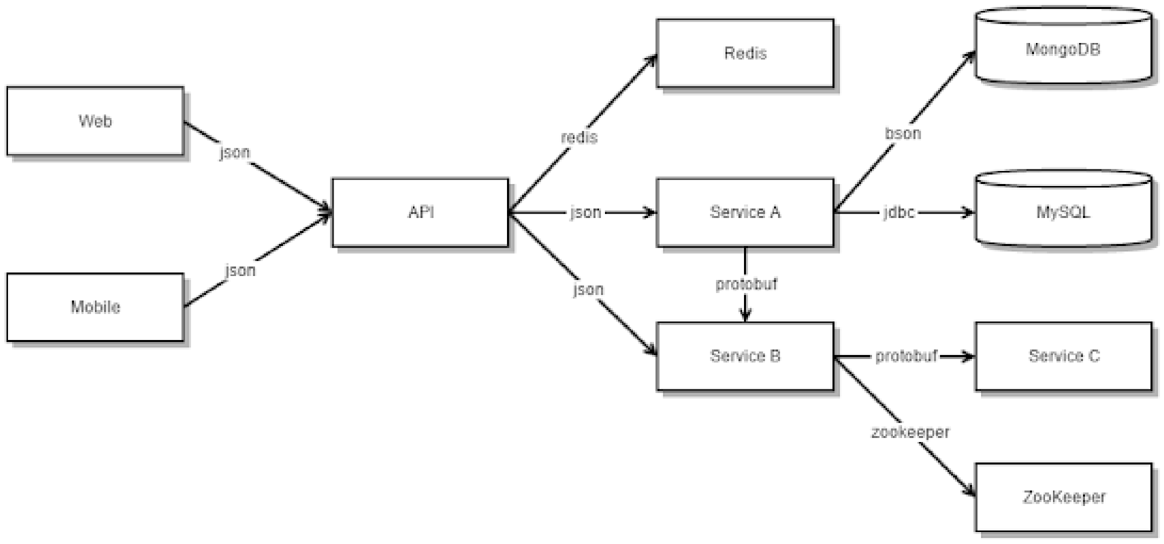

Fig. 1 Typical representation of a solution based on microservices.
Source: http://www.antonkharenko.com

In a standard monolithic architecture, the application is monitored using black box techniques by pushing key infrastructure and performance metrics, such as server CPU, memory, disk IO and network utilization to solutions such as Nagios. White box monitoring is done passively using SNMP traps, a process that requires developers to either alter the application logging subsystem or to add an external asynchronous mechanism to process existing applications logs and send via SNMP the required metrics to Nagios. Nagios also supports active monitoring, where Nagios would do a periodic check on the application, but, usually, this type of monitoring is used to check on specific states of the application and it is not intended to poll metrics.

An example of how Nagios is used is shown in Figure 2 where we have a monolithic application that resides on the same server as the rest of support applications, such as the database. This deployment pattern is common in most enterprises and the increase in complexity, which usually means more instances of the same application server, is handled efficiently directly in Nagios and, occasionally, by doing iterative adjustments to SNMP trap system.

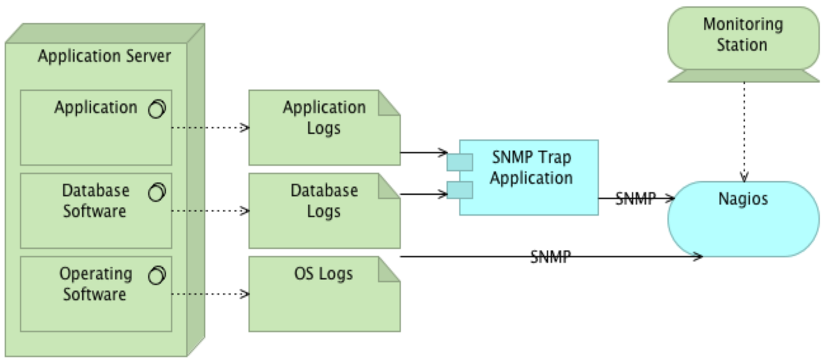

Fig. 2. Monitoring a single application using Nagios

Problems occur when the application is decomposed into services, each service having its own deployment pattern and requiring additional support applications such as Message Exchange Brokers. In this scenario, Nagios becomes unproductive and time-series databases that support data labeling, query languages and interfaces for easy-integration are considered better choices.

2. Monitoring Tools

Monitoring platforms focus largely on the gathering and analysis of the data that is collected from applications and operating system and network platforms on which they run.

Prometheus is an open source monitoring solution originally developed by SoundCloud. It is widely used to store and query "time-series data," which is data that describes actions over time. Prometheus is often combined with other tools, especially Grafana, to visualize the time series data and to provide dashboards.
It uses LevelDB as storage engine and features[4]:

- a multi-dimensional data model with support for labels;
- a functional expression language that lets the user select and aggregate time-series data in real time and which can be used by external systems via HTTP API calls;
- a pull model via HTTP for collecting time-series;
- a push model model via intermediary gateway for batch collections;
- autodiscovery of target sources.

There are two methods of integrating Prometheus jobs/exporters into a microservice architecture:

- implement standalone exporters, independent of the microservice, that consume microservices logs and expose metrics via HTTP;
- integrate the exporter mechanics into microservice using already existing software libraries written in Python, Go, Java, Scala, Ruby, C++, Erlang, Rust, Node.js, .NET/C#, PHP.

There are also a number of libraries and servers which help with exporting existing metrics from third-party systems as Prometheus metrics;

- databases: PostgreSQL, MySQL, MongoDB, Oracle, Memcached, CouchDB, MSSQL, Redis, ElasticSearch;
- hardware: Node/System exporter for Linux/Unix, IoT Edison, apcupsd, IPMI, Ubiquiti UniFi;
- messaging systems: RabbitMQ, MQTT, Kafka, Beanstalkd;
- storage: Hadoop HDFS FSImag, Lustre, ScaleIO, Gluster;
- HTTP: Apache, HAProxy, Nginx, Passenger, Tinyproxy, Varnish;
- APIs: AWS ECS/Health/SQS, Cloudflare, DigitalOcean, Docker Cloud/Hub, OpenWeatherMap;

A very useful capability Prometheus has is that of „scraping" data from systems that are already exposing metrics in Prometheus format such as Collectd, Kubernetes, NetData, Grafana.

OpenTSDB is built on top of HBase and Hadoop and is focused on being a distributed and passive time-series database with a query language and graphing features. Unlike Prometheus, OpenTSDB is not aware of the surroundings, it has no knowledge of the endpoints and no mechanics for finding faults and alerting. This knowledge has to be implemented independently, outside OpenTSDb.
Because OpenTSDB is built on Hbase and could scale horizontally, it is ideal for long term retention of data. Prometheus does not scale natively, instead it requires explicit sharding which raises additional issues when considering a full automation of the monitoring system.

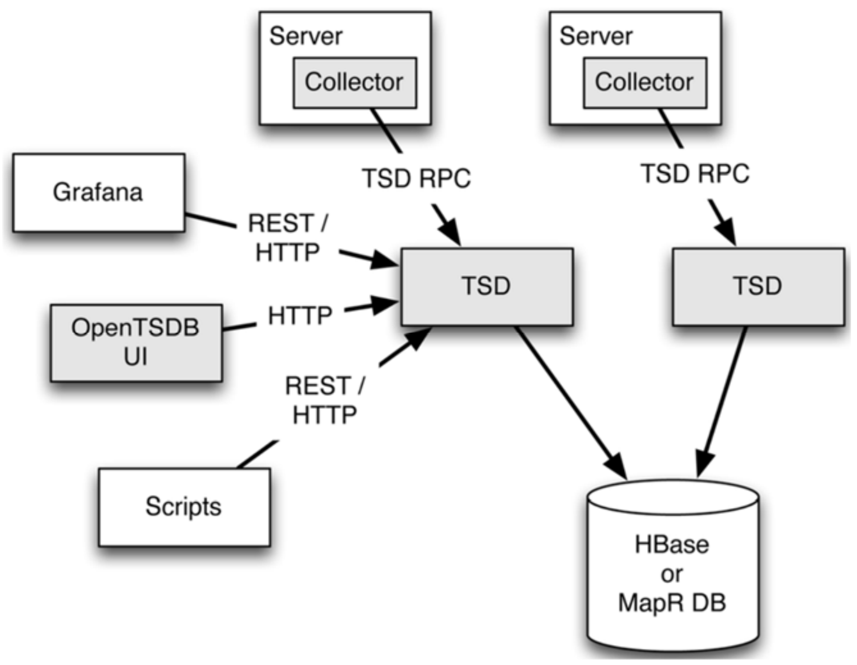

Fig. 3. OpenTSDB architecture

Elastic APM is an application performance monitoring system built on the Elastic Stack. It allows you to monitor software services and applications in real time, collecting detailed performance information on response time for incoming requests, database queries, calls to caches, external HTTP requests, etc. This makes it easier to pinpoint and fix performance problems quickly.
Elastic APM also collects automatically unhandled errors and exceptions. Errors are grouped based primarily on the stacktrace, so you can identify new errors as they appear and keep an eye on how many times specific errors happen.
Elastic APM consists of four components:

- Elasticsearch: a search engine based on the Lucene library. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents;
- APM agents: persistent applications that collect performance metrics and send it to Elasticsearch via a middleware application called APM Server;
- APM Server process data sent from APM agents and stores it to Elasticsearch;
- Kibana UI: an open source data visualization plugin for Elasticsearch. It provides visualization capabilities on top of the content indexed on an Elasticsearch cluster

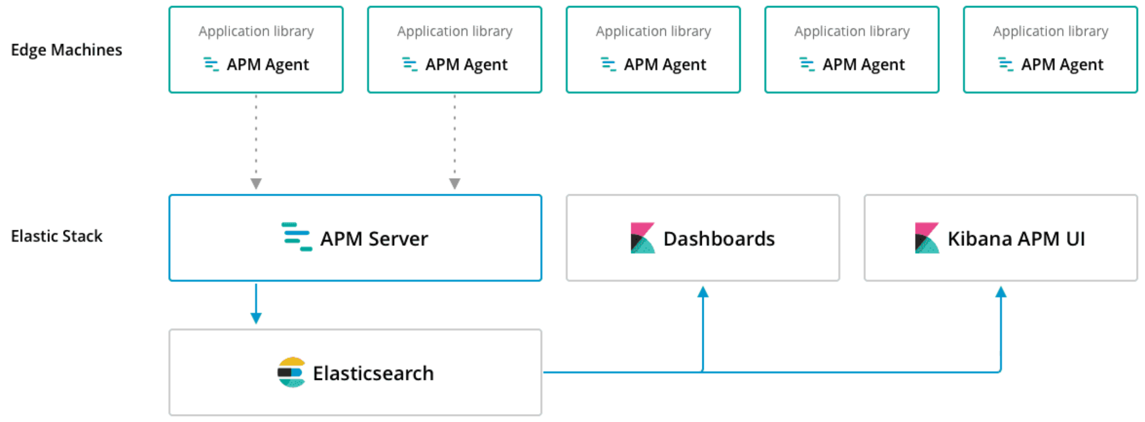

Fig. 4. Elastic APM architecture
Source: https://www.elastic.co

The APM Server is a separate component for the following reasons:

- It helps to keep the agents as light as possible and since the APM Server is a stateless separate component, it can be scaled independently;
- For Real User monitoring data is collected in browsers. APM Server prevents browsers from interacting directly with Elasticsearch (which poses a security risk) and controls the amount of data flowing into Elasticsearch;
- In cases where Elasticsearch becomes unresponsive, APM Server can buffer data temporarily without adding overhead to the agents;
- Acts as a middleware for source mapping for javascript in the browser.
- Provides a JSON API for agents to use thereby improving compatibility across different versions of agents and the Elastic Stack.

TICK stack is an open source time-series platform designed from the ground up to handle metrics and events an is built on top of:

- **Telegraf** is a plugin-driven server agent for collecting and reporting metrics. Telegraf has plugins or integrations to source a variety of metrics directly from the system it's running on, to pull metrics from third party APIs, or even to listen for metrics via a StatsD and Kafka consumer services. It also has output plugins to send metrics to a variety of other datastores, services, and message queues, including InfluxDB, Graphite, OpenTSDB, Datadog, Librato, Kafka, MQTT, NSQ, and many others.
- **InfluxDB** is a Time Series Database built from the ground up to handle high write & query loads. InfluxDB is a custom high performance datastore written specifically for timestamped data, including DevOps monitoring, application metrics, IoT sensor data, and real-time analytics. Conserve space on your machine by configuring InfluxDB to keep data for a defined length of time, and automatically expiring and deleting any unwanted data from the system. InfluxDB also offers a SQL-like query language for interacting with data.
- **Chronograf** is the administrative user interface and visualization engine of the platform. It makes the monitoring and alerting for your infrastructure easy to setup and maintain. It is simple to use and includes templates and libraries to allow you to rapidly build dashboards with real-time visualizations of your data and to easily create alerting and automation rules.
- **Kapacitor** is a native data processing engine. It can process both stream and batch data from InfluxDB. Kapacitor lets you plug in your own custom logic or

user-defined functions to process alerts with dynamic thresholds, match metrics for patterns, compute statistical anomalies, and perform specific actions based on these alerts like dynamic load rebalancing. Kapacitor integrates with HipChat, OpsGenie, Alerta, Sensu, PagerDuty, Slack, and more.

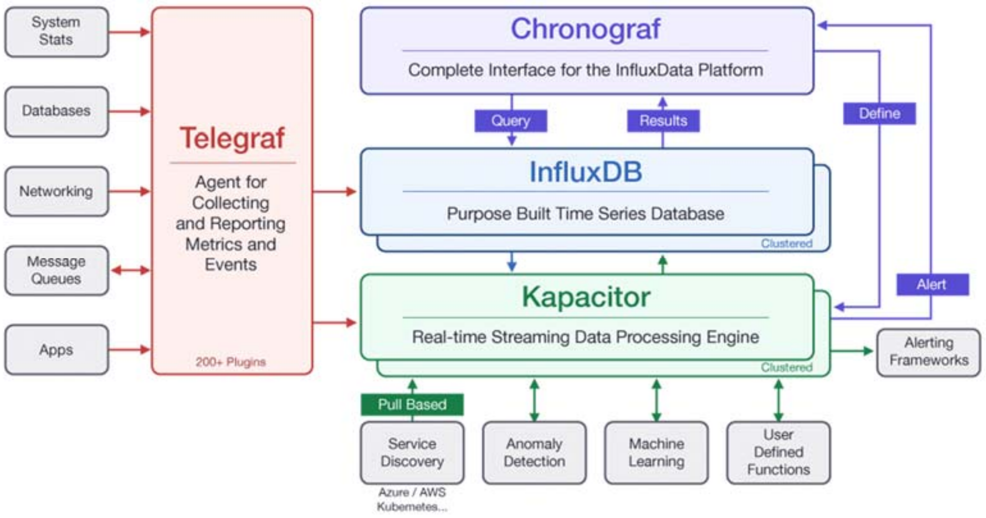

Fig. 5. TICK platform overview
Source: https://www.influxdata.com

Raygun's APM platform is a commercial complete system that provides both instrumentation and collector processes, as well as a dashboard to visualize metrics data. It can integrate GitHub, Slack, Jira Software, PagerDuty, VictorOps and it features detailed transaction tracing, dashboards, user experience reporting and real user monitoring.
Zipkin is an open-source tracing system designed specifically to trace calls between microservices. It is especially useful for analyzing latency problems. Zipkin includes both instrumentation libraries and the collector processes that gather and store tracing data.
Kafka is a streams-processing system. It uses a "publish/subscribe" methodology for reading and writing data to a logical "stream," which is similar in concept to a messaging system such as RabbitMQ. Kafka can be combined with other tools such as Zipkin to provide a robust solution for transmitting and storing metrics data.

Table 1. Comparison of capabilities between Prometheus, OpenTSDB and Elastic APM

	Prometheus	**OpenTSDB**	**Elastic APM**
Storage	LevelDB	Hadoop/HBase	Elasticsearch
Data ingestion strategy	Active (polling - scraps data from endpoints) Passive (push – applications push metrics)	Passive	Passive
Alerts	Yes	No	No
Targets autodiscovery	Yes *Integration with Consul*	No	No
Horizontal scaling	No *Scaling done through explicit sharding and federated architecture*	Yes	Yes

Data labeling	Yes	Yes	Yes
Integration API	HTTP REST	HTTP REST	
Query language	Yes	Yes	Yes
Data aggregation	Yes	Yes	Yes
Data filtering	Yes	Yes	Yes
Data downsampling	Yes *Can be emulated using combination of filters and aggregators, such as query_range and max_over_time*	Yes	No
Arithmetic binary operators	Yes	Yes *As with version 2.3*	No

4. Integration architecture

One conclusion we can draw by viewing Table 1 is that Prometheus can be used to actively collect metrics associated with microservices, support applications and those associated with he host and operating system. Because Prometheus does not scale horizontally, we will use OpenTSDB to store and analyze historical data. In this regard, Prometheus is configured (the remote_write directive) to push data to OpenTSDB using a remote storage adapter. Elastic APM can be used to store events/transactions using APM agents developed and integrated at microservice level. Additionally,we can capture exceptions and errors, including the stack trace.

For exemplification, let's consider a common application which is accessing a database and which has been decomposed into two microservices, each microservice having its own dedicated machine. The database server also resides on a separate machine. In Figure 6 we propose an example integration architecture for monitoring the microservices and associated infrastructure.

We have deployed two Prometheus instances:

- one dedicated to scraping software metrics by using in-house implemented exporters for the two microservices and an open source exporter for the database;
- one to scrap hardware metrics provided by node exporter, an open source exporter which provides statistics about CPU, disk IO operations, average loading, memory, network, file system, entropy, etc.

We have configured Prometheus instances to forward metrics to a deployed cluster of OpenTSDB instances for long term preservation of data. Old metrics are automatically purged from Prometheus after a period of time specified via storage.local.retention flag. Normally we would not want Prometheus to keep data more than one month, as we would be using OpenTSDB to do data mining and statistics.

We are using Prometheus to do real-time monitoring, such as triggering alarm events. For that we use Prometheus Alert Manager, a component that can be deployed separately and which handles alerts sent by clients, such as Prometheus and which supports deduplicating, grouping, and routing of alerts to the correct receiver integration such as email, PagerDuty, or OpsGenie.

Additionally we have deployed an Elasticseach APM server and Elasticsearch cluster database to store microservice events and transactions, along with exceptions raised during runtime.

In our example we have deployed two alert managers and we have integrated our own receiver which forwards the alerts to a Message Broker dedicated channel. This allows other applications that listen on the channel, to access the alert content and react.

The metrics from Prometheus and OpenTSDB can be visualized using Grafana, an open source analytics solution that supports easy integration with a large collection of time-series database engines, including Prometheus and OpenTSDB.

The events of the system will be monitored using Kibana, an open source data visualization plugin for Elasticsearch.

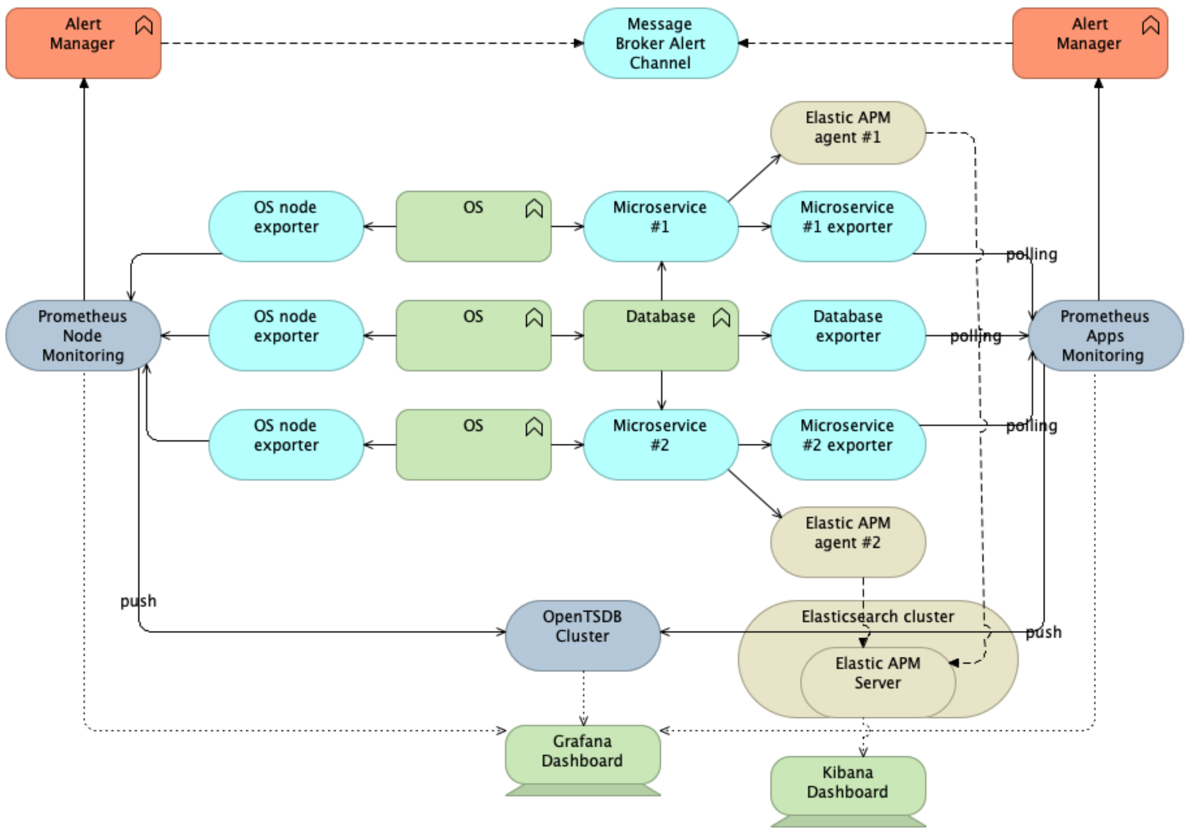

Fig. 6. An example of integration of Prometheus (used to store host and application related metrics), OpenTSDB (long term retention) and Elastic APM for capturing transaction details

In the example described above, instead of two Prometheus instances, we can use one. The same applies to the Alert Manager component. However, we wanted to highlight the logical separation of metrics as part of architecture, software vs hardware and infrastructure vs applications.

Host metrics

The OS node exporter provides with host level metrics such as:

- node CPU for each core as seconds the cpus spent in each mode (idle, user, system, iowait, guest, irq, etc);
- the total number of bytes read successfully for each physical device attached to the node;
- the total number of bytes written successfully for each device;
- the number of I/Os currently in progress;
- the time in miliseconds spent doing I/Os;
- the weighted of milliseconds spent doing I/Os;
- the he total number of milliseconds spent by all reads;
- the total number of reads completed successfully;
- number of reads merged;
- the total number of sectors read/written successfully;
- the total number of milliseconds spent by all reads/writes;
- bits of available entropy;
- filesystem space available to non-root users;
- filesystem size;
- node load average with intervals for 1

minute, 5 minutes and 15 minutes;

- physical and virtual memory used;
- current mapped memory;
- kernel stack size;
- free memory;
- network related metrics: number of packets and packet size for each protocol (TCP/UDP).

4. Conclusions

The transition from monolithic architecture to microservice oriented architecture requires a different approach when it comes to monitoring the applications, the support software and the infrastructure. As the complexity kicks in, due to service decomposition and distribution, standard monitoring solutions such as Nagios are not flexible enough to support the automatization of the monitoring process. Instead, multiple integrated and specialized solutions are needed, working in tandem to collect metrics and events from the dynamically distributed components across the IT infrastructure. Such a solution includes:

- Prometheus, a monitoring and trending system with autodiscovery capabilities, that can scrap metrics from targets and which provides a reach query language;
- OpenTSDB, a distributed, scalable, monitoring system that can be used to keep available the historical data for data mining and complex analytics, featuring Hadoop, a powerful framework that allows for distributed processing of large data sets across clusters of commodity computers using a simple programming model;
- Elastics APM, an application performance monitoring system built on the Elastic Stack. It allows you to monitor software services and applications in real time, collecting detailed performance information on response time for incoming requests, database queries, calls to caches, external HTTP requests, etc.
- Grafana, an analytics platform which features visualizations, alerts, notifications, dynamic dashboards, mixed data sources, annotations and ad-hoc filters.
- Kibana, a data visualization plugin for Elasticsearch.

Acknowledgment

This work was supported by the research programme Planul Sectorial M.C.S.I. Plan sectorial de CD în programul ”Agenda Digitală pentru România” – Secțiunea ”Servicii electronice 2018-2020”, project number 76/19.06.2018 „Studiu privind sisteme adaptive de recunoastere in stadii incipiente a atacurilor cibernetice asupra resurselor statale”.

References

[1] Allen Wang, Sudhir Tonse. Announcing ribbon: Tying the netflix midtier services together, January 2013. http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html

[2] Paulo Merson. Microservices Beyond the Hype: What You Gain and What You Lose. March 2018. https://insights.sei.cmu.edu/saturn/2015/11/microservices-beyond-the-hype-what-you-gain-and-what-you-lose.html

[3] Monolithic vs. Microservices Architecture, http://www.antonkharenko.com /2015/09/monolithic-vs-microservices-architecture.html

[4] R. Boncea, I. Bacivarov. A System Architecture for Monitoring the Reliability of IoT. In Proceedings of the 15th International Conference on Quality and Dependability, pp.143-150.

[5] R. Chen, S. Li, and Z. Li. From monolith to microservices: A dataflow-driven approach. In 24th Asia-Pacific Software Engineering Conference (APSEC), pages 466–475, Dec 2017.

[6] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, Today, and Tomorrow, pages 195–216. Springer International Publishing, Cham, 2017.

[7] S. Joshi. Organization & cultural impact of

microservices architecture. In M. Hoffman, editor, Advances in Cross-Cultural Decision Making, pages 89–95, Cham, 2018. Springer International Publishing.
[8] R. Perrey and M. Lycett. Service-oriented architecture. In 2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings., pages 116–119, Jan 2003.
[9] S. Prasad and S. B. Avinash. Smart meter data analytics using opentsdb and hadoop. In 2013 IEEE Innovative Smart Grid Technologies-Asia (ISGT Asia), pages 1–6, Nov 2013.
[10] T. W. Wlodarczyk. Overview of time series storage and processing in a cloud environment. In 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings, pages 625–628, Dec 2012.
[11] M. Richards. Microservices vs. Service-Oriented Architecture. O'Reilly Report, 2016.
[12] Paul C. Brebner, Performance modeling for service oriented architectures, Companion of the 30th international conference on Software engineering, May 10-18, 2008, Leipzig, Germany.
[13] A. Brunnert et al. Performance-oriented DevOps: A Research Agenda. Technical Report SPEC-RG-2015-01, SPEC Research Group -- DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), August 2015
[14] https://www.elastic.co
[15] http://opentsdb.net/
[16] https://prometheus.io/
[17] https://www.influxdata.com/

Radu BONCEA is a Researcher at *I.C.I. Bucharest*. He has been involved in several large European projects such SPOCS, eSENS, Cloud for Europe and The Once Only Principle. As a Ph.D. student at Electronics, Telecommunications and Information Technology, he’s interested in IoT and Cloud Computing related technologies.

Alin ZAMFIROIU has graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 2009. In 2011 he has graduated the Economic Informatics Master program organized by the Academy of Economic Studies of Bucharest and in 2014 he finished his PhD research in Economic Informatics at the Academy of Economic Studies. Currently he works like a Senior Researcher at “National Institute for Research & Development in Informatics, Bucharest”. He has published as author and co-author of journal articles and scientific presentations at conferences.