Final Report on:
Pretix Redundant Operation

Filipe Pires [85122], João Alegria [85048]

Computational Infrastructures Management
Department of Electronics, Telecommunications and Informatics
University of Aveiro

June 14, 2020

Introduction

This report aims to describe the work developed for the final phase of the practical assignment of the discipline of Computational Infrastructures Management (*I*) from the Msc. degree in Informatics Engineering of the University of Aveiro at the Department of Electronics, Telecommunications and Informatics. It is assumed that the reader has knowledge about the previous report, and it is here included: the characterization of the service level agreements for each service; the description of the load balancing mechanisms among software components; the redundancy strategy; the listing of metrics and computational resources monitored, along with the respective defined alarms; and other relevant aspects such as horizontal scalability and component fault tolerance.

The service provided is Pretix, an online shop, box office and ticket outlet already successfully used by other service providers for conferences, festivals, exhibitions, workshops and more. The previously delivered work focused on the product presentation, distributed installation, and resource and performance analysis. All code developed is publicly accessible in our GitHub repository:

`https://github.com/FilipePires98/GIC`.

1 System Architecture

1.1 Infrastructure

To provide the Pretix Ticketing Software (*2*) to the public, it was required to set up an infrastructure containing their web application, an instance of a Web Server Gateway Interface (WSGI) hosting the application and of a reverse-proxy for the web application deployment in production mode, as well as a database management system (DBMS) (where PostgreSQL was used) for handling disk storage and a caching server (where Redis was chosen) for in-memory storage and asynchronous queuing of tasks.

Our strategy is thoroughly described in the previous report, which already considered minimum redundancy to ensure availability. It is achieved through the usage of Docker (*3*) to isolate each component for greater control and easier configuration, and Docker Swarm (*4*) for the orchestration of the entire infrastructure. Nevertheless, as it was a still primitive solution, naturally several upgrades were applied, mostly internally to each container. The latest architecture is visually presented in Figure 1. This solution considers quality attributes whose assurance is described in the sections below.

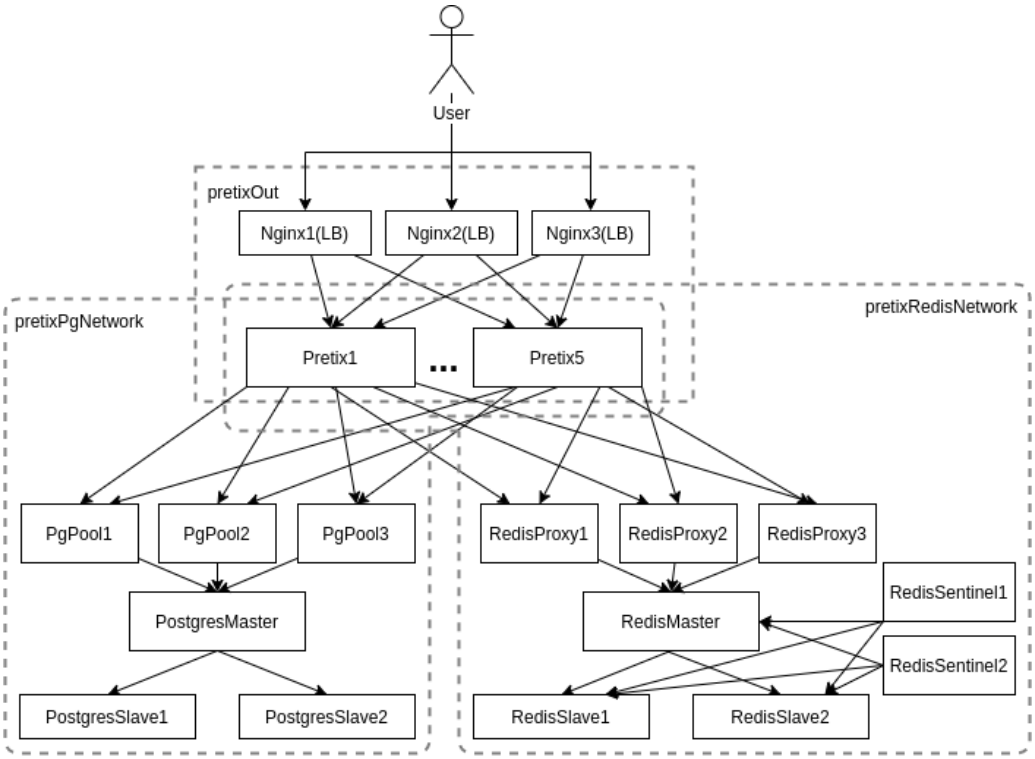

Figure 1: Infrastructure architecture diagram deployed in Docker Swarm.

1.2 Load Balancing

The knowledge gathered so far regarding the operation of Pretix affirms that user activity peaks are to be expected when in production. This has many implications and requires the implementation of mechanisms that maximize the infrastructure's efficiency.

With this in mind, we have come up with mechanisms for load balancing based on proxy servers. These servers, when strategically placed, act as intermediaries for requests seeking resources from the system itself. This not only reduces the complexity of requests made to our services, but also provides additional benefits with regards to security and load balancing, as they allow a controlled and intelligent distribution of requests among components.

An NGinX proxy (*5*) was installed between the end users and the Django-based web application. With the help of Docker Swarm, replicas of both the proxy and the web application were deployed. This is transparent for the user and it works as if he/she is sending requests to one single access point.

The storage clusters also resort to proxies. For the PostgreSQL server cluster (*6*), we adopted Pgpool-II (*7*). As our database is replicated in master/slave mode, Pgpool is able to take advantage of the replication feature in order to reduce the load on each PostgreSQL server by evenly distributing queries among available servers and consequently improving the overall throughput. Pgpool provides other useful features such as connection pooling, where it maintains established connections to the servers and reuses them whenever a new connection with the same properties comes in. For the Redis server cluster (*8*), the adopted solution was the well-known HAProxy (*9*). The instances installed spread requests evenly across the Redis cluster. Here, a master/slave mode was also used, meaning that the proxies interact with master instances that then delegate work among slaves.

Load tests executed during development proved that such additions had a significant impact on the performance of the system. The usage of networks, seen in Figure 1, where the proxies mentioned above are inserted help orchestrating and logically grouping components.

1.3 Redundancy

In order for us to provide Pretix, it is only required to maintain one instance of each component running. However, this is highly unadvisable as it susceptible to failure from any point, i.e. if any of the system components fails, the entire system goes down. This is referred to as a redundancy measure of N.

Ensuring actual redundancy of a system requires at least a measure of N+1, where each component can be replaced by another if required. In most cases this is achievable through the replication feature of Docker Swarm. Pretix's stack has K replicas of NGinX, of the Pretix web

application, of Pgpool proxy and of HAProxy, where K can be defined on deployment with a value greater than 2.

Redundancy was also the reason why a master/slave mode was adopted on both clusters. By instantiating 1 master and K slaves, it is possible to fulfill the requirements implicit to N+1 since if any slave fails the others can replace it and if the master fails a slave is elected the new master. Fortunately, for the PostgreSQL cluster this reelection process is done automatically, but for the Redis cluster the use of sentinels is required: sentinels observe the behavior of masters and slaves, detect failures and reassign roles when needed; here we insist on providing N+1 redundancy by having multiple instances of sentinels as well.

The redundancy supported also enables the system to be fault tolerant, since with the N+1 replication of each component, we guarantee that if any of of the replicas of our service fail and for some reason stops responding, the other replicas and the proxy and redirecting mechanisms ensure that our systems doesn't stop. This kind of failures are transparent for the end-user.

1.4 Scalability

Any project intended to be successful must consider the scalability of the product in the future. Ours is no exception since, if the ticketing platform becomes popular, the infrastructure must be prepared to respond to greater loads of income traffic.

There were 2 means to ensure scalability: vertical or horizontal. As we had access to limited computational resources and adding more physical RAM or CPUs was not an option, the only alternative left was to support horizontal scaling.

The idea here is to make possible the deployment of new replicas of any component and integrate them to the existing infrastructure without significant negative impact on insertion and in a completely transparent way to the users. This is exactly what we achieved. Additionally, each component identifies itself to the monitoring server on initialization so that the data related to what is being managed is constantly up-to-date.

Scaling a portion of the stack can be done both manually or in an automated way. By default we support manual scaling, but we found it was logical to implement automated mechanisms to scale up or down according to the online users and the resources used. This was achieved through the monitoring of meaningful metrics collected from each container instance, as we present in section 2.3.

2 Infrastructure Management

2.1 Service Level Agreement

As service providers, we are supposed to fulfill a commitment with our clients, whether it is formally defined or not. Obviously it is a good practice to elaborate an agreement where particular aspects of the service are agreed between the service provider and the service user - this is called a Service Level Agreement (SLA). We describe ours in this section.

Considering the scope of our project, we've defined the following commitment key points:

- Pretix is to be provided through web browsers, by accessing a particular address, and it will only be accessible to those within the university's virtual private network (VPN).
- Its usage is fully dependent on the usability of the Pretix service itself and no complexity should be added by the infrastructure we provide with regards to this.
- We are responsible for providing free access to the ticketing platform through our servers and keeping data integrity even when the service is unavailable
- The installations hosting our deployment belong to a third party (the university) and thus we are not accountable for possible availability issues related to networking and machine up time. Nevertheless, if this is not considered, we ensure an availability of 100% under normal circumstances and of 70% for high activity peaks [1], with a maximum response time of 50 seconds for ticket purchasing requests.
- No support is guaranteed for issues about features related to the internal implementation of the ticketing software.

It is worth mentioning that our analysis to the (free version of the) Pretix product showed that the software had strong limitations on maximum load capacity and response time that, according to the authors, could not be solved through horizontal scaling.

Focusing on the infrastructure itself and the computational resources, we have defined a list of Service Level Objects (SLO) - the metrics to be observed - representative of the stack's state, and a list of Service Level Indications (SLI) - the thresholds or functions applied to the metric values - that will help us react to situations where the SLA compliance is threatened. These SLOs revolve around: hardware capacity, operation latency, availability percentage and reliability.

Table 1 presents a simplified view on the metrics used to monitor the SLA and some thresholds that aid on the prevention of a cop-out. Note that the chosen metrics are dependent on what is made available by each component.

[1] Over 50 requests per minute.

Category	Metric	Upper Threshold
Hardware (SNMP)	System Up Time	-
	% User CPU Time	90%
	Total RAM	-
	Total RAM Used	-
	% RAM Usage	90%
	Total Disk	-
	Total Disk Used	-
	% Disk Usage	90%
Pretix	Total # of Events	-
	Total # of Orders	-
	Total # of Purchases	-
	Response Time	50 seconds
NGinX	Total # of Requests	10% (failure)
	Avg. Response Time	50 seconds
Redis (& HAProxy)	Redis Up Time	-
	Memory Used	-
	# of Connections	-
	# of Connected clients	-
	Avg. Response Time	30 seconds
PostgreSQL	Total # of Queries	-
	Total # of Failures	-
	Max. Transaction Duration	25 seconds

Table 1: Monitored metrics, categorized by source.

2.2 Monitoring

Understanding the state of the infrastructure and components is essential for ensuring the correct functioning and stability of the service. Information about the health and performance of deployments not only helps us react to issues, it also gives security to make changes with confidence. One of the best ways to gain this insight is with a robust monitoring system that gathers logs and metrics, visually displays data, and alerts us operators when undesired events occur or when the maximum capacity of some resource is almost reached. This kind of system also allows the definition of reactions for specific SLA violation threats.

Our monitoring system is hosted on an independent machine with 32 cores of 2.1GHz, 64Gb of RAM and 7Tb of disk space. Just as the infrastructure hosting our Pretix stack, this too was shared amongst other operator teams. The virtual machine (VM) we instantiated for our monitoring system runs Ubuntu 20.04 LTS and has 2 CPU cores, 4Gb of RAM and 50Gb of disk space. The separation between our product and its monitoring was understood to be optimal, since this way there is no added load to the service facility, keeping its resources fully dedicated to itself. There is also the advantage of allowing the deployment of our system both with or without monitoring enabled.

2.2.1 Log Collection

There are 3 major reasons for keeping log messages of software. For developers, logging tools are often deployed in places where they don't have access or there are no debugging tools, but log messages can still help to localize problems. For operators, or system administrators, they provide insight on the state of a group of connected machines, thus helping them to make sure all run as smoothly as possible. But perhaps the reason with greatest impact is security. Log monitoring systems oversee network activity, inspect system events, and store user actions that occur inside the operating systems. Reviewing this information is a strong means of detecting suspicious activity and identify the damage caused in case of an attack.

A centralized point of log monitoring simplifies this process of message analysis, so we installed a platform in our monitoring VM responsible for collecting log files from each component, storing them and presenting the data in an organized, visually useful and empowering form. There are a few options to achieve this and after some brainstorming and research, we found the Elasticsearch stack (ELK) (*10*) to be the most appropriate solution for our scenario. This stack comprises 3 open-source projects, each with its own purpose:

- Elasticsearch - a search and analytics engine, optimized for textual data such as logs; logs are stored in it.
- Logstash - a data ingestion tool, that serves as the stack entrypoint; responsible for collecting, transforming and sending data to a desired destination.
- Kibana - a data visualization and exploration tool for reviewing logs and events; enables the creation of dashboards, alerts and more.

The first approach attempted used rsyslog (*11*). This technology allows the definition of a server and multiple clients, each one assigned to a service that gathers the system logs of the machine and sends them to the predefined rsyslog server. This would provide us with centralization, but several integration issues were faced. Soon we found that it was not an optimal solution, since when receiving exterior logs, the rsyslog server persists them in the system logs file of the hosting machine, resulting in a duplication of logs both on hosting machine and on Elasticsearch (due to its indexing process).

The second approach ended up being much cleaner and simpler. After some research we found that one of the logging drivers provided by the docker platform was syslog (*12*). Syslog enables the redirecting of all logs to a syslog server specified in the driver options - which was aligned with our scenario. We also found that Logstash allows the creation of a server that can receive any type of message (since filters can be applied to the incoming messages), enabling the direct connection between the docker syslog driver and the ELK stack.

After the logs arrive at Logstash, the internal ELK flow was configured so that Elasticsearch indexes the information and Kibana is able to access it. The system is thus capable of collecting

all logs and present them in the Kibana interface. Further processing over collected logs can be made if the information is context specific and allows these type of processing, such as counters or other metrics. However, in our case this was found unnecessary since we were able to obtain metrics from all the services necessary. Figure 2 shows an example of the Kibana GUI.

2.2.2 Metric Collection

Metrics monitoring and alerting are all interrelated concepts with the ability to provide visibility into the health of a system, to help you understand trends in usage or behavior, and to understand the impact of changes applied. If the metrics fall outside of expected ranges, a monitoring system that incorporates such concepts can send notifications to prompt an operator to take a look, can then assist in surfacing information to help identify the possible causes, and can even have predefined response measures triggered on specific states.

Centralizing metrics has the same benefits as doing so with log messages. However, metrics do not provide a historical record of the system's behavior, rather they present in great detail the current state of the system, mostly through numeric data like counters, gauges and service or context-specific indicators.

Naturally, there are a few approaches and technologies that allow metric collection. After some research, we found some that are commonly used for this purpose, such as Nagios, Zabbix, Munin, TICK stack, Prometheus and more. We resorted to Prometheus (*13*), since our product Pretix - our main service - already exported metrics in a format compatible with Prometheus.

For the remaining services we installed metric exporters parallel to the services themselves and compatible with Prometheus. Since our infrastructure was deployed using docker in swarm mode, the addition of exporters was made by changing the Docker files and respective entry points for each service. The mapping between service and installed exporter is:

- HA Proxy - Official Statistics Exporter.
- Nginx - VTS Statistics Exporter.
- PostgreSQL Master/Slave - Server Metrics Exporter.
- Redis Master/Slave/Sentinel - Server Metrics Exporter.

An additional exporter was installed in the monitoring VM, so that hardware-related metrics could also be monitored, such as CPU, memory and disk usage, To achieve this, we resorted to the Simple Network Management Protocol (SNMP), an internet standard protocol for collecting and organizing information about managed devices on IP networks and for modifying that information to change device behavior. The protocol support was installed *a priori* on every compute node, being only necessary to request it's metrics when needed. However, this posed an additional challenge: since the deployment is made in Swarm mode, it was impossible for us to know where our services would be deployed in the Swarm computing network. Statically

indicating every computing node to be monitored would cause stress on machines and would not scale properly, so we implemented a dynamic IP address discovery mechanism.

This custom address collector has an independent service and was created using Python's Flask library, which enabled us to quickly develop the necessary endpoints and respective logic. At initialization, each service sends an HTTP Post to our custom service with its name, replication number and identifier. The collector service then registers or updates it's IP table and modifies the respective Prometheus target files for each type of exporter.

Our choice for the graphical user interface (GUI) to interact with Prometheus was Grafana (*14*), commonly integrated with Prometheus as it is the GUI indicated by the engine itself. Grafana allows us to explore the collected metrics, construct meaningful dashboards and configure critical alerts, as we will see further ahead.

Figures 3 to 10 are all populated examples of the created dashboards, one for each type of service, namely: Nginx, PostgreSQL, Redis, Pretix and the SNMP infrastructure monitoring.

2.3 Alerts and Automation

At this stage we had built an arsenal for the operation of our product. Guided by our SLA, we proceeded to defining meaningful alerts triggered by thresholds applied to specific metrics, and automatic reaction measures in order to prevent agreement breaches and provide flexibility to the whole infrastructure. All the generated alarms and notifications are sent to a dedicated Microsoft Teams channel and to a custom service responsible for processing and reacting. This service is none other that the IP address collector that, using Docker's Python SDK (*15*), is capable of automatically scaling up or down a service that triggered a received alarm. Following is a list of the implemented alarms and respective reaction behaviors:

1. NGINX - Pretix Response Time alert
2. PRETIX - Number of Events alert [for testing purposes]
3. PRETIX - Order POST Time Distribution alert - On these 3 alerts a new replica of the `pretix_web` is created, as they indicate an overload on the existing ones.
4. REDIS - # Connected Clients alert
5. HAPROXY - Proxy Average Response Time of Redis alert - As Pretix might not be the one causing slower response times, these 2 alerts indicate the Redis cluster is close to its maximum capacity and create a new replica of the `pretix_redisslave` to avoid it.

Alert number 2 was created for testing purposes only, this was our means to achieve a consistent and controlled manual trigger: by defining a low threshold and simply creating Pretix hosted events, the alarm was triggered and the notification channels and the automatic rescaling reaction mechanism could easily be tested. Figure 11 is an example of the alerts received in Microsoft Teams.

3 Additional Remarks

3.1 Swarm Stability

We have mentioned the fact that all remote computational resources were shared among developer/operator teams and naturally this had consequences in terms of performance and accessibility, with activity peaks close to delivery dates and occurrences of human errors with generalized consequences. Additionally, due to external limitations, maintenance also suffered some difficulties and access to the machines was not 100% assured.

To partially mitigate these limitations, much of our tests and debugging was executed with local deployments to allow an independent and frictionless evolution in each system's component. Nevertheless, often we were forced to postpone necessary remote tests due to unexpected errors related to the instability of the connection to the remote hardware and lost time doing research on a problem who's solution wasn't at our reach. These problems were predominant in the PostgreSQL cluster, since internally it's components need to establish a replication scheme by exchanging control messages through the network; inconsistencies in these exchanges would then prevent the correct initialization of the cluster. This was a reality we had to face during the project lifetime, but was fortunately reduced thanks to the efforts of those responsible for the assignment.

3.2 Assignment Contributions

Due to external issues that prevented physical proximity among students and professors, a confined approach had to be adopted during this second development phase. Nevertheless, the close-contact strategy was kept as this project required a broad perspective on the infrastructure and the tasks to be implemented by both team members. Regarding the work distribution among developers, tasks were assigned to each according to a predefined plan and a common working schedule was applied.

The technology-related decisions were first made in conjunction. Then, Filipe set up the monitoring VM to gather the data from the stack while João began to work on the mechanism to collect the logs from each container. Filipe moved on to installing metric exporters on each container, and João found means to collect lower level metrics using the SNMP. The container identification mechanism of each instance to the monitoring system was then implemented. At this stage, both operators worked on defining the SLA and João designed the monitoring dashboards. Finally, the alarms and automated response mechanisms were implemented. It is needless to say that bug and error solving was made along the development phase by both developers any time it was required. Once a stable deployment fulfilling all predefined requirements was completed, this report became our primary concern, with both contributing equally.

Conclusions

With the work conducted during these two development phases, we were able to achieve a robust solution for the remote automatic deployment, hosting and operation of our Pretix stack in production mode. Our delivered code provides a system who's architecture is fully redundant, tolerant to failures, scalable and with a level of automation of great value for the system administrators. Moreover, it provides an organized and complete set of monitoring tools as a centralized platform using ELK stack, Prometheus and Grafana, in an independent access point.

It is our belief that now it is possible to continue the development of the service while ensuring the correct maintenance of what is deployed, i.e. not only is the infrastructure prepared to host online events, but it is also prepared to increment updates to the internal logic without compromising the overall stack of services.

For future work, the strategy would be to first conduct trials with real users in order to learn more about the system's behavior and determine what areas would benefit from additional automated response measures. Although the offered monitoring dashboards already have great detail over the collected data, personalizing them to the specific case of Pretix would also help on providing end users with a better experience. Lastly, the limitations of the Pretix service itself should also be addressed in order to improve the user load capacity for activity peaks and take full advantage of the scalability potential of our clusters.

The perspective of operators is perhaps the most valuable knowledge we take from this project. Not only it empowered us with the capabilities to address similar projects in the future, it also made us more aware of aspects to take in consideration as software developers in order to improve and take full advantage of the DevOps environment.

References

1. J. P. Barraca, *GIC - Report no.2: Redundant Product Operation*, University of Aveiro, 2019/20.
2. *About Pretix*, `https://pretix.eu/about/en/`. Pretix.eu, retrieved in June 2020.
3. *Docker Homepage*, `https://www.docker.com/`. Docker Inc., retrieved in June 2020.
4. *Docker Docs: Getting Started with Swarm Mode*, `https://docs.docker.com/engine/swarm/swarm-tutorial/`. Docker Inc., retrieved in June 2020.
5. *NGinX News*, `https://nginx.org/`. NGinX.org, retrieved in June 2020.
6. *PostgreSQL: The World's Most Advanced Open Source Relational Database*, `https://www.postgresql.org/`. The PostgreSQL Global Development Group, retrieved in June 2020.
7. *Welcome to Pgpool-II*, `https://www.pgpool.net/mediawiki/index.php/Main_Page`. PgPool Global Development Group, retrieved in June 2020.
8. *About Redis*, `https://redis.io/`. Redis Labs, retrieved in June 2020.
9. *HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer*, `https://www.haproxy.org/`. HAProxy.org, retrieved in June 2020.
10. *What is the ELK Stack*, `https://www.elastic.co/what-is/elk-stack`. Apache Software Foundation, retrieved in June 2020.
11. *RSyslog: The Rocket-Fast Syslog Server*, `https://www.rsyslog.com/`. Adiscon GmbH, retrieved in June 2020.
12. *Docker Docs: Configure Logging Drivers*, `https://docs.docker.com/config/containers/logging/configure/`. Docker Inc., retrieved in June 2020.
13. *Prometheus: from Metrics to Insight*, `https://prometheus.io/`. The Linux Foundation, retrieved in June 2020.
14. *Grafana: the Analytics Platform for all your Metrics*, `https://grafana.com/`. Grafana Labs, retrieved in June 2020.
15. *Docker SDK for Python*, `https://docker-py.readthedocs.io/en/stable/`. Docker Inc., retrieved in June 2020.

A Kibana and Grafana Populated GUI

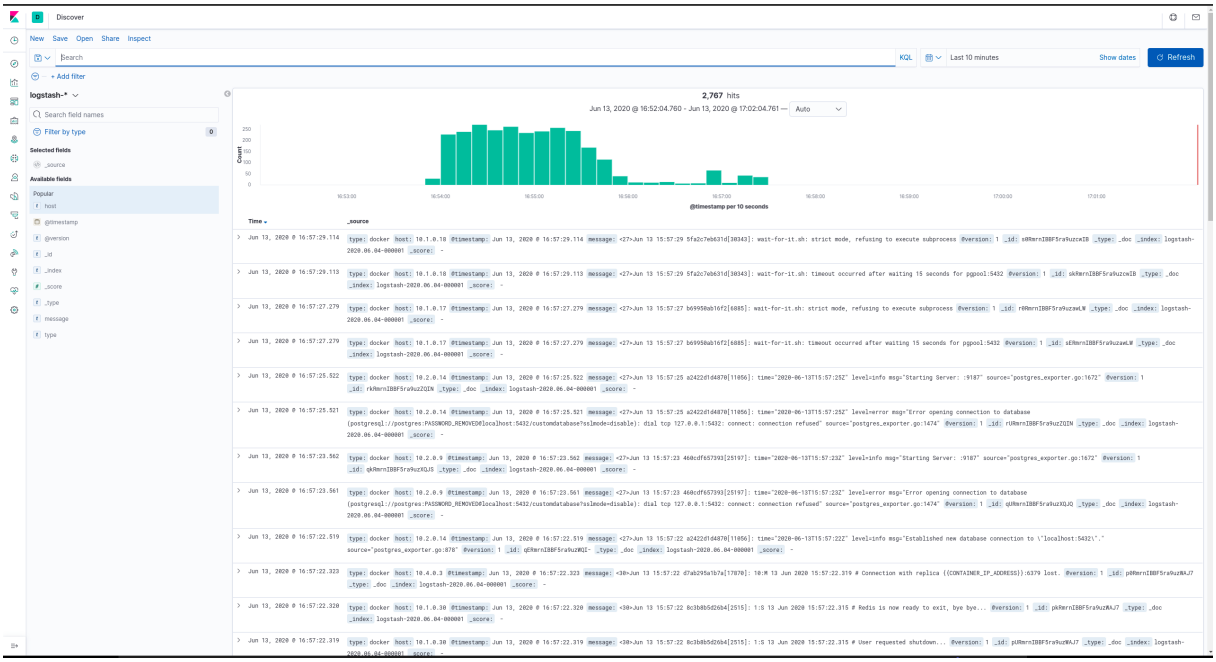

Figure 2: Populated Kibana GUI.

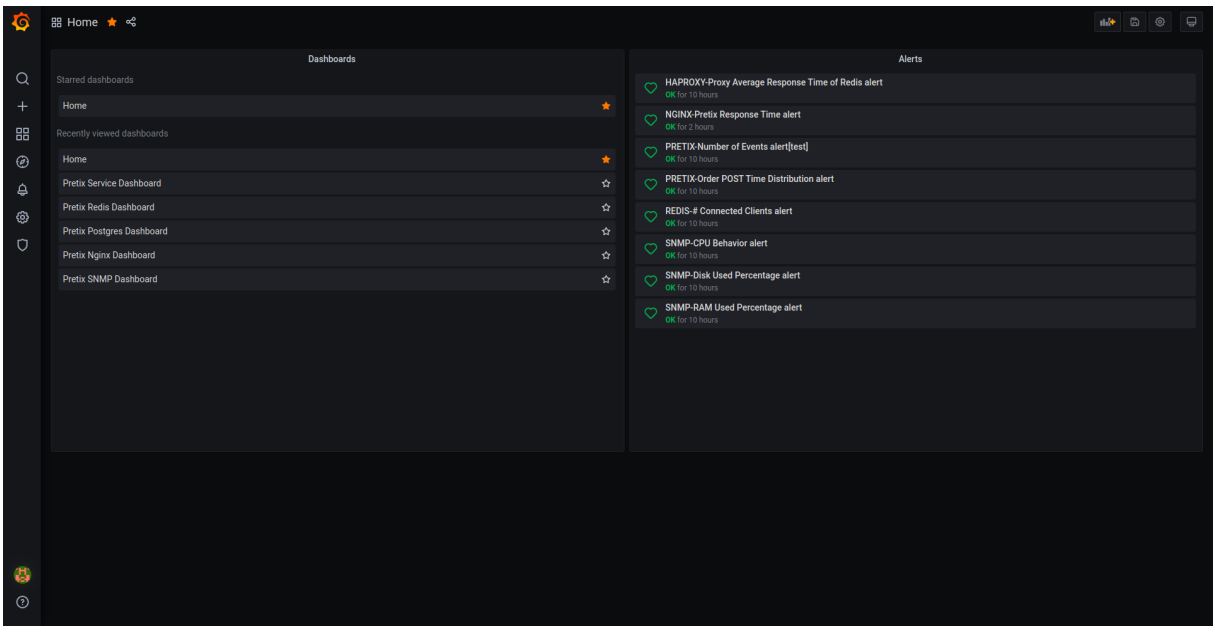

Figure 3: Grafana Custom Home Dashboard.

Figure 4: Populated Pretix Service Dashboard.

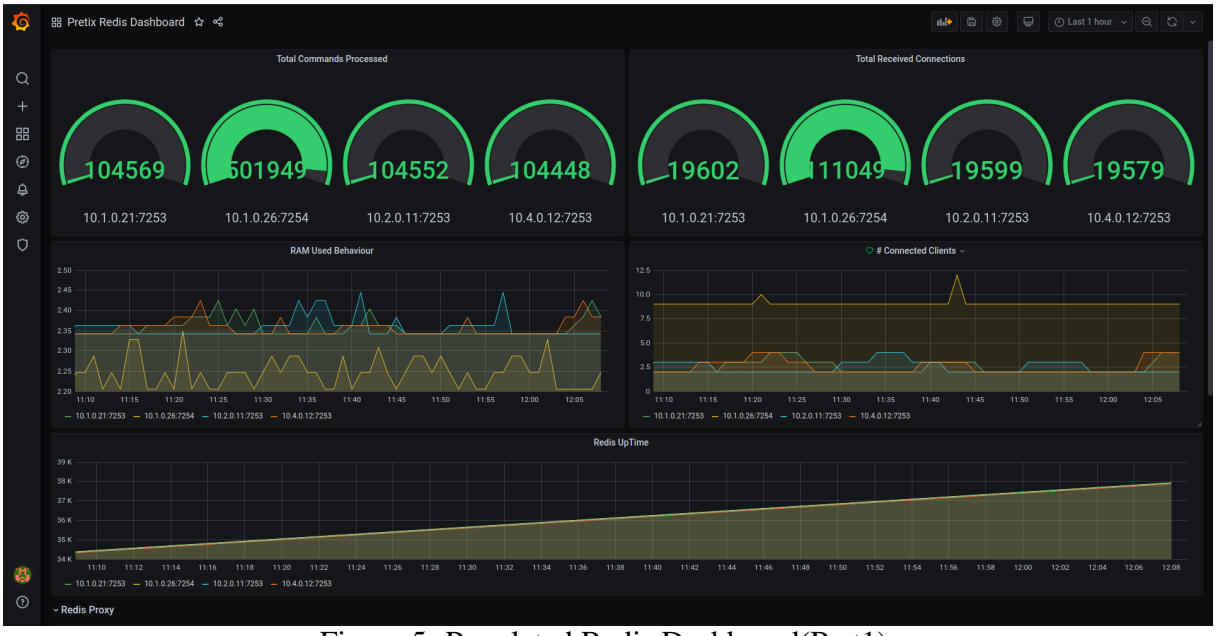

Figure 5: Populated Redis Dashboard(Part1).

Figure 6: Populated Redis Dashboard(Part2).

Figure 7: Populated PostgreSQL Dashboard.

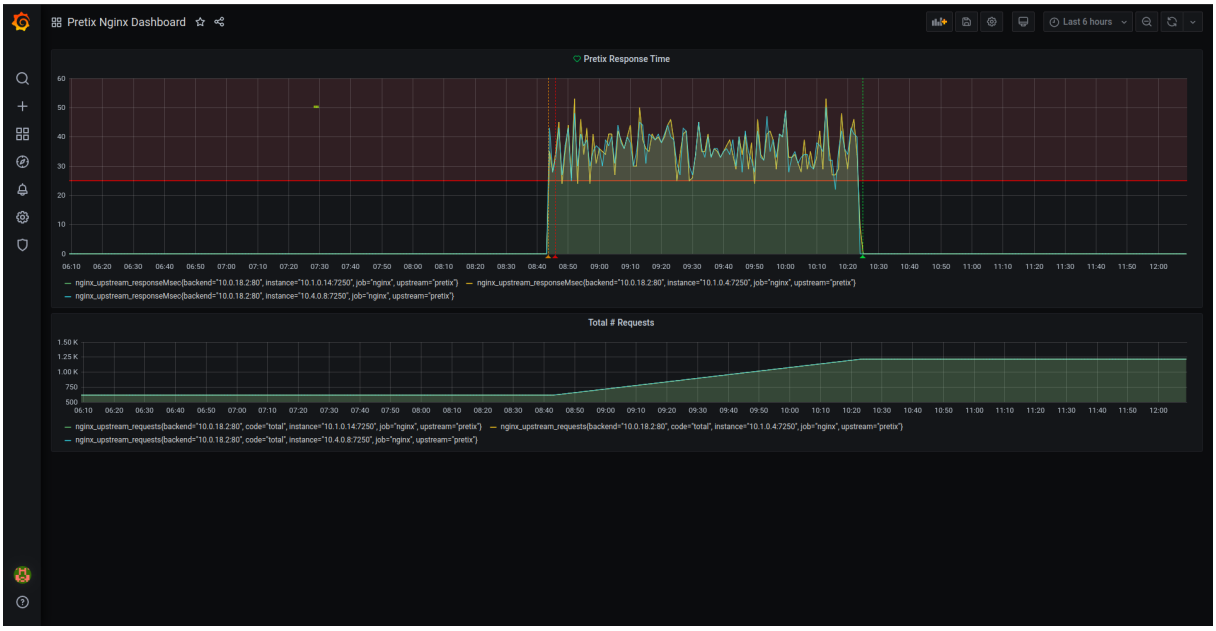

Figure 8: Populated Nginx Dashboard.

Figure 9: Populated SNMP Dashboard(Part1).

Figure 10: Populated SNMP Dashboard(Part2).

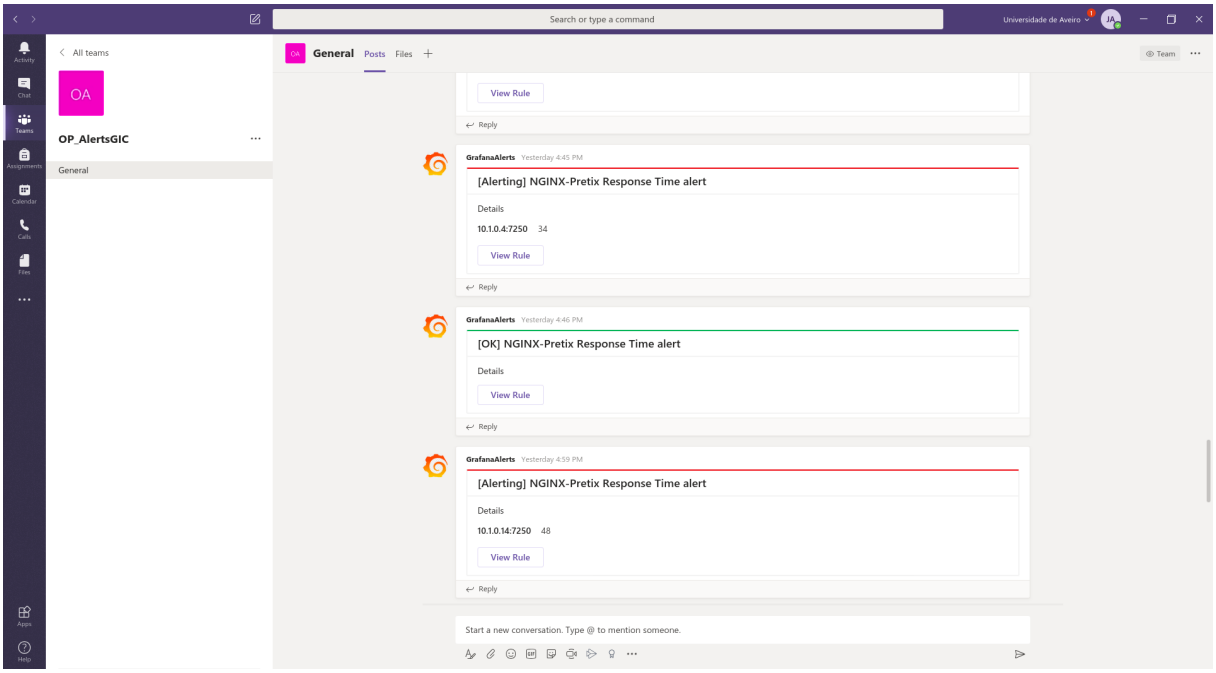

Figure 11: Microsoft Teams alert examples.