REST APIs Development and Performance Evaluation on the ICELab Kubernetes-based Architecture

Fabio Chiarani - VR445566

***Abstract*—The following document shows how a REST APIs server was added to increase the interoperability of the ICE lab architecture, based on Kubernetes, and how they were tested obtaining 56k req/s for HTTP PUT calls, and 120k req/s for HTTP GET calls.**

I. Introduction

The state-of-the-art ICE lab has an architecture for data collection based on Kubernetes and OPC-UA protocol. The scope of the project is to add a RESTful server exposing some REST APIs to increase the number of clients that could interact with the architecture.

After that, a tool (HIVE) was created to test the performance of the architecture using the REST APIs developed. The next sections will explain the architectural choices for adding the REST APIs, the tests performed and finally the results obtained.

II. Background

A. How Kubernetes came into the existence

Before, containers were the best concept to deploy applications. It gave a new horizon for developing and maintaining software. With containers, it was easy for the software developers to package up an application including the components like libraries and other dependencies. It can ship a package as a whole without the need for a traditional virtual machine. These microservices enabled users to individually scale key functions and have the ability to handle millions of customers.

But, once the application gets matured and complex, there will be a need to run multiple containers across multiple machines. You need to figure out which are the right containers and at the right time of course, how they can communicate with each other, tackle the large storage need, and deal with a failed container. Doing all this manually can be a nightmare!

Hence, to solve the orchestration needs of the containerized application, Kubernetes came to be. [1]

B. What is Kubernetes?

Kubernetes is an open-source container management system that is used in large-scale enterprises in several vertical industries to perform a mission-critical task. Some of its capabilities include [2]:

- Automated rollouts and rollbacks
- Storage orchestration
- IPv4/IPv6 dual-stack
- Horizontal scaling
- Service Topology routing
- High Availability
- Designed for deployment

C. What is a REST API?

Let's say you're trying to find videos about Batman on Youtube. You open up Youtube, type 'Batman' into a search field, hit enter, and you see a list of videos about Batman. A REST APIs works in a similar way. You are searching for something, and you get a list of results back from the service you're requesting from.

An API is an application programming interface. It is a set of rules that allow programs to talk to each other. The developer creates the API on the server and allows the client to talk to it.

REST is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in a variety of ways. It stands for 'Representational State Transfer'. It is a set of rules that developers follow when they create their API. One of these rules states that you should be able to get a piece of data (called a resource) when you link to a specific URL.

Each URL is called a request while the data sent back to you is called a response. [3] [4]

III. ICE Lab Architecture

Currently, at the state of the art, the ICE lab uses a data collection architecture based on Kubernetes and OPC-UA with the goal of performing monitoring operations, logging, and analyzing the plant status.

ICE lab has three separate servers on which to keep the architecture active. Specifically, they use k3s, a distribution of Kubernetes created ad hoc for IoT and Edge Computing, and is it running on the following hardware configuration:

- Intel Core i5-3470
- 3x 8GB DDR3 1600MHz
- Ubuntu Server

And two edge boards:

- Atom E3940 1Ghz 4core
- 2GB RAM

Meanwhile the figure 1 shows an high-level view of the ICE lab data collection architecture.

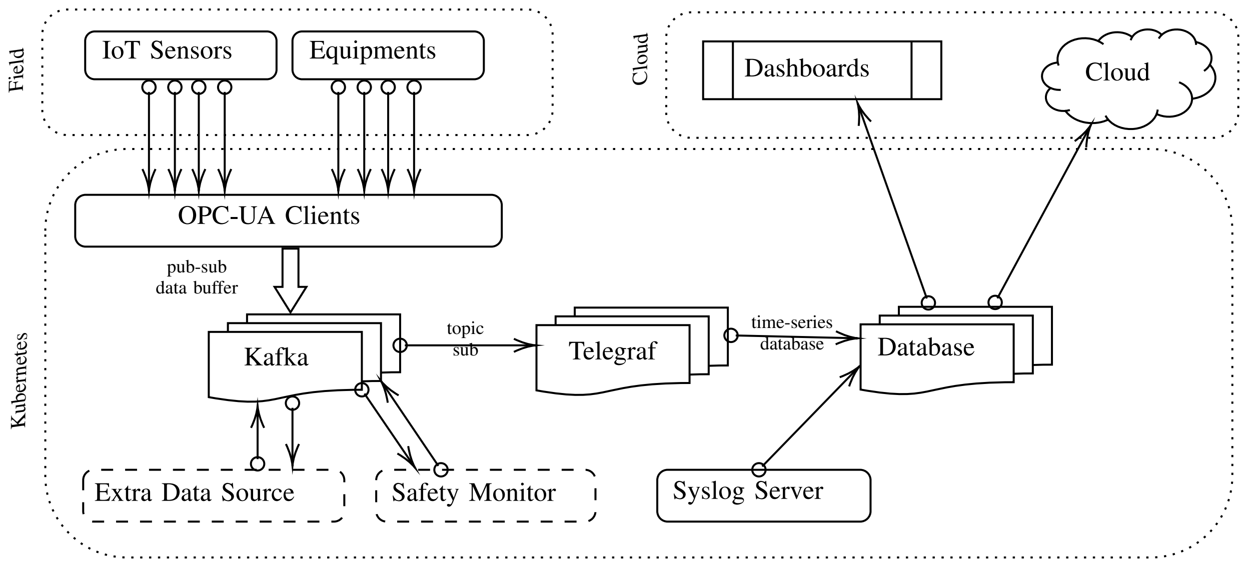

Figure 1. A high-level view of the state-of-the-art ICE lab data collection architecture

Is possible to see three different areas: the *field* area, where data is generated by the various clients (such as the Kuka robot, the production line, and/or the 3d printers) which use the OPC-UA protocol to communicate with the architecture; the *kubernetes* area, where all the core architecture and data collection services reside; and the *cloud* area, where the dashboards and analysis tools from *PaaS* are located (e.g. Siemens cloud).

IV. ICE LAB Data flow

Referring again to the figure 1, it is quite clear to understand what is the data flow: OPC-UA clients generate data and communicate it to the respective server, i.e. a pod running an OPC-UA service initialized within the k3s architecture. This data, subsequently, passes within the heart of this architecture: Kafka. Apache Kafka [5] is an event stream platform, which combines three key capabilities so you can implement your use cases for event streaming end-to-end with a single battle-tested solution, e.g.:

- To publish (write) and subscribe to (read) streams of events, including continuous import/export of your data from other systems;
- To store streams of events durably and reliably for as long as you want;
- To process streams of events as they occur or retrospectively.

So, Apache Kafka, thanks to the public-subscriber protocol, become the main broker of the architecture: all clients that will send data to Kafka will be publishers, as in this case, the OPC-UA server clients, while, clients like the safety monitor, or Telegraf, which will consume data, will be consumers.

In order to take the data from Kafka and put it on the InfluxDB database, a tool was needed to act as a consumer and redirect the data to InfluxDB. For this purpose, Telegraf is used.

Telegraf is a server agent written in Go that relies on plugins to collect and send metrics from a variety of input systems to InfluxDB. It allows you to collect data written in different formats and is very versatile, as to add a new input system, you simply need to download the relevant plugin and edit the configuration file.

In our case, Telegraf is in charge of consuming topics from an Apache Kafka server using the related plugin and uploading the data to a dedicated InfluxDB bucket.

The data flow then ends up in InfluxDB, a time-series database that can boast of being able to handle up to 1 million writes per second [6]. Other applications (such as ICETOM [7]) will then upload the InfluxDB data to a cloud (e.g. Siemens Cloud) for data analysis or dashboards.

All nodes, within the area Kubernetes visible in figure 1, have the ability to take advantage of the replication techniques, which allow you to quickly scale horizontally the same pod. This leads to better performance and self-balancing thanks to k3s when a node is too overloaded.

V. our addition to the architecture

Being an architecture currently working only for clients that can communicate on the OPC-UA protocol, it was thought to include a *RESTful* server, to make available some REST APIs and increase the interoperability of the architecture, giving the possibility to integrate new clients that do not support the OPC-UA protocol.

This server will not modify the architecture, but it becomes a new microservice added.

The figure 2 shows highlighted in blue our contribution to architecture. First of all, is chose to implement a *RESTful* server, to give the possibility to different clients to communicate with the architecture through the HTTP protocol (1.1 or 2), but not only: in fact, the *RESTful* server could implement in the future different REST APIs for other uses, such as, for example, to provide an interface to the safety monitor, or other future applications allowing to interact through a well-known protocol, or use some real-time

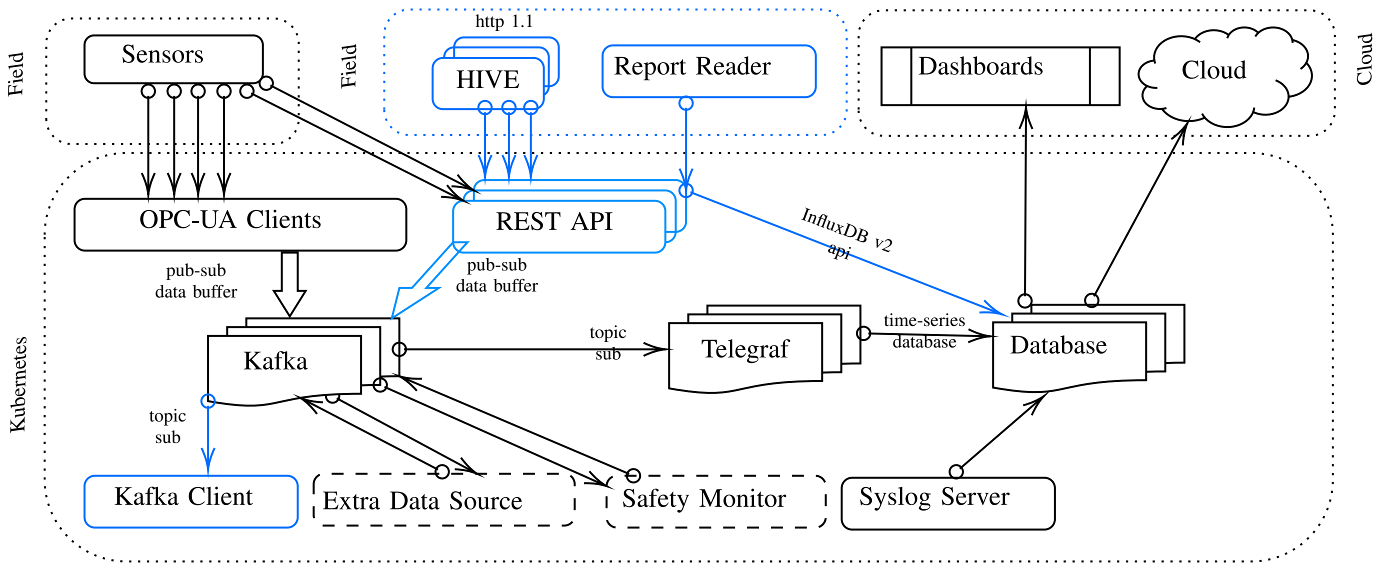

Figure 2. Our addition to the ICE architecture: a RESTful server to expose REST APIs, a HIVE tool to generate HTTP requests, and a report-reader to display benchmark results.

communication, for example, *socket.io*.

The *RESTful* server will interact with the heart of the architecture, the Kafka Broker. The Kafka-client, the HIVE node, and the report-reader have been developed to perform performance testing of the architecture, which will be described in detail in the following chapters.

Once the REST APIs to implement in the first place are identified, the question meant to be answered is ”*with respect to the state of the art, what performance is possible to achieve on this architecture?*”.

Before giving an answer to the performance evaluation, the architectural choices, and how the benchmark tool has been carried out are analyze in more detail.

A. Why fastify

An initial analysis was done to determine which technology was appropriate to create the RESTful server. Since there are several languages, some more powerful and some less so, it has been decided to follow the following parameters to evaluate which was the best to choose:

a) Community: It is known know that before evaluating a library it is important to observe the community around it. A large and active community is a good indication that the library is supported and valid.

b) Learning Curve: An important factor that led to the discrimination of the chosen language over the others was the language learning curve.

c) Performance: When we asked ourselves, ”*for the same language, which one is the better library?*” we relied on benchmarks and, if any, on real application scenarios.

d) Deployment speed: How much time, and how many resources the language and the specific library require to deploy.

Since most of the libraries for creating REST APIs are open-source, the first thing that was done was to filter for the language in which the APIs is written, and later, with which specific library implement the RESTful server.

Having free will, it has been chosen to proceed with the Node.js language, which allows a very fast writing and deployment on all platforms, regardless of the operating system.

In our specific case, it would have been deployed to an *alpine* docker image anyway, and it would have been fine. Node.js is a single-threaded language, which means that it does not use 100% of the available hardware, but despite this, using the concept of fork processes (aka multiprocess) is still possible to get very good performance.

On GitHub [8] is possible to see in a repo a performance test between various web server libraries, where there is the possibility to observe that for the Node.js language there were valid some libraries, such as *nanoexpress*, *polkadot* and *fastify*. Our choice of the library, based on the experience in having already used it, and on the parameters mentioned above, fell on Fastify, an Italian open-source RESTful server library.

Fastify [9] is a server built for high performance, an efficient routing (the same used by Netflix: *find-my-way*), the possibility to extend it with middleware (e.g. authentication, logging, swagger, and more), and the possibility to have validation and serialization with schema.

The project followed a robust, standard writing style to allow for the simplest possible maintenance, testing, and future additions. Since Nodejs is a language that allows writing in multiple styles, the repository has been structured with a CI/CD that will lint the code against the AirBnB style and run tests before committing. In this way, it is possible to force a robust and standardized writing of the code also in the future modifications.

As for the REST APIs definition and documentation, thanks to the swagger (*OPEN-API SPEC*) middleware on Fastify, the doc is autogenerated, and it is possible to export it both in *yaml* and *json* format. There is no way to access it through the GUI, available on the REST APIs `/documentation` endpoint. All specific documentation is available in the repository.

At the state of the art, there are two APIs within the RESTful server: one to redirect a payload to Kafka, and one to perform queries on InfluxDB, both of which are used by the HIVE tool to perform performance testing.

VI. How we benchmarked the architecture

To perform the performance tests of the architecture, two tools have been developed. As you can see from the figure 2: the first one is HIVE, a tool written in Nodejs that generates `HTTP/1.1` calls and produces a performance report for what concerns the communication between client and RESTful server, and subsequently, a tool written in python has been developed, in the figure 2 called *report-reader*, that performs some queries on the database to calculate that *a)* all the data arrived on the database and *b)* with what time delay they arrived.

Another interest was to observe the delay directly from the Kafka broker, and so a Nodejs node was added to listen to the topics and calculate the delay.

Now it will be described how the two tools were built and how and how it was possible to calculate the delay times.

A. HIVE

HIVE wraps the autocannon [10] library to generate `HTTP/1.1` requests. It is possible to execute HIVE in two ways: in static mode, where you set the parameters of concurrent connections, size of the HTTP pipelining, and the number of connections you want to execute per second, or in dynamic mode, where the various parameters are tested in combination with each other, to find out later which configuration has been able to create more traffic, and therefore more stress to the architecture.

The algorithm used, in dynamic mode, is:

```
func runDynamic(
    conn,
    maxConn,
    connOffset,
    pip,
    maxPip,
    pipOffset,
    wrk) {
        for( conn = 1;
             conn <= maxConn;
             conn += connOffset) {
               for( pip = 1;
                    pip <= maxPip;
                    pip += pipOffset) {
               run POST_REQ(conn, pip, wrk)
            }
        }
    }
```

The payload used is in InfluxDB-inline format. In order to calculate the time difference between when the data is generated and when it arrives on the DB, it's exploited the fact that when you send an InfluxDB-inline payload to InfluxDB, if you don't specify the timestamp, it's generated at reception himself.

It's then inserted a field called `hivets` containing the timestamp generated by HIVE, were then later once arrived on the DB, InfluxDB generates the `_time` field automatically. It will be then the report-reader that through a query calculates the delay of these two fields.

The HIVE tool will generate the maximum number of requests possible based on the resource capabilities of the client. In our case, having a high-performance machine, there were no no problem generating too few requests per second to stress the REST API.

The HIVE tool, once the session of generating `HTTP/1.1` POST calls to the REST APIs is finished, it generate a report in *json* format where it store some information about: number of successful connections (`2xx` status code), number of errors (non `2xx`), a distribution in percentiles of latency, number of requests and throughput obtained per second.

B. Report-reader

The *json* file generated by HIVE is collected and analyzed by the *report-reader* tool, that combines the *json* data with the obtained by calculating the delay time from HIVE to InfluxDB (using a query), generating a human-readable report and graphs to visually analyze the progress of the test.

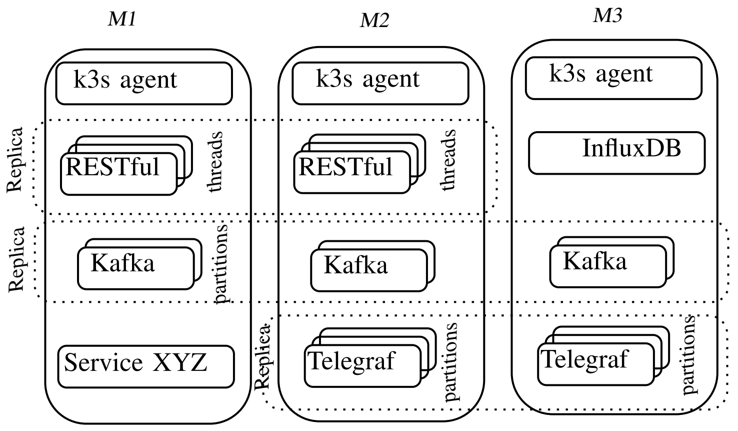

Figure 3. A topological view of how, with respect to the 3 servers in the ICE lab, how the various services are instantiated. For example, for Kafka, having an instance on all 3 servers, the k3s agent will act as a load balancer to distribute the traffic among the various servers. (M1, M2 and M3).

VII. Performance Evaluation

Figure 3 shows at the topological level how k3s could distribute the various pods and replicas among the three servers during the test execution. In fact, the tests were run starting with a distribution of pods without replicas and multithreads, and later augmenting and combining the parameters.

Two types of tests were then identified: the first type, using HIVE in a static manner, to test given the specific configuration of HIVE how the architecture response behaves with respect to the configuration of the pods. The second test, instead, was to execute HIVE dynamically, in order to perform a stress test of longer duration and with a higher number of requests once the configuration of the architecture to be tested was identified.

The purpose of the test, was to be able to find the load limit that the REST APIs can handle, but not only: in fact, another goal is to find out how is possible to reduce the delay between the REST APIs and the InfluxDB database, exploiting replication and topic management between Kafka and Telegraf.

The test run was done in one day, so targeted testing had to be done based on HIVE and architecture configurations. Table I shows the results as number of requests per second handled by the RESTful server by the HIVE configuration and architecture parameters variance.

VIII. Results

The table I shows the tests carried out (about 28 in total) with the relative performances obtained. All tests are for a maximum duration of 10 seconds in both static and dynamic HIVE mode. The columns *HIVE.wrk*, *HIVE.maxConn* and *HIVE.maxPip* refer to the parameters for dynamic execution of the HIVE tool. The remaining 3 columns instead refer to a static execution. It can be noticed in fact that they are mutually exclusive.

One thing that should be noted is that the estimate of the delay from HIVE to the Kafka client connected directly to the broker, is not precise (column *K Delay*): the client has been written in Nodejs language, using only single-thread execution. It has been seen that during testing that the core used was always at 100%, indicating that in all likelihood the data would be queued creating an additional delay. It should be noted, however, that in all tests where the machine running the Kafka client subscriber was not especially under stress, the calculated delay was below one second, indicating that the Kafka broker had no traffic flow problems.

Regarding the delay from HIVE and InfluxDB, however, you must take into account that the data represented in the *Db Delay* column, has a delay due to the publishing of the data of the Kafka broker and Telegraf (note the column values of *Telegraf.push* and *Telegraf.interval*): if there is a flush of the data every *10s*, it is normal that there is a delay of *+10s* compared to the normal communication delay.

Another piece of information, but not less important, is the delay generated by HIVE to the REST API. This is the first delay that the architecture encounters and it is added to the one obtained later by the Kafka broker and the actual writing to InfluxDB. This value is reported in the *R Delay* column.

a) Test 1-2: Tests 1 and 2 are really low load: the tool generate 10 requests per second (for a total of about 100 connections handled in 10 seconds), to verify that all services are up and ready within the architecture.

b) Test 3-8: The tests then continued with number 3 to 5 increasing the number of connections per second up to 500, while, from test 6 to 8 the number of request, the number of concurrent connections and the size of the HTTP pipelining factor was increased. With the configuration of test 8, 12k total requests were reached in *10s*, but shows that an average delay (*Db Delay*) quite high.

c) Test 9-12: With test 9 through 23, is lowered the *flush* and *interval* values of Telegraf, increasing the number of requests/second to be satisfied by double. Specifically, with test 12, 83k total requests (7,5k/s on average) were reached. A great result knowing that the REST APIs are single threaded on a single core, and the architecture still all with a single replica and worker.

d) Test 13: In test 13 the REST APIs were deployed with 2 workers to handle the load of requests. It is possible see how the total went up to 120k. This value starts to give us some insight into how linearly the load can scale when scaling vertically on the architecture.

e) Test 14-16: In tests 14, 15, and 16 is tried to increase the number of requests to be suffused but it is noticed that the limit reached was around 130k total requests, making us realize that the REST APIs could handle at 100% CPU usage around 70k total requests per core (7k req/s).

f) Test 17: In test 17, was done a stress test with HIVE in dynamic mode, and is possible to notice that without having errors, varying the configurations of request generation, the 7,5k req/s was not exceeding, confirming the maximum load on average manageable by each core dedicated to the REST API.

g) Test 22-27: From test 22 to 24 REST APIs were deployed to more cores, but also increased Kafka partitions and Telegraf replicas. In tests 25, 26, and 27, it was scaled the REST APIs horizontally as well, resulting in about 482k satisfied requests (for an equal of about 48k requests per second). A very good value knowing that all the data arrived to the DB and there were no response errors from the REST API: it means that despite the load, they could be pushed a bit more before starting to fail to handle incoming requests.

h) Test 28: Test 28 was performed in dynamic HIVE mode and was the last test performed on the test day. As you can see from the table I, but also from the Figure4 you can see that with varying configurations the maximum load of 594k total requests (56k req/s) was obtained with a factor of 100 of concurrent requests and 50 as a factor of HTTP pipelining. Also, thanks to the delay Figure (5), is visible that the best handling is with 50 pipelining factor and 20 concurrent connections without having errors (6).

i) What about GET requests?: While is not identify a real-world scenario in which the REST APIs would be queried frequently to move the architecture on a critical point, just for information, a test to see how many GET requests the server could handle it has been carried out the same. Note that all the tests run so far were PUTs with a payload. A total of $1,2*10^6$ requests were reached in *10s*. In Figure 7 is reported the variance of the requests average and in Figure 8 the variance of the delay.

IX. Conclusion

It has been demonstrated how the RESTful server can handle 56k requests per second with 100 concurrent connections.

Is possible to see that the request load is variable with respect to the available hardware resources: in fact, the increase

Session	HIVE						REST API			Kafka	Telegraf			Results				
Number	wrk	Max Conn	Max Pip	Amount	Conn	Pip	Replicas	wrk	CPU [%] Usage	Partitions	Flush [s]	Interval [s]	Replicas	Tot Req	Avg Req/s	Db Delay [s]	K Delay [s]	R Delay [s]
1	1	-	-	10	1	1	1	1	10	1	10	5	1	100	10	0,81	-	0,08
2	1	-	-	10	1	1	1	1	10	1	10	5	1	100	10	0,67	0,92	0,08
3	1	-	-	500	1	1	1	1	30	1	10	5	1	5k	500	12,1	-	0,01
4	1	-	-	500	1	1	1	1	30	1	10	5	1	5k	500	14,1	0,10	0,07
5	1	-	-	500	2	2	1	1	40	1	10	5	1	5k	500	21,3	0,10	0,06
6	1	-	-	1500	1	1	1	1	40	1	10	5	1	15k	1263	46,3	0,86	0,07
7	1	-	-	1000	1	1	1	1	40	1	10	5	1	10k	1000	35,0	0,88	0,71
8	1	-	-	1200	2	1	1	1	40	1	10	5	1	12k	1194	46,2	0,78	3,34
9	1	-	-	1200	32	8	1	1	100	1	5	1	1	71k	7000	-	-	3,34
10	1	-	-	1200	32	8	1	1	100	1	5	1	1	71k	7000	10,0	0,86	3,30
11	1	-	-	1000	32	24	1	1	100	1	5	1	1	80k	7424	11,0	0,95	3,67
12	1	-	-	2000	32	24	1	1	100	1	5	1	1	83k	7511	10,0	0,96	3,67
13	1	-	-	2000	32	24	1	2	200	1	5	1	1	120k	12000	20,0	0,92	3,22
14	1	-	-	2000	32	24	1	2	200	1	1	1	1	125k	12000	5,0	2,99	3,20
15	1	-	-	1000	16	16	1	2	200	1	1	1	1	126k	12000	4,0	1,48	3,20
16	4	-	-	1000	16	16	1	2	200	1	1	1	1	135k	14000	4,2	-	3,31
17	1	100	50	-	-	-	1	2	200	1	1	1	1	160k	15000	-	1,50	3,52
18	4	-	-	2000	32	24	1	4	400	1	1	1	1	247k	23000	11,2	3,99	3,49
19	4	100	50	-	-	-	1	4	400	1	1	1	1	261k	30000	-	-	-
20	4	-	-	2000	32	24	1	4	400	1	1	1	3	270k	50000	40,8	4,83	3,59
21	4	-	-	1000	1	1	1	4	400	1	1	1	3	42k	4000	-	0,66	0,019
22	4	-	-	1000	1	1	1	4	400	3	1	1	3	42k	4000	0,55	0,69	0,015
23	4	-	-	2000	32	24	1	4	400	3	1	1	3	250k	25000	13,5	5,58	0,017
24	4	-	-	2000	16	8	1	4	400	3	1	1	3	223k	20000	0,76	0,35	3,30
25	4	-	-	2000	32	16	4	4	250	3	1	1	3	300k	30000	3,96	6,52	3,54
26	4	-	-	2048	64	16	4	4	350	3	1	1	3	417k	40000	6,23	10,92	3,26
27	4	-	-	4096	64	16	4	4	400	3	1	1	3	482k	48000	6,71	15,21	3,17
28	4	100	50	-	-	-	4	4	400	3	1	1	3	594k	56000	-	-	-

Table I

THE TABLE SHOWS THE CONFIGURATION OF THE K3S CLUSTER (REST API, KAFKA AND TELEGRAF COLUMNS), THE CONFIGURATION OF THE HIVE TOOL (HIVE COLUMN) AND THE RESULTS OBTAINED (RESULTS COLUMN).

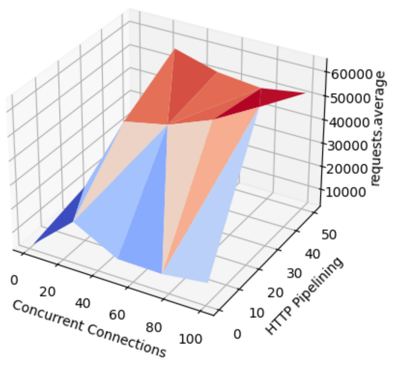

Figure 4. Graph showing how the number of PUT req/s varies with respect to the number of concurrent connections and the HTTP pipelining factor. The maximum peak is 56k req/s. Each test had a duration of 10 seconds.

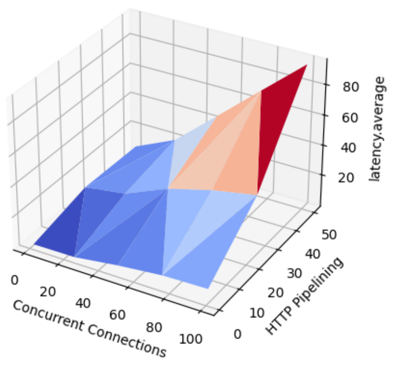

Figure 5. Graph showing how the number of ms of latency req/s varies with respect to the number of concurrent connections and the HTTP pipelining factor in the communication between HIVE and the RESTful server. Each test had a duration of 10 seconds.

of requests per second is linear with respect to the number of cores available to the RESTful server.

Thanks to this test, an alternative to the OPC-UA protocol is proposed to communicate with the data collection architecture of the ICE lab, but it is also demonstrated that the bottleneck is actually the RESTful server. The Kafka broker, Telegraf and InfluxDB, handled without difficulty the large load of requests, indicating the fact that the architecture could probably handle a much larger amount of data than we achieved in these tests.

X. FUTURE WORKS

Nodejs configuration with Fastify may not the best. According to the github [8] performance document, the RESTful server in java might be the one that can handle the largest number of requests. Future works could undoubtedly be to try a java-based architecture. Further analysis can be done on how Kafka and Telegraf handles data flushing, and how pods can be more efficiently balanced within the architecture. As far as the request generator (HIVE) is concerned, it is possible to find (or develop) new alternatives that generate even more requests.

XI. ACKNOWLEDGEMENTS

I would like to thanks Professor D. Carra for the opportunity given to me to create an application in the ICE lab, M. Panato and C. Tadiello for supporting me in the development and execution of the tests within the ICE Lab. All the work done is available and documented on the gitlab repository of ICE lab.

REFERENCES

[1] sindhuja cynixit. Kubernetes in 10 minutes: A complete guide. [Online]. Available: https://medium.com/faun/kubernetes-in-10-minutes-a-complete-guide-a9230124a02c
[2] kubernetes. kubernetes doc. [Online]. Available: https://kubernetes.io/it/docs/

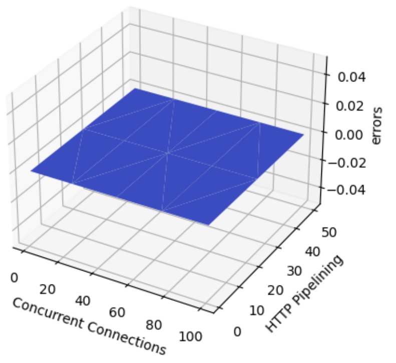

Figure 6. Graph showing how the number of errors varies with respect to the number of concurrent connections and the HTTP pipelining factor in the communication between HIVE and the RESTful server. Each test had a duration of 10 seconds.

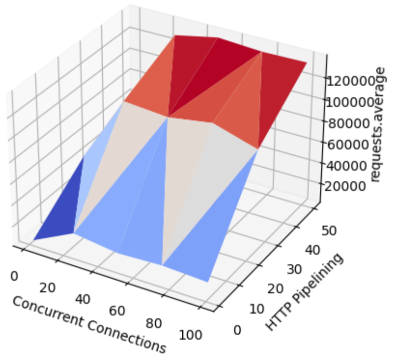

Figure 7. Graph showing how the number of GET req/s varies with respect to the number of concurrent connections and the HTTP pipelining factor. The maximum peak is 120k req/s. Each test had a duration of 10 seconds.

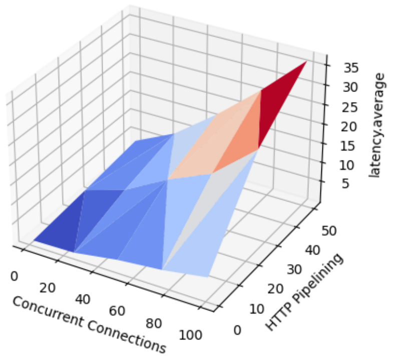

Figure 8. Graph showing how the number of ms of latency req/s varies with respect to the number of concurrent connections and the HTTP pipelining factor in the communication between HIVE and the RESTful server. Each test had a duration of 10 seconds.

[3] zell. Understanding and using rest apis. [Online]. Available: https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/

[4] R. Hat. What is a rest api? [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api/

[5] Kafka. Kafka doc. [Online]. Available: https://kafka.apache.org/intro

[6] Influx. Influx technical paper. [Online]. Available: http://get.influxdata.com/rs/972-GDU-533/images/InfluxDBClusterPerformance12072016.pdf

[7] M. M. Bragoi Vladislav, "Icetom."

[8] the benchmarker. web-frameworks. [Online]. Available: https://github.com/the-benchmarker/web-frameworks

[9] fastify. fastify. [Online]. Available: https://www.fastify.io

[10] mcollina. autocannon. [Online]. Available: https://github.com/mcollina/autocannon