Implementation of an email-based alert system for large-scale system resources

Robert Poenaru[1,2]
[1]*Horia Hulubei* National Institute of Nuclear Physics and Engineering, Magurele, Romania
[2]Doctoral School of Physics, University of Bucharest, Romania
robert.poenaru@nipne.ro

Re-do in Grammarly

***Abstract*—Tackling the current problems of interest for physicists that deal with various topics requires lots of computing simulations. Identifying and preventing any unusual behavior within the system resources that execute large-scale calculations is a crucial process when dealing with system administration since it can improve the run-time performance of the resources themselves and also help the physicists by obtaining the required results faster. In the present work, a simple *pythonic* implementation which 1) monitors a given computing architecture (i.e., its system resources such as CPU and Memory usage), and 2) alerts a custom team of administrators via e-mail in almost real-time when certain thresholds are passed, is presented. Using existing packages written in Python, with the current implementation it is possible to send e-mails to a predefined list of clients containing detailed information about any machine running outside the ”normal” parameters.**

***Index Terms*—python, system resources, alerting, email, smtp, monitoring, watchdog**

I. Introduction

The computing resources within a physics department must be up to speed and ready for a continuous run-time of small-, medium-, but also large-scale simulations, in order to assure a consistent and optimal workflow for the research teams that require calculations. Usually, there is a cohesive workflow between the scientists that want to run their simulations and the system administration (sysadmins) team that provides the necessary resources for executing them. The sysadmins must check that the resources which are performing calculations behave well, but they must also take care of the computing equipment that is sitting in *idle*-mode, in case new requests for allocating resources are issued by scientists. The process of resource management is crucial since it involves many factors in deciding which are the most optimal compute nodes that could start a new job, in terms of efficiency and speedup [1], these being deciding factors in the total run-time of the simulations themselves.

Proper job allocation, management, and execution will result in a minimal impact of resource slowdown, process blocking, or even dead-locks in the executing pipeline, giving the possibility of the researchers to obtain the desired numerical results as fast as possible.

On the other hand, any code optimization [2], [3] on the submitted simulations (done exclusively by the scientific teams) will also take advantage of the allocated resources, since *better code* implies faster execution, lower impact on the memory pool, and much lower probability of program interruption.

One can conclude that indeed, better resource management (done by the sysadmins) will result in better code execution, helping thus the scientists, but in the same way, any code optimization made by scientists will help the sysadmins, since the degree of failure within the executing simulations stack could be decreased. This reciprocal mode of improvements (for both *communities*) is sketched in Fig. 1, where the key characteristics of both *communities* (i.e., physicists issuing simulations and sysadmins dealing with computing management) are emphasized. The arrows signify the *reciprocal improvement cycle* between the two.

However, due to the large degree of complexity of the underlying computing infrastructure, issues related to memory bandwidth, network stability, CPU throttling [4], cache availability [5] and so on are highly probable, especially when the machines are running continuously. Frequent updates, unexpected network traffic, errors within the services running in the application layer [6] could also affect the idling nodes. While the former issues will only affect the simulations that are currently being executed, the latter set of issues could produce unexpected delays in the starting process of the job queue [7], which will increase the wait-time of simulation results. One can see that indeed, any issues which occur with the computing resources, even the idling ones, can affect the workflow of the research teams, causing the entire research department to finish their projects. As such, the sysadmins are essential by taking the proper actions on the computing infrastructure.

Unfortunately, there will always be moments when the sysadmin team that is monitoring a particular computing node (e.g., cluster, server racks, etc.) cannot keep track of the *health status* of the entire architecture at all times (although recently, some interesting models emerged within the literature that could improve the machine health status [8], [9] in an automated way). In order to properly, securely, and efficiently maintain a large-scale server infrastructure up and running (as the one that is present in a physics research department), there must be some kind of *alert system* that is constantly analyzing the system resources and tries to identify unusual behavior. When a potential malfunction is identified, the core feature of the alert system is to inform the sysadmins with regards to the occurring issue(s).

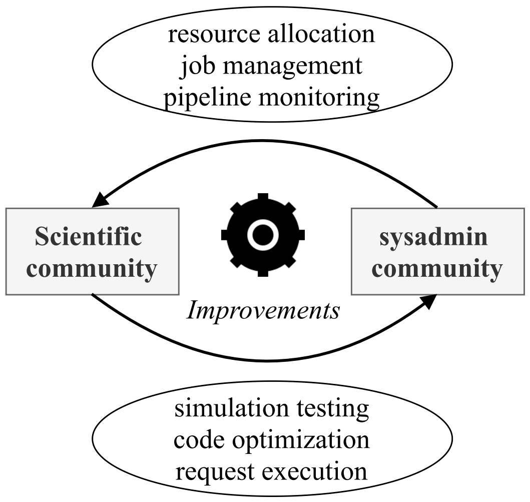


Fig. 1. A basic relationship between the scientific community that issues simulations to be executed on the computing resources, and the system administration team that deals with process allocation, execution, and management of resources. The continuous improvements between the two communities involve the steps indicated next to the arrows.

In the present work, such a model is implemented; namely, a service that analyzes the logs generated by the system resources of a computing cluster (i.e., CPU usage, RAM then usage, and so on) ~~and compares the~~ values with predefined thresholds decides if the monitored values indicate some and finally unusual behavior. If indeed, the thresholds are exceeded, the service will immediately raise an alert, sending information with the occurring issue to the sysadmin teams, which then can take action.

The paper is organized as follows: in Section I, a brief introduction of system administration is made, with motivation for implementing an alert system within large-scale computing infrastructure. Furthermore, in Section II the general overview on how the *pythonic* implementation of the current research is given actually works, mentioning its main features and advantages. Section III discusses the services that are used in order to keep track of changes within the log files generated by the system resources, while Section IV presents the method of deciding between *normal* and *unusual* behavior of said resources. Finally, an overview of the implementation which sends the actual alerts to the sysadmin team (via email) is made in Section V, followed up by some concluding remarks and ~~potential improvements of the project itself,~~ given in Section VI.

II. Alert-System Workflow

Implementation of the alert system is quite straightforward, following a procedure that does not require too much information. Fig. 2 shows how the process of taking action (i.e., fixing occurring issues on the computing nodes) by the administration team is taking place. Usually, the personnel that deal with system administration is located on-site, since the need-of-action would sometimes require physical access on the barebone servers. However, considering the current pandemic situation [10], the paradigm has shifted a lot to remote work, with the possibility of remote access by the sysadmins [11] to any computing cluster located in the departments they monitor. As a result, the alert system will eventually inform both groups, and depending on the type of issue encountered, actions will be made by one or the other (or even both).

The reason behind choosing *Python* [12] as the development tool here lies in its overall great compatibility with different operating systems, consistency between different system architectures (e.g., x86, ARM, [13]), well-documented resources, robust packages that allow the implementation of all the required instructions securely and reliably.

Between the monitoring phase and the alert + action phases, the current approach involves three major stages:

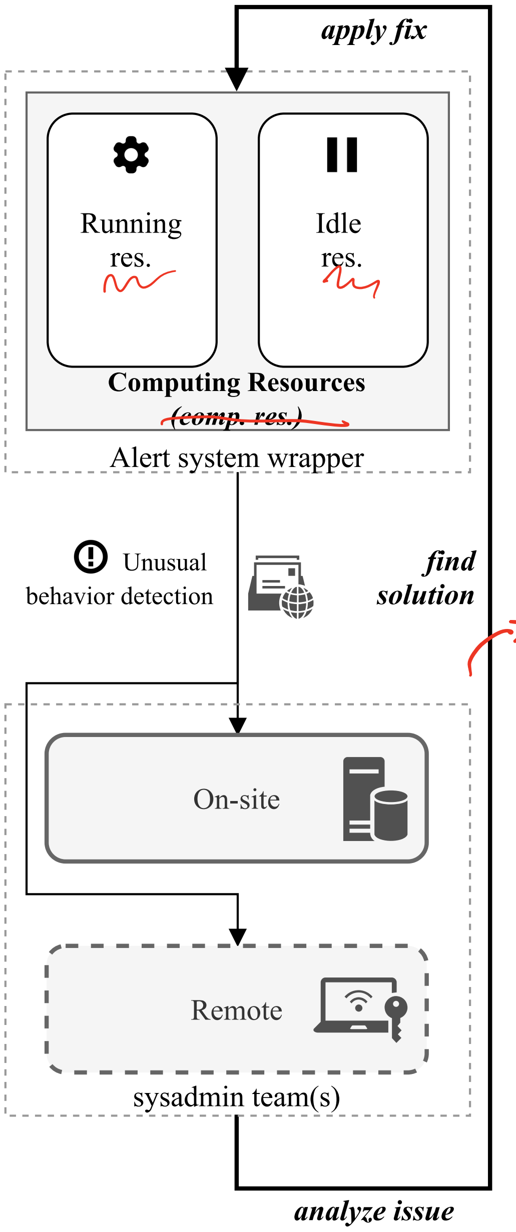
Step-0 - *Service configuration*
: This is done before the alert system is turned on, by setting some parameters to the desired values. The sysadmin team must decide on their values, depending on the required regime of execution. Discussion on the parameters will be made throughout the paper.

Stage-1 - *Data ingest*
: This is the first step, in which the implemented service is extracting the data coming from the computing resources. These data contain information with regards to the state of each resource, the status of all the running processes, and so on. They usually come in form of *log files*. Incoming data is stored at a fixed path, and the alert service will check for changes within those files.

Stage-2 - *Log analysis*
: With the incoming data from the monitored machines finally read, the service will analyze them in terms of their numerical values, i.e., for each system stat (CPU usage, RAM usage, incoming/outgoing network traffic). The stats of interest will be compared with the *thresholds*, which dictate whether the resources behave under normal or unusual conditions.

Stage-3 - *Alerts*
: If the monitored resources show that indeed their behavior is unusual (considering any of the controlled stats), the python implementation will trigger a so-called *alert*: a chain of procedures that follow the set of parameters defined in *Step0* which will inform sysadmins with regards to the encountered issue. The process of informing the sysadmin teams is done via email [14]. Its ease-of-use and ease-of-access allow the members to quickly and efficiently be informed, since the e-mail clients can run on pretty much any device (from a personal computer to a laptop, tablet, and smartphone), becoming a great solution in the current work.

Now that the core stages of the implementation were sketched (see also Fig. 3 for a graphical representation that encapsulates its features), it is instructive to describe each stage in particular, with its main goals and characteristics.



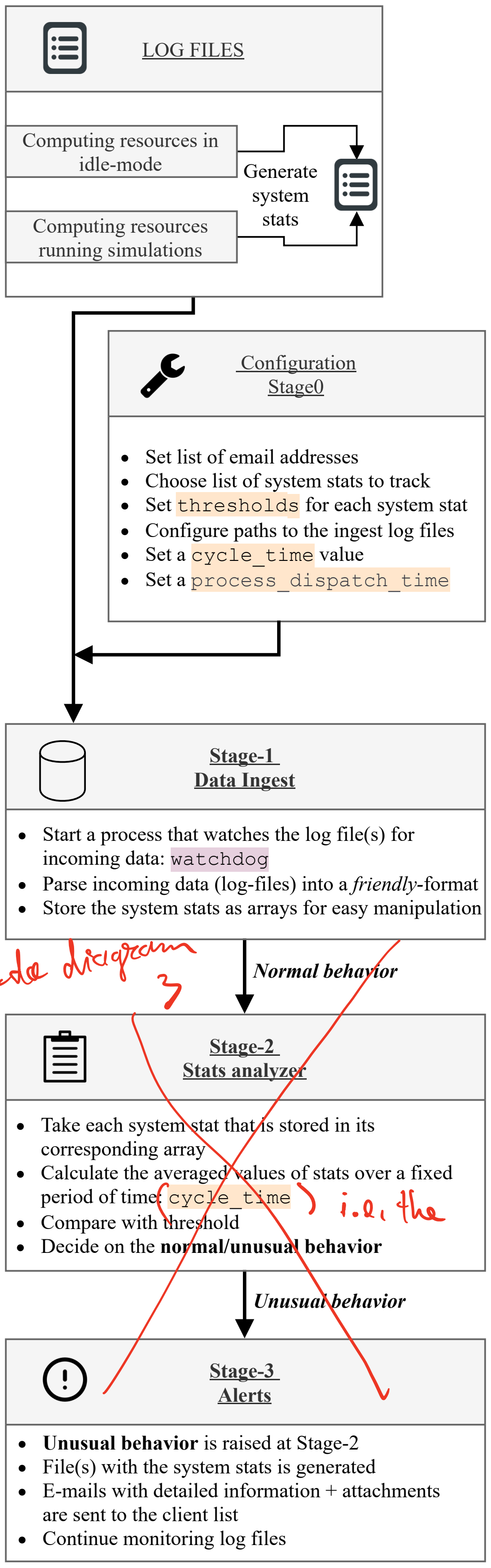
Redo diagram 2

Fig. 2. The general workflow of a system administration team, when an alert system would be deployed on the computing resources that need to be monitored.

III. Watchdog - Data Ingest

It was already mentioned that one can consider the underlying computing infrastructure as being composed of two types of resources: 1) the computing nodes that execute jobs (i.e., numerical simulations) and 2) the idling resources (i.e., nodes that wait for additional jobs picked-up by the resource management tool and allocated on the job pool by the sysadmins).

Both resource groups are required to send information related to their current status to the alert system. When dealing with large-scale systems, each node on the cluster must send



Configuration Stage0

- Set list of email addresses
- Choose list of system stats to track
- Set `thresholds` for each system stat
- Configure paths to the ingest log files
- Set a `cycle_time` value
- Set a `process_dispatch_time`

Stage-1 Data Ingest

- Start a process that watches the log file(s) for incoming data: `watchdog`
- Parse incoming data (log-files) into a *friendly*-format
- Store the system stats as arrays for easy manipulation

Redo diagram 3

Stage-2 Stats analyzer

- Take each system stat that is stored in its corresponding array
- Calculate the averaged values of stats over a fixed period of time (`cycle_time`) i.e. the
- Compare with threshold
- Decide on the **normal/unusual behavior**

Stage-3 Alerts

- **Unusual behavior** is raised at Stage-2
- File(s) with the system stats is generated
- E-mails with detailed information + attachments are sent to the client list
- Continue monitoring log files

Fig. 3. Complete overview of the 3-stage operation mode of the alert implementation.

its corresponding system stats, usually under the form of a log file. An easy way of assuring that all the nodes send their information to a centralized ”master node” is by using a so-called *log shipper*. One great example is the Filebeat [15] service, which collects system information and ships it as a log file via the network to any client. In fact, in a previous research [16], an implementation that can deliver logs via Filebeat was tested for multiple machines. The ”master node” consists of a central machine that collects all the logs coming from the entire computing infrastructure, with each log file usually updating very often. The alert service is also deployed on the master node (so far it was tested as a python script that is running in the background, as a process).

With all the logs from each computing node saved in a centralized location, the *python watcher* - part of Stage 1 - can monitor those files continuously. In the current testing procedures, new log data arrived (on average) each 2-3 seconds. Since the log files are constantly changed and overwritten by newly ingested data, it is required that the files must be read again and again by the python watcher service. This is done via the `watchdog` package [17]. With the help of this python module, one can take a list of log files, create a set of *file system handlers* wrapped into an `Observer` class, and then the implementation starts to ”watch” every file within the list, keeping track of every new *event* that arrives within any of the files. An event is represented by a trigger of the type `file_modified`, meaning that once a file has some new content in it, its state is modified, and as such, the file event handler returns the newly modified state. By default, the `Observer` class comes with some predefined methods that must be used within the alert system. The method `start()` is called before all other steps within the multi-stage pipeline. The methods `stop()` and `join()` are used once no other log entries are arriving in the corresponding paths (signaling that there could be issues with the connection, or the Filebeat service ~~failed within the computing cluster)~~ Stopped running. Listing 1 shows a basic example of a method that checks for new events in a file, as long as no keyboard keys are pressed.

Listing 1. A straightforward example of using the watchdog module to track a file for any changes.

```
event_handler = LoggingEventHandler()
    observer = Observer()
    observer.schedule(event_handler, path,
                          recursive=True)
    observer.start()
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        observer.stop()
    observer.join()
```

In the current implementation, the watchdog module is monitoring a set of log files in which content is written constantly, then with the help of the so-called `Reader` class, the incoming log data is parsed and stored in memory as arrays (the arrays are cleared every once in a while to avoid any memory overhead).

IV. Thresholds - Normal vs. Unusual Behavior

With the arrays generated in Stage-1, it is possible to go into Stage-2, where some numerical analysis is performed on the collected data. This is a crucial step since it involves deciding if any of the incoming stats for a particular computing node is ~~expecting~~ behaving normally or not.

The decision between normal vs. expected unusual behavior on a system stat is taken by comparing its averaged value over a fixed period of time. The fixed period is configured within Stage-0 (i.e., the `cycle_time`) by the user (sysadmin). Then, within a `cycle_time`, every new log data that arrives at the master node is collected by the watcher (via the `Observer()` class and its file event handler), added in its and added corresponding array (denoted by ”stack” within the code-base). After a `cycle_time` has passed, the filled arrays (stacks) are analyzed. As an example, for the CPU usage and RAM usage, the corresponding functions that perform numerical analysis are defined below (see Listing 2). Due to the space constraints of the current paper, further examples will only focus on the CPU usage and the RAM (memory) usage within the system resources.

Listing 2. The methods for calculating averaged stats values used within the project code-base.

```
@classmethod
    def Analyze_CPU_Usage_Stack(cls,
                cpu_usage_stack, cpu_threshold):
  @classmethod
    def Analyze_MEM_Usage_Stack(cls,
                mem_usage_stack, mem_threshold):
```

With the obtained average values for each stat (e.g, CPU usage, RAM usage), the next step is to compare them with the `thresholds`. Within the code-base, they are declared as a dictionary (see Listing 3).

Listing 3. Declaration of some thresholds within the code-base.

```
thresholds = {"cpu": 75,
              "mem": 75}
```

In the example shown in Listing 3, both the CPU and the RAM usage upper limit were set to 75, meaning that if within a `cycle_time`, any monitored node that exceeds these limits (e.g., the CPU usage or RAM usage - or both - are higher than 75%) will raise the *unusual behavior*. Obviously, if the conditions for exceeding the upper limit are not met, the computer nodes are considered to behave as expected (normal behavior).

As Fig. 3 shows, this is the crucial step in deciding whether the alert via e-mail stage should be called or not. Under normal behavior of the monitored nodes, after each `cycle_time` the arrays are cleared, and the watcher module continues to read new log entries.

V. Alert via Email

The last step in the implementation is the actual alert itself, which is informing the sysadmins with regards to some detected unusual behavior, and this is done by e-mail to a list of clients (each client represents a member of the sysadmin team and it requires a valid e-mail address). As already mentioned at the beginning of Section II, the choice of an e-mail based alert system was simply because of reliability and convenience (no extra apps or web services required).

In terms of required packages, the following python modules are used [18]:

- `smtplib`
- `ssl`
- `email.mime.base.MIMEBase`
- `email.mime.text.MIMEText`
- `email.mime.text.MIMEMultipart`

The built-in Python module `smtplib` [19] allows one to send e-mails using the well-known SMTP protocol (Simple Mail Transfer Protocol). The module uses the standard protocol that was developed a long time ago ~~by~~ [14].

It is worth mentioning certain characteristics of the current approach. The e-mails are sent through a Gmail account that was created for *Development*, turning on the feature *Less secure app access* [20]. This way, the implemented algorithm can send e-mails securely to a list of clients.

In the code-base shown in Listing 4, the address that is used for sending alerts is configured, together with the list of clients that should receive the alert and a subject.

Listing 4. Setting up the e-mail headers, such as the destination and subject.
```
 message = MIMEMultipart()
         message["From"] = ROOT_EMAIL

         message["To"] = email_address
         message["Subject"] =
              f'{str(datetime.utcnow())[:19]}
␣␣␣␣␣␣␣␣␣␣␣␣−␣Alert␣via␣DFCTI␣Monitoring␣System'
```

With the subject properly configured, the next step is to create the attachments that will be added to the final message. The attachments consist of graphical representations (done with the `matplotlib` package [21]) with the system resources which raised unusual behavior (in Listing 5 it is denoted by part2). For example, if one of the nodes has an unusually large CPU usage, then a plot with the CPU usage within the last `cycle_time` time-interval is plotted and compared with the threshold. Another attachment is a file that contains all the numerical data with the recorded logs (within the ~~code-base~~ shown in Listing 5 it is the ”dat” file denoted by the variable `part1`).

\texttt{ } code snippet

Listing 5. Create attachments and save them securely by encoding them and embed them as string objects within the final object.
```
# Adding the body to the actual email
message.attach(MIMEText(message_body, "plain"))

# Open the dat file in binary mode
with open(attachment_files[0], "rb") as attachment:
     # Add file as application/octet−stream
     part1 = MIMEBase("application", "octet−stream")
     part1.set_payload(attachment.read())

     # Open the plot file in binary mode
with open(attachment_files[1], "rb") as attachment:
     # Add file as application/octet−stream
     part2 = MIMEBase("application", "octet−stream")
     part2.set_payload(attachment.read())

# Encode file in ASCII characters to send by email
encoders.encode_base64(part1)
encoders.encode_base64(part2)

# Add header as key/value pair to attachment part
part1.add_header(
     "Content−Disposition",
     f"attachment;␣filename=␣{attachment_files[0]}",)

# Add header as key/value pair to attachment part
part2.add_header(
     "Content−Disposition",
     f"attachment;␣filename=␣{attachment_files[1]}",)

# Add attachment to message +
# convert message to string
message.attach(part1)
message.attach(part2)

final_alert = message.as_string()
```

With the alert message properly constructed, the last step is to send it securely to the client list. This is done by opening an `ssl` connection. The code example sketched in Listing 6 creates a secure connection with Gmail's SMTP server, using the `SMTP_SSL()`1 of `smtplib`2 to initiate a TLS-encrypted connection [18]. The default context of `ssl` validates the hostname and its certificates and optimizes the security of the connection.

1 function
2 class

Listing 6. Procedure for sending the e-mail securely, through *ssl*. The dotted code-base marks some debug steps that are irrelevant to the current discussion.
```
CONTEXT = ssl.create_default_context()
with smtplib.SMTP_SSL("smtp.gmail.com",
              PORT, context=CONTEXT) as mail_server:
     time_stamp = str(datetime.utcnow())[0:19]
     # log−in stage
     try:
          mail_server.login(ROOT_EMAIL, UNICORN_ID)
     except Exception as exc:
          ...
     else:
          ...
     # sending stage
     try:
          mail_server.sendmail(ROOT_EMAIL,
                         email_address, final_alert)
     except Exception as exc:
          ...
     else:
          ...
```

With the code-base now complete (Listings 4-5-6), in case of any unexpected behavior detected within the monitoring process, alerts will be constantly sent to the sysadmin team. Fig. 4 shows the plot with the CPU usage for a node that has unexpected behavior.

VI. Conclusions & Future Work

In this work, a description of the development process for an alert system was made. The importance of a system administration team that assures efficient resource management was illustrated in the beginning. The alert system consisted of several stages of procedures that were incorporated in a general workflow which performed the constant monitoring of a set of log files, then perform numerical analysis on the parsed log data. Afterward, a comparison of each system stat with the configured thresholds is made, followed by the

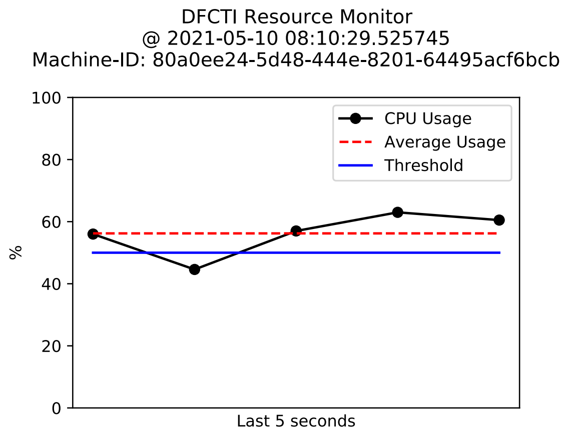


Fig. 4. The plot file within the attachments for an alert which informs the sysadmins on high CPU usage of a particular node (the unique `machine-id` corresponds to that node).

decision on the normal/unusual behavior of each compute resource within the infrastructure, and then finally, create an automated e-mail message containing i) detailed text with the detected "anomaly" and ii) attachment files with graphical representations and tabulated data.

exclusively

The current code-base used only open-source modules developed excursively in Python, assuring in this way a great degree of compatibility with most system architectures, operating systems, and environment type (i.e., bare-bone, virtualized). Moreover, the scalability of Python allows the current implementation to be deployed on large-scale clusters even through containerized approaches (i.e., Docker [22], K8s [23]). In fact, this is a major next step in the development process of the ~~project~~. service

It should be highlighted that the implementations which read and parse log files, perform numerical analysis on the parsed stats, and also decide on the behavior of the computing nodes, are developed *from scratch* for this project. The built-in functions from `watchdog`, `email`, and `smtplib` modules were properly used to create customized classes (also from scratch) which ultimately send e-mails securely. The resulting code-base is to be considered an original contribution within the current work, followed by the impact of the alerting service itself within the maintenance process of the sysadmin team.

Acknowledgment

This work was done within the Department of Computational Physics and Information Technology at *Horia Hulubei* National Institute of Physics and Nuclear Engineering. The development was funded by the European Regional Development Fund through project CECBID-EOSC (POC/397/1/1-124405).

References

[1] Ashkan Paya. Resource management in large-scale systems. *Department Of Computing Science Umea University*, 2015.
[2] PVS Studio. Code optimization, 2017. Last accessed 24 September 2021, `https://pvs-studio.com/en/blog/terms/0084/`.
[3] Krishna Sai. Code optimization, 2018. Last accessed 24 September 2021, `https://www.techtud.com/`.
[4] IBM Corporation. Cpu throttling, 2012. Last accessed 24 September 2021, `https://www.ibm.com/docs/en`.
[5] Caching challenges and strategies. Last accessed 24 September 2021, `https://aws.amazon.com/builders-library/`.
[6] Pstree - linux man page. Last accessed 24 September 2021, `https://linux.die.net/man/1/pstree`.
[7] IBM Corporation. Job queues, 2015. Last accessed 24 September 2021, `https://www.ibm.com/docs/en`.
[8] Ramin Hasani, Guodong Wang, and Radu Grosu. A machine learning suite for machine components' health-monitoring. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 2019.
[9] Chia-Yu Lin, Tzu-Ting Chen, Li-Chun Wang, and Hong-Han Shuai. Health-based fault generative adversarial network for fault diagnosis in machine tools. *The 4th International Workshop on Artificial Intelligence of Things*, 2020.
[10] Marco Ciotti, Massimo Ciccozzi, Alessandro Terrinoni, Wen-Can Jiang, Cheng-Bin Wang, and Sergio Bernardini. The covid-19 pandemic. *Critical reviews in clinical laboratory sciences*, 57(6):365–388, 2020.
[11] Amira B Sallow, Hivi Ismat Dino, Zainab Salih Ageed, Mayyadah R Mahmood, and Maiwan B Abdulrazaq. Client/server remote control administration system: Design and implementation. *Int. J. Multidiscip. Res. Publ*, 3(2):7, 2020.
[12] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.
[13] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2013.
[14] Jonathan Postel. Rfc0821: Simple mail transfer protocol, 1982.
[15] Filebeat quick start. Filebeat overview. Last accessed 24 September 2021, `https://www.elastic.co/guide/en/beats/`.
[16] Robert Poenaru and Dragos Ciobanu-Zabet. Elk stack – improving the computing clusters at dfcti through log analysis. In *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–8, 2020.
[17] Watchdog. Last accessed 24 September 2021, `https://www.elastic.co/guide/en/beats/`.
[18] Joska de Langen. Sending emails with python. Last accessed 24 September 2021, `https://realpython.com/python-send-email`.
[19] smtplib — smtp protocol client. Last accessed 24 September 2021, `https://docs.python.org/3/library/smtplib`.
[20] Less secure app access. Last accessed 24 September 2021, `https://myaccount.google.com/lesssecureapps`.
[21] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.
[22] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.
[23] Eric A Brewer. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing*, pages 167–167, 2015.