JOINT INSTITUTE FOR NUCLEAR RESEARCH
Meshcheryakov Laboratory of Information Technologies

FINAL REPORT ON THE START PROGRAMME

Development of the server part of the system for monitoring the state of computing nodes of the “HybriLIT” Heterogeneous Platform based on asyncronous data transfer technologies and the use of web sockets

Supervisors: *Dmitry Belyakov, Maxim Zuev*

Student:
Far Eastern FU *Maxim Skazkin*

Participation period: *03 March - 13 April*

Dubna, 2024

Abstract

The “HybriLIT” Heterogeneous Platform computing platform, used at the JINR, is designed to solve various complex computational tasks. These include collecting and processing data from experiments as well as simulation and modeling physical processes. ”Govorun”, the main component of HybriLIT, consists of more than one hundred two-processor servers that continuously run user programs and communicate with each other over a low latency network fabric. Therefore, it is necessary to monitor the state of these computing nodes, including their CPU loads, memory and storage usage, and network traffic. The purpose of this project was to develop software that would provide a simple way to monitor all these aspects.

CONTENTS

1 INTRODUCTION

HybriLIT is a high-performance computing platform that comprises more than 100 physical servers. When all the various virtual machines and services running on servers that support the platform are taken into account, the total number of devices in the system is approximately double that number. These devices have different technical specifications and are designed for different purposes. The main component of HybriLIT is the ”Govorun” supercomputer. This supercomputer is used to process data from experiments and solve various scientific problems such as physical and mathematical modeling, simulation, machine learning models training and numerous other types of research that require significant computing power.

It is essential for HybriLIT administrators to be able to monitor resource usage in real-time in order to effectively manage any anomalies or improper use of computing nodes that may arise. While existing tools provide some level of monitoring, they are not sufficient for the specific use cases there are in mind. Open source solutions like Telegraf [1] require development a custom plugin for specific monitoring goals and the existing tools [2] are not maintained. Therefore, it is proposed to create a custom monitoring system tailored to the existing requirements. These requirements are extensively described in the chapter on requirement development 2.

1.1 Problem statement

The goal of this work was to design and develop backend side of the monitoring system, that consists of two parts: software for sensors and for a server. The first one is intended to run on each computing node and collect data such as CPU load, memory usage, storage usage, and network traffic at regular intervals. The second one is intended to collect information from sensors and send it to clients. The client application is an essential part of the monitoring system, and it is not the least important component. It is described in Gennady Karpov’s [3] work dedicated to the development of tools for visualizing usage data from HybriLIT computing nodes.

The main steps to achieving the stated goal include requirements development for the operation of the sensors and server, defining the protocol used to communicate between all parts of the system, and implementing a minimum viable product to create a working system that allows for further modifications.

2 REQUIREMENTS DEVELOPMENT

The desired system for monitoring the state of computing nodes should consist of three main components: software for sensors, a backend server, and a client application. The scheme is depicted on Figure 1. The software for sensors is a program that runs on each computing node and collects data such as CPU load, memory usage, storage usage, and network traffic at regular intervals. This information is then transmitted to the backend server for further processing. The backend server is responsible for aggregating and storing the data received from the sensors. It should be located on the same local network as the computing nodes to ensure fast and reliable communication. When a server establishes a connection with a particular computing node, it collects information about that node and waits for sensor data. Once the data is received, the server aggregates it and sends it to the client application.

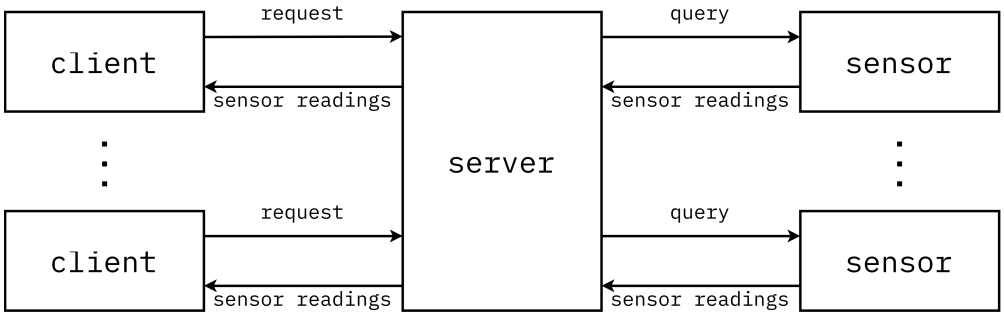

Figure 1 — System components

Next, the requirements for sensors and server operation will be discussed in detail.

2.1 Sensor software requirements

Since the sensor software runs on the computing nodes, it is the only component that has access to the information that is important. This information is:

- General technical specifications
- Current hardware load

The sensor should collect information about general technical specifications at the start of its operation and then send this information to the server when it is connected. After this, the sensor can receive requests to send data about the current system load at regular intervals, until it receives a signal to stop sending. The sending can then continue with the next request.

Sensors should be given with a name and associated with a group to make it easier to navigate among them and distinguish between them.

The sensor software should not put a heavy load on the system it is running on. It is also important not to use blocking calls to wait for the next time data needs to be sent or for a new

request to be received. Once the sensor is ready, it should attempt to connect to the server socket on the specified port. If the connection is lost due to network issues or other problems, the sensor should try to reconnect.

2.2 Backend server requirements

The backend server acts as a link in the communication between sensors and clients. First, the server should constantly listen for connections from client applications. Once a connection is established, the server must identify and authenticate the clients. Then, clients can send requests for information about the computing nodes they are interested in. Because different clients may not send the same requests, or may request different amount of information, even about the same computing node, the server needs to aggregate these requests to form a single, optimal query for each computing node. When the server receives a response from a sensor, it should define all the clients who requested that information and send them the response.

Second, the server waits for connections from sensors on the specified port. When a sensor connects, the server receives information about it, including its group, name, and technical specifications. When a request is received about a specific sensor, the server should update the query for that sensor by sending a request. The server then waits for responses from the sensors and extracts the information requested by clients. It sends the requested information to the clients. When no one is interested in a particular computing node, the server should suspend its sensor activity.

The server does not need to store the history of client requests or sensor responses. Storing information can be further achieved by adding a database as an additional client application.

The heavy-load optimization requirements apply to the server the same way as to sensors: it should not use blocking calls to wait for the next time that data needs to be sent or for a new request to be received.

3 PROTOCOL DESIGN

The described process is data streaming [4]. Streaming data has several unique characteristics: it is unique, non-uniform, imperfect, and continuously flowing. This last characteristic means that the overall system must have low latency, measured in seconds.

The non-uniformity and imperfection of the data mean that the processing service, in this case the backend server, must be able to process it correctly. To achieve this, we can use an event-driven architecture [5]. In this architecture, events are the triggers that keep the system running. If there are no events due to some failure, the overall system will not fail, but will simply remain idle.

The only events in the monitoring system are sensor readings and client requests. These events need to flow through the system in a way that follows a predefined protocol, which allows different components to communicate with each other.

All communication is done using messages in JSON format, which maps fields to their corresponding values in JavaScript-like way.

3.1 Communication between sensors and the server

3.1.1 Messages from the server to sensor

The only kind of message the server send to the sensors is the queries on what clients want to obtain from them. The server generates a query to each sensor, specifying the names of the fields that need to be filled out. These field names are predefined and can't be changed while the system is running. The server can also specify, for each category of CPU, network, memory, and disks, whether it wants extended sensor readings, i.e., per each core, network interface card, or disk. In the query header, the server provides information about it, including the mark, the time interval for taking readings, and the preferred unit of measurement. Using the query mark server can further distinguish between different responses from the one sensor if they come with some latency. The structure of JSON query looks like this:

- `"mark"` - a mark for each query (or just its name) that the sensor sends in its response. This way the server will know that this is the current response and not the previous one, which may have come with delay due to latency,
- `"interval"` - a time interval in seconds that the sensor will delay the next response,
- `"measure"` - a unit of measurement in which the sensor should present data. Only the values `"b"`, `"kb"`, `"mb"` are supported,
- `"cpu_fields"` - the names of the fields related to CPU that need to be filled out. It is the list of the following possible values:
  - `"freq"` - current frequency,

  - `"user"` - percentage of time spent by normal processes executing in user mode since last call,
  - `"system"` - percentage of time spent by processes executing in kernel mode since last call,
  - `"nice"` - percentage of time spent by niced processes executing in user mode since last call,
  - `"iowait"` - percentage of time spent waiting for I/O for complete since last call,
  - `"irq"` - percentage of time spent for servicing hardware interruptions since last call,
  - `"softirq"` - percentage of time spent for servicing software interruptions since last call,
  - `"steal"` - percentage of time spent by other operating systems running in a virtualized environment since last call,
  - `"guest"` - percentage of time spent running a virtual CPU for guest operating systems since last call,
  - `"guest_nice"` - percentage of time spent running a niced guest since last call,
  - `"idle"` - percentage of time spent being idle since last call,

- `"cpu_extended"` - boolean means whether to send CPU usage data per each core,
- `"net_fields"` - the names of the fields related to network interfaces that need to be filled out. It is the list of the following possible values:

  - `"recv"` - amount of data received,
  - `"sent"` - amount of data sent,
  - `"dropin"` - number of dropped packets while receiving,
  - `"dropout"` - number of dropped packets while sending,

- `"net_extended"` - boolean means whether to send network usage data per each network interface,
- `"mem_fields"` - the names of the fields related to memory that need to be filled out. It is the list of the following possible values:

  - `"used"` - amount of used RAM,
  - `"shared"` - amount of shared memory,
  - `"cached"` - amount of cached memory,
  - `"swap"` - amount of used swap memory,

- `"mem_extended"` - boolean means whether to send memory usage data counted as percentage or as number of bytes (or kilobytes or megabytes according to the `"measure"`),
- `"dsk_fields"` - the names of the fields related to disks that need to be filled out. It is the list of the following possible values:

  - `"used"` - amount of used storage,
  - `"read"` - number of reads,
  - `"write"` - number of writes.

- `"dsk_extended"` - boolean means whether to send disks usage data per each disk.

Thus, the sensor is not dependent on who makes requests to it or why. Its functions are abstracted from the rest of the system.

3.1.2 Messages from sensor to the server

Each sensor sends two types of messages to the server: one that contains information about the computing node's technical specifications with header `spec` and the other that contains sensor readings with header `resp`. These two types of messages are distinguished by the type of information they contain, which is specified in the header of each message. The headers of both messages also contain information about the group and name of the sensor, which helps the server distinguish between different sensors.

The message about the technical specifications is only sent when the sensor first connects to the server, while the message with sensor readings is sent continuously. The structure of JSON is as follows:

- `"header"` - string `"spec!group!name"` where group and name are actual for computing node,
- `"cpu"`:
  - `"cores_phys"` - number of physical cores,
  - `"cores_logic"` - number of logical cores,
  - `"min_freq"` - list of minimal frequencies per each physical core,
  - `"max_freq"` - list of maximal frequencies per each physical core,
- `"net"`:
  - `"nics"` - list of network interface card names,
- `"mem"`:
  - `"mem_total"` - total amount of RAM,
  - `"swp_total"` - total amount of swap memory,
- `"dsk"` - list of devices described as follows:
  - `"name"` - name of device,
  - `"mountpoint"` - mountpoint of device,
  - `"total"` - total volume of device.

The sensor readings follow the server queries as soon as they are received and processed by the sensor. The structure of JSON is as follows:

- `"header"` - string `"resp!group!name!mark!timestamp"` where group and name are actual for computing node and mark is actual for the query the sensor answers,
- `"cpu"` - if `"cpu_extended"` was false then this is list of fields given in `"cpu_fields"` with its values. If `"cpu_extended"` was true then these fields with values is listed for each core,

- `"net"` - if `"net_extended"` was false then this is list of fields given in `"net_fields"` with its values. If `"net_extended"` was true then these fields with values is listed for each network interface,
- `"mem"` - it is list of fields given in `"mem_fields"` with its values,
- `"dsk"` - if `"dsk_extended"` was false then this is list of fields given in `"dsk_fields"` with its values. If `"dsk_extended"` was true then these fields with values is listed for each disk.

3.2 Communication between clients and the server

The main purpose of the server is to provide each client with the information they need about a particular computing node or group of nodes. Therefore, if a client does not know what sensors exist, the server should first provide him with a list of sensor groups. This can be done via a request `lsob` from a client. The server receives that message and responses with the JSON having following structure:

- `"header"` - string `"lsob"`,
- `"groups"` - list of groups available that the server is aware of.

After receiving a list of groups a client may then want to know which fields are present in the sensors readings in the specified group collected by the server so far. So a client can send request `head!group` and the server will response with the JSON having following structure:

- `"header"` - string `"head!group"`,
- `"cpu"` - list of all fields received from sensors within the specified group in `"cpu"` block of their responses and the units of measurement for each field as a string,
- `"net"` - list of all fields received from sensors within the specified group in `"net"` block of their responses and the units of measurement for each field as a string,
- `"mem"` - list of all fields received from sensors within the specified group in `"mem"` block of their responses and the units of measurement for each field as a string,
- `"dsk"` - list of all fields received from sensors within the specified group in `"dsk"` block of their responses and the units of measurement for each field as a string.

Sending request `desc!group!name` will provide a client with the description of fields the server collected from single sensor in the same form.

This information can be used for rendering a web application, creating a database schema, or other purposes.

And last but not least, a client can request the latest sensor readings interested to it. This can be done by sending request `mstd!group` for all sensors in the specified group, or `mext!group!name` for a single sensor with the specified name and group. After receiving the request, the server will start sending a client the responses over a certain time interval, but as soon as readings are received from the sensors. So it fits the publisher-subscriber model. More on it is presented in the next chapter 4.

The structure of JSON response on these requests is as follows:

- `"header"` - string `"mstd!group!name!timestamp"` or `"mext!group!name!timestamp"`,
- `"cpu"` - fields that listed in `"cpu_fields"` and their values averaged for all cores in case of the request `mstd!group` and listed for each core independently in case of the request `mext!group!name`,
- `"net"` - fields that listed in `"net_fields"` and their values summed up for all network interfaces in case of the request `mstd!group` and listed for each network interface independently in case of the request `mext!group!name`,
- `"mem"` - fields that listed in `"mem_fields"` and their values,
- `"dsk"` - fields that listed in `"dsk_fields"` and their values summed up for all disks in case of the request `mstd!group` and listed for each disk independently in case of the request `mext!group!name`.

The client can suspend this subscription by sending request `stop` and no more sensor readings will be sent.

4 IMPLEMENTATION DETAILS

4.1 Used technologies

When it comes to selecting the appropriate technology, Python ($\geq$ v. 3.10) is a great choice due to its ease of maintenance and portability. FastAPI, a web framework, is used because of its high performance, flexibility, asynchronous capabilities, and support for WebSockets. Another valuable tool in the Python ecosystem is the psutil library. This library provides access to information about running processes, system utilization, and various system resources such as CPU, memory, network, and disks. It is an essential component of sensor software running on computing nodes.

4.2 Asyncrony

As stated in the chapter on requirements 2, the system relies heavily on input-output operations. To ensure optimal performance and minimize CPU usage, it is essential to avoid using blocking calls to wait for I/O operations to complete. So it needs to run I/O operations concurrently. One way to address this issue is by using multiple threads, however, this approach can lead to performance degradation due to Python's Global Interpreter Lock and frequent context switching [6]. It can also be a source of errors, as the context switching is managed by the operating system and cannot be fully controlled. To avoid data corruption, it is necessary to use locks and other synchronization mechanisms. Another approach is to use the systems `select` calls [7] or even Python built-in `selectors` module [8], which allows for efficient multiplexing of I/O by selecting from a list of sockets that are ready for reading or writing. This approach avoids blocking and helps optimize performance. However, due to the nature of WebSockets in the FastAPI framework [9], this option is not available. In such cases, asynchrony becomes a useful tool. It is inexpensive, it does not require the use of the GIL, and does not necessitate the use of multiple interpreter instances or message passing channels. Additionally, context switching can be performed predictably and controlled from the program. Python provides asynchrony through the built-in `asyncio` module [10].

4.3 System design

The server uses two main event loops. The first one is the FastAPI event loop, which handles requests received over HTTPS or through WebSockets. The second loop is used for communicating with sensors and is started with `asyncio.start_server` coroutine [11].

Communication between the server and clients is based on the Publisher-Subscriber behavioral design pattern. Each client subscribes to updates from sensor readings that it is interested in. The server is not aware of each individual subscriber, but it provides an interface through which a

client can become a subscriber [12]. This interface uses requests with the `mext` and `mstd` headers to request subscription for either a whole group of sensors or a specific sensor. Clients can unsubscribe by sending a response with a `stop` header. When the server receives updates from sensors, it iterates through the list of subscribers and sends them the received updates.

The server assumes that monitoring for a whole group of sensors requires fewer sensor readings than monitoring a single sensor. Therefore, the server has predefined query templates for these two scenarios. The template for the entire group uses fewer fields and does not request extended readings, such as per each CPU core, network interface, or disk. In contrast, the query template for a single sensor requests more extended information.

It is worth noting that clients can choose different levels of monitoring for a single computing node. In this case, the server does not send both queries for that sensor, but only requests extended readings. Upon receiving them, it checks whether any clients have subscribed for monitoring of the group in which that sensor is located. If so, the server reduces the amount of information sent to those clients. The server removes unnecessary data from the readings, averages it across the CPU cores, and sums it up through the network interfaces and disks.

Communication between two beforementioned event loops is not done directly. Instead, it follows the Mediator behavioral design pattern [12]. A set of objects store the global system state and help to communicate between clients and sensors. These objects, which are used to manage clients, sensors, queries, and responses, are called `ClientsRepo`, `SensorsRepo`, `QueriesRepo`, and `ResponsesRepo`, respectively. The overall system workflow is as follows:

1. Sensors connect to the server and corresponding socket streams `StreamReader` and `StreamWriter` [11] and computing nodes specifications is stored in `SensorsRepo`.
2. Client connects to the server and subscribes for monitoring for some group of sensors or one single sensor. His `WebSocket` and subscription is stored in `ClientsRepo`. The query is added to `QueriesRepo` if it is not presented yet.
3. Adding query to `QueriesRepo` triggers sending it to the corresponding sensors.
4. Receiving sensor readings triggers adding it to `ResponsesRepo`.
5. Adding new sensor readings triggers looking for subscribers in `ClientsRepo` and sending them the requested data.

The bunch of the last triggers is the key feature of event-driven architecture. Only events like receiving requests or data can trigger the system for further processing. This is depicted on the Figure 2:

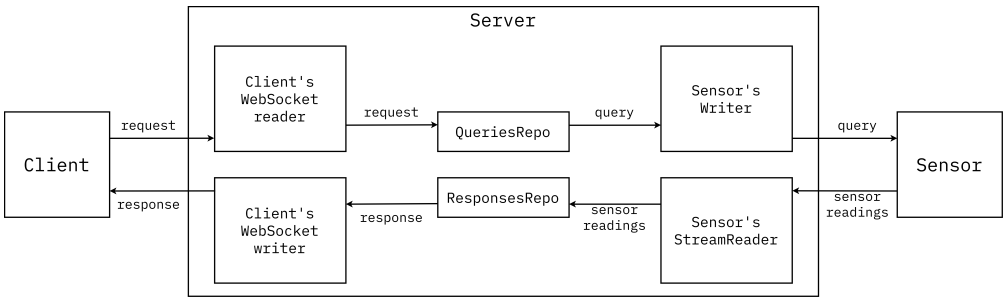

Figure 2 — Event triggers

Sensor software also makes use of asynchronous features. The receiving, processing, and sending of sensor readings are all done concurrently as coroutines. When a new query is received, the global `Query` object is updated, following the Singleton design pattern [12]. Since it is used every time the sensor reads the system's resource usage, this object also serves as a Mediator [12]. Additionally, sensors will attempt to reconnect to the server if the connection has been lost.

5 CONCLUSION

As a result of this work, the server component of the application has been implemented to monitor the status and workload of the computing nodes in the “HybriLIT” Heterogeneous Platform. The flexible architecture allows for easy extension and modification of the application. In the next phase of the project, the following features will be introduces: GPU status monitoring, identifying users who generate the majority of local network traffic, add a database to store sensor readings, and conduct further analytics.

5.1 Acknowledgements

I would like to express my sincere gratitude to the entire JINR team and the START team for making my participation in this event possible. I am also grateful to our supervisors, Dmitry Belyakov and Maxim Zuev, for their comprehensive support. Gennady Karpov deserves special thanks for his productive cooperation on the project. Additionally, I would like to extend my gratitude to Andrey Andreevich Sushchenko from the Far Eastern Federal University for providing me with the opportunity to work outside the walls of my university and develop my skills. The one-and-a-half-month program allowed to deepen my knowledge in programming and system design, and master new technologies.

REFERENCES

1. Telegraf | InfluxData. — 2024. — URL: https://www.influxdata.com/time-series-platform/telegraf (visited on 03/11/2024).
2. SALSA. — 2017. — URL: https://home-hlit.jinr.ru/#/ (visited on 03/06/2024).
3. *Karpov G.* Development of tools for visualization of monitoring the state and usage of computing nodes of the Heterogeneous HybriLIT platform. — 2024.
4. What is streaming data? — 2024. — URL: https://aws.amazon.com/what-is/streaming-data/ (visited on 03/18/2024).
5. *Гарри Персиваль Б.Г.* Паттерны разработки на Python TDD, DDD и событийно-ориентированная архитектура. — O'Reilly Media, Inc, 2022.
6. *Ramalho L.* Fluent Python: Clear, Concise and Effective Programming. — O'Reilly Media, Inc, 2022.
7. select - Waiting for I/O completion - Python 3.12.2 documentation. — 2024. — URL: https://docs.python.org/3/library/select.html (visited on 03/26/2024).
8. selectors - High-level I/O multiplexing - Python 3.12.2 documentation. — 2024. — URL: https://docs.python.org/3/library/selectors.html (visited on 03/26/2024).
9. WebSockets - FastAPI. — 2024. — URL: https://fastapi.tiangolo.com/reference/websockets/ (visited on 04/07/2024).
10. asyncio - Asynchronous I/O - Python 3.12.2 documentation. — 2024. — URL: https://docs.python.org/3/library/asyncio.html (visited on 03/26/2024).
11. Streams - Python 3.12.2 documentation. — 2024. — URL: https://docs.python.org/3/library/asyncio-stream.html (visited on 03/28/2024).
12. *Швец А.* Погружение в паттерны проектирования. — 2021.