



Dipartimento di Ingegneria dell'Informazione
M.Sc in Computer Engineering

Distributed Weather Monitoring System

Arsalen Bidani, Bruno Augusto Casu Pereira De Sousa, Chuer
Huang

879II Distributed Systems and Middleware Technologies
Year 2022/2023

Contents

1	Introduction	1
2	Actors and requirements	1
3	System architecture	3
3.1	Regional Server (Erlang)	4
3.2	Central Server (Erlang)	7
3.2.1	Communication interface	8
3.2.2	Web socket	10
3.3	Application Server (Tomcat)	14
3.4	Sensor node	15
4	Deployment and testing	16
4.1	Back end network testing	16
4.2	Front end testing and platform deployment	22
5	Conclusions	27

1 Introduction

The project presented here is the implementation of a monitoring platform using a distributed Erlang server and a tomcat based server running a web application for remote evaluation of weather parameters. The goal of the platform is to provide a flexible and expandable monitoring application, displaying real-time data from different remote regions, as well as an event and warning interface.

In the developed application, emulated sensors will periodically generate temperature and humidity data, transmitting the information to a set of deployed Regional Servers. These components will use an exposed REST interface to receive data frames from the sensors and, by exploiting the Erlang message-passing system, send the telemetry data to a Central Server. At this central unit a web socket will be used to make the stream of data available to a Tomcat based server, that will collect and display the data in real-time charts. On the tomcat Application server, the readings will also be processed, as a way to generate events and warnings, depending on a set of pre-defined thresholds.

A detailed description of the interfaces used to pass the data between the distributed application will be described in the report, as well as the overall system operation and tests conducted to demonstrate the deployed platform. The complete source code for all servers is available at: <https://github.com/brunocasu/weather-monitoring-erlang-server>.

2 Actors and requirements

Three main actors are considered in the monitoring system, they are Sensors, Network manager and End Users:

Sensors are devices scattered through the monitoring area, and able to transmit temperature or humidity measurements over the underlying REST protocol to one of the regional servers.

Network manager is able to access the regional servers and check the sensors log, confirming the information and identification of the messages. The manager can also access the central server to check connectivity and the web socket messages content, ensuring the platform correct operation.

End user is able to visualize live data plots, averages, and warning events on measure crosses through the web application.

To provide the mentioned services to the main actors, a number of functional requirements were established for the distributed system developed:

- The Regional Server must provide a message passing service, exposing a REST interface for nearby sensor nodes to send POST request messages containing the temperature and humidity readings.
- The Regional Server must provide an Erlang message sender interface, to forward incoming sensor data to the Central Server.
- The Regional Server ID must be added in the messages sent to the Central Server.
- The Regional Server can provide a Log function to store latest sensor readings in memory.
- The Regional server can provide a static HTML page as a GET response to its main resource address. The page will contain the sensor Log and Server information.
- The Central Server must provide connectivity to all Regional Servers using the Erlang message passing functions.
- The Central Server must implement a web socket service, and provide an info message every time a sensor message arrives.
- The Central Server web socket must handle multiple connections simultaneously, serving the sensor messages to all hosts connected.

- The Application Server must use a Tomcat based web application service, that will connect to the Central Server web socket, and receive the sensor data stream.
- The Application Server must implement a web application that allows End Users to connect to the service and check the environment parameters for every region.

3 System architecture

The overall system architecture proposed uses three different types of servers, they are Regional, Central, and Application servers. With this configuration, the services and functions of the monitoring platform are distributed, using different machines to execute the necessary tasks for the system. An overview of the implemented architecture is shown in Figure 1.

This architecture then exploits the Erlang functions for message passing and multi-threading as a way to efficiently transmit the sensor's information in the network, allowing multiple regions to be connected to the monitoring platform. With this, the access to the platform is provided by the Regional servers, by exposing a REST interface on the local Erlang nodes. This interface is made by the Cowboy library, available at: <https://github.com/ninenines/cowboy>.

As those servers receive the data messages, they will re transmit the information to a Central Server, using the Erlang message interface. This central node then will run a web socket, also using the Cowboy library, allowing the deployed Application Server to connect to it and receive a stream of data from the multiple Regional Servers. This server then will manage connections and also filter and process the frames sent by the sensors, providing the data stream in the json format. With the data provided, the last node of the monitoring platform, the Application Server, will implement a user interface and a dynamic graphical representation of the data, allowing connected users to monitor the weather parameters on the different regions.

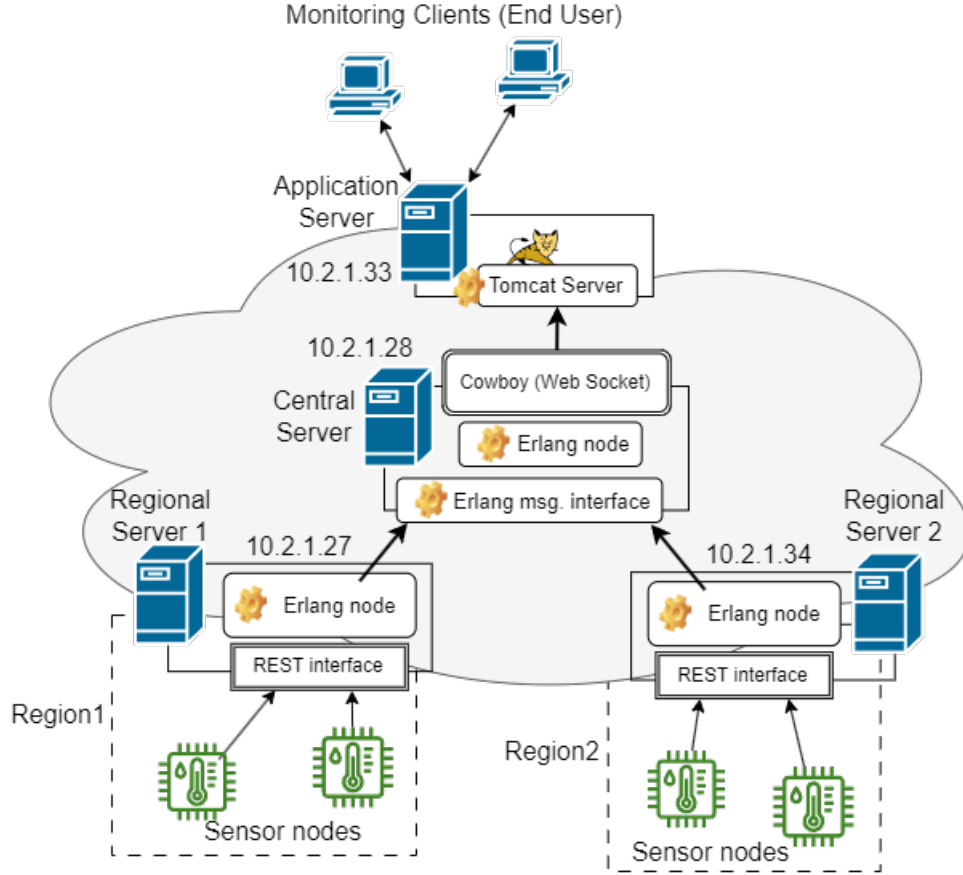


Figure 1: Weather Monitoring System network architecture.

3.1 Regional Server (Erlang)

Considering that a distributed monitoring application was proposed, the platform must provide connectivity to a set of sensors on separate locations, defined then in the scope of this project as Regions. To provide this service, a Regional Server node was developed using Erlang. For the basic operations, this type of server was designed using the Cowboy library, exposing a resource that can be accessed by a REST interface. The resource then provides a response for POST and GET requests, using the Cowboy listener function. For the GET request response, the server replies with a static HTML page, used to provide server information, such as the Regional Server ID (identify which

region is being monitored) and the latest sensor readings. For the POST request reply, a string of plain text is sent, informing the correct reception of the operation.

Using a custom handler for the POST request message, the server main operation processes the frame received (on this system, the sensor must send a string of plain text as the body of the POST request), parsing the fields in the message. An example of usage when retrieving the sender Sensor ID is as follows:

```
1 %% Handler for POST/GET messages
2 server_request_handler(<<"POST">>, true, Req0) ->
3     {ok, PostContentBin, Req} = cowboy_req:read_urlencoded_body(Req0),
4     SensorIDBin = proplists:get_value(<<"sensor.id">>, ...
```

The format of the string sent by any sensor node must use the following configuration (example):

```
1 'sensor_id=TS01&sensor_data=20&sensor_data_type=temperature&time=1450879'
```

As the correct field values are read from the POST message, and the server provides the POST reply message, the Regional Server will add received the frame in a Log, using the Erlang List data structure. This List will be managed by a separate task function in the Regional Server and will be spawned as the server is executed. The objective of this List is to store the latest sensor readings, adding the information displayed in the static HTML page. To format the data properly for the page, a custom function is used, adding the syntax necessary to show the sensor data in a columnar form in the HTML page. The information is then added to the HTML code in the GET response, by reading the current Log List and adding the strings in the body of the message. The web page for Region 1 is shown in Figure 2 (only data and data type columns).

After storing the frame in the server Log, an Erlang message is sent by the Regional Server to the interface in the Central Server node (a new frame is formed). This message will use the atom 'interface' to identify the process in the receiver node. Noticeably, the address of the receiver must be configured in the known hosts file on the Ubuntu OS (this is done by editing the file

STATIC HTML - REGIONAL MONITORING SERVER

Sensor Log - Server ID: RS01

READING	DATATYPE
92	humidity
45	temperature
91	humidity
16	temperature
92	humidity
44	temperature
96	humidity
16	temperature
99	humidity
13	temperature
98	humidity
32	temperature

Figure 2: Screen capture of the Regional Server static HTML page.

/etc/hosts), allowing for the address resolution for the atom 'central', once the server executes the command:

```
1    ...
2    CENTRAL_SERVER_NODE = erlcomm.interface@central
3    {interface, CENTRAL_SERVER_NODE} ! {{SERVER.ID, SensorIDBin, DataBin, ...
    DataTypeBin, TimeBin}, self()}; ...
```

The Regional Server also adds its internal Server ID parameter , to be used on the application server, as a way to identify which region the data is coming from. Figure 3 illustrates the overall interfaces used on the Regional Server communication with the Sensor nodes.

In the illustration, it is noticeable that the node name created for the REST server was regional_server@localhost. This implementation is a constraint found in the Cowboy library, as it set the Erlang node name based on its internal configurations. For the servers communication using the Erlang message passage architecture, a solution was designed and will be detailed in section 3.2.1.

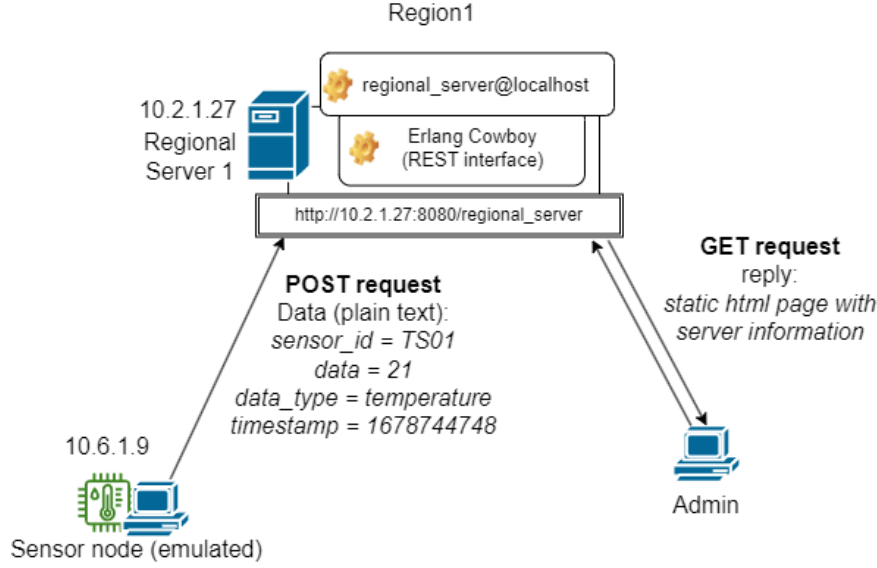


Figure 3: Communication scheme between Sensor nodes and a Regional Server.

3.2 Central Server (Erlang)

As multiple access points are provided in the monitoring network, an Erlang-based Central Server was also introduced in the platform, gathering the sensor data from all the regions and providing a data stream for the Application Server. For this server, the Cowboy library was used to deploy a web socket, combined with a custom interface for fast communication between Regional nodes and the Central node, using the Erlang message-passing system. The communication interface is used as an alternative solution due to some constraints found when using the Cowboy library. Thus, in the Container running the Central Server application, two Erlang nodes will be set, one created for the web socket and one for the communication interface. The overall communication interfaces for the Central Server node are shown in Figure 4.

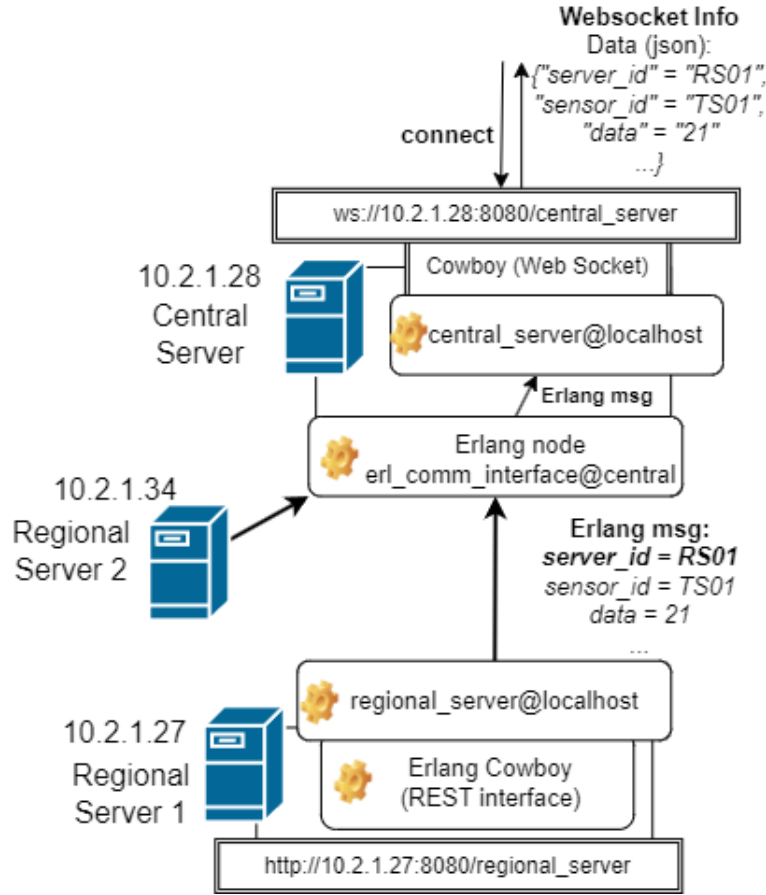


Figure 4: Communication scheme between Regional Servers and the Central Server.

3.2.1 Communication interface

Cowboy is primarily designed to handle HTTP requests and responses between a client and a server as it provides a powerful and flexible framework for building web applications. However, the overall system design is not optimized for Erlang inter-node communication as the Erlang name node configuration is not clearly exposed and customizable in the web socket creation. Therefore, it was not possible to define specific node names for the servers in the platform, which creates an issue for setting a direct connection between the Regional Server and the Central Server node in the monitoring platform.

To overcome this limitation, an additional Erlang interface was designed to allow the communication between servers deployed in different Containers. This interface provides a smooth transit point, and the middle node name “erl_comm_interface@central” is flexibly set, allowing the proper IP address resolution in the Regional Servers transmission to the Central Server communication interface. Though this interface solves the connectivity issues for the data path of the platform, it is not a complete solution, as the Regional Servers still have its Erlang node name created from Cowboy. This then makes the backward communication impossible (erl_comm_interface@central to regional_server@localhost). The overall workaround is illustrated in Figure 5.

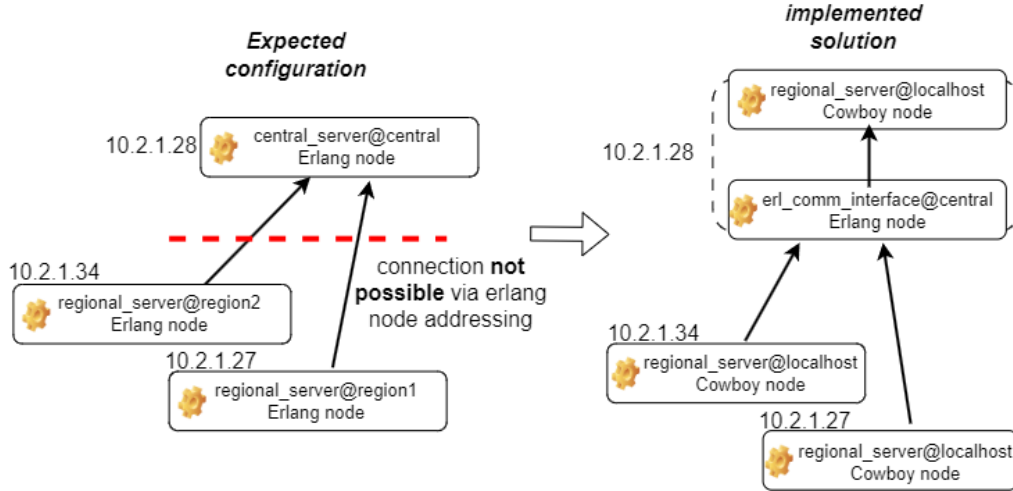


Figure 5: Erlang communication interface adaptation for the Central Server.

The erl.interface component is then an Erlang module with several functions related to a message listener and sender, as the main purpose of this component is to transmit the messages received from the Regional Servers to the Erlang listener in the Central Server web socket node. A number of functions were then introduced in the application:

- The “start_listener/1” function initializes the listener process and registers it under the name “interface”. The “loop/1” function is the main process of the listener and waits for incoming messages. When a message arrives, it validates it using the “validate_message/1” function and

attempts to send it to a central server using the “retry_send/4” function. If the sending fails due to an error, the “retry_send/4” function will retry up to a maximum number of times or until a timeout is reached. If the message is invalid, it will be ignored.

- The ‘retry_send/4’ function uses a times counter to retry sending the message, with a delay between retries defined by “RETRY_INTERVAL”. The function uses the “catch” construct to handle errors and sleeps for the defined interval before attempting to resend the message. If the maximum number of retries is exceeded or the timeout is reached, the function will ignore the message.
- The “validate_message/1” function ensures that a received message is valid according to the defined constraints. The message must have a valid server ID, sensor ID, data type, and timestamp. The function checks each of these components and returns ‘true’ if all are valid, or “false” otherwise. The function also logs the received message to the console for debugging purposes.

3.2.2 Web socket

As the multiple Regional server provide the sensor data to the communication interface, the weather monitoring system design uses a Cowboy web socket to generate a stream of data in the json format, allowing the application server to retrieve and display the readings. The functionalities provided by this model uses three main functions, provided by Cowboy, to handle the web operations:

- websocket_init/1 - This function is called by the Cowboy listener every time a new connection is requested. This is the initialization of the session, and on the application developed for the Central Server, this function will execute a procedure to start the communication with the previously introduced communication interface.
- websocket_handle/2 - This function is called when the web socket receives a request. On the current implementation, the web socket does not reply to any requests.

- `websocket_info/2` - The main operations of the Central Server were developed in this function call. This function is called internally in the server by other spawned processes. After the call, the parameters forwarded by the communication interface (sender Server ID, Sensor ID, Data, Data type and timestamp) will be converted to a json string, and the `info` function will return to the Cowboy handler the frame to be transmitted. The code implemented then is:

```

1 websocket_info({ServerID, SensorID, Data, DataType, Time}, State) ->
2     Body = build_json_reply(ServerID, SensorID, Data, DataType, Time),
3     ...
4     [{text, Body}], State};

```

Considering that the design choice for message passing in the monitoring network (custom Erlang messages are sent from regional nodes to the central node), an Erlang listener loop was introduced inside the provided Cowboy web socket.

As the first session is created, the listener and list processes are spawned, (the PID of the current session is added to the list). Once a message arrives at the listener, the task reads all PIDs that are currently registered on the list and forwards the sensor data message to the `websocket_info` handler of all current threads (sessions) running. The function called at every start of new sessions is:

```

1 start_monitoring_listener(WSPID) ->
2     %% IMPORTANT: data_comm is the PID for the regional servers to send the ...
3     message
4     %% check for registration
5     List = registered(),
6     Status = reg_check(List),
7     WsPidLst = pid_to_list(WSPID),
8     if
9         Status == clear -> %% First connection
10         io:fwrite("-p-n", ["data_comm init..."]),
11         register(data_comm, spawn(fun()->loop() end)), %% process that ...
12         listens to incoming erlang messages
13         start_task(log, listener_task, WsPidLst); %% process that keeps the ...
14         log of websockets PID
15     true -> %% New session
16         log_access(write, WsPidLst),
17         already_running
18     end.

```

This architecture then allows the Erlang communication interface developed to reach all sessions created at the Central Server web socket task (handles multiple sessions in the web socket). An Illustration of the process handling is shown in Figure 6.

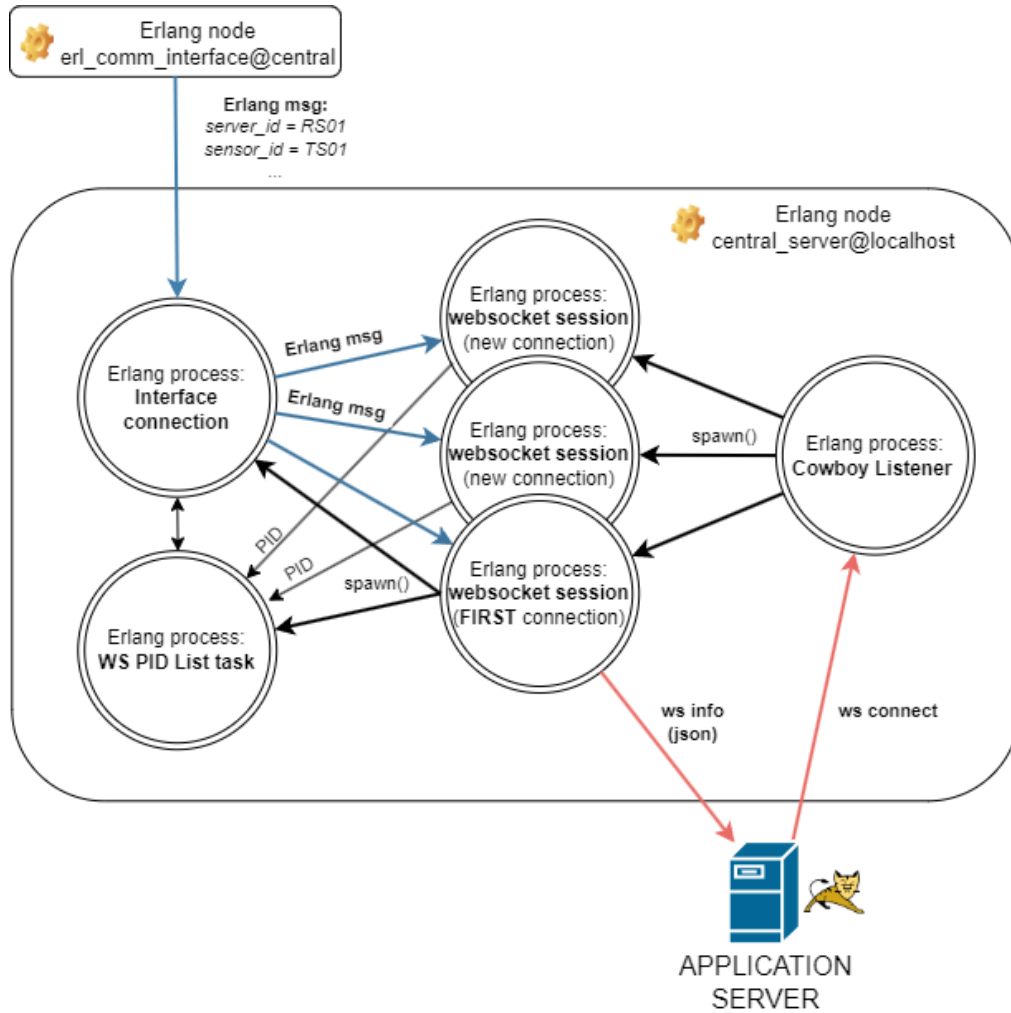


Figure 6: Central Server internal processes scheme.

In this architecture, a single process is then used to receive all messages from the communication interface and call the `websocket_info` process:

```
1 loop() ->
2 receive
```

```

3      {Msg, _From} -> %% received from the communication interface ...
      (erl_comm_interface@central node)
4      PIDList = log_access(read, []),
5      io:fwrite("\n", ["sending to websocket info..."]),
6      send_all(PIDList, Msg), %% message is sent to all websocket PIDs ...
      (connected nodes)
7      loop()
8  end.

```

From the message handler used inside the loop function presented, all the sessions PIDs are retrieved and used in the `send_all/2` function, that iterates over the list and transmits the sensor data frame to the corresponding websocket info thread. The message flow between the threads running at the Central Server are shown in the example:

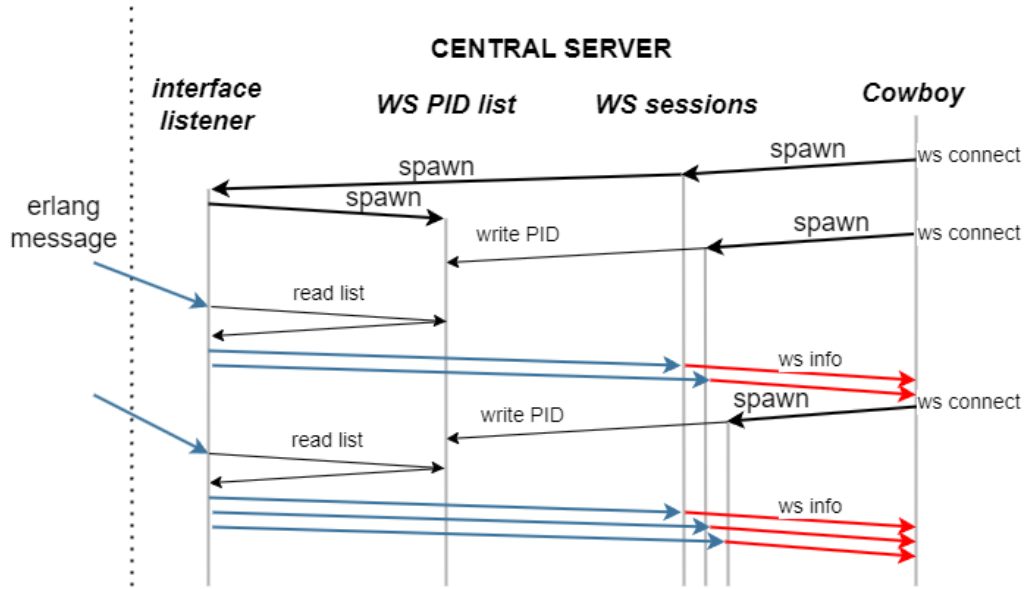


Figure 7: Central Server internal message flow scheme.

As messages flow from the Regional servers to the Central server using the Erlang format, on the web socket info function a formatting function was also added, as a way to convert the information from the sensor to a JSON format. This conversion simplifies processing at the application server, and ensures fast updates in the graphical interface used to show the data. The format set for each sensor data frame, that will be parsed in the Application server, is as the example shown:

```
1      {"server_id":"RS01", "sensor_id":"TS01", "data":"23", ...  
      "data_type":"temperature", "time":"1679904045432"}
```

3.3 Application Server (Tomcat)

As a way to diversify the technologies used in the Weather monitoring platform, a Tomcat based server will be used for the application layer of the system. In this section, the data generated by the monitoring sensors will be parsed and converted to a readable format, provided by a graphical interface app.

As an efficient way to set the user interface, the React java script library was used. This allows then a simple integration along with the Tomcat running server, as the application will request the connection to the web socket and use the provided sensor data to build a number of charts (on the current configuration, two Regional Servers are displayed in the application).

As an example on how React manages its service, the following code illustrates the procedure to connect to the web socket, and parse the json messages received (simplified code, not used in the final implementation):

```
1      function App() {  
2          ...  
3          const ws = new WebSocket("ws://10.2.1.28:8080/central_server");  
4          ...  
5      };  
6      ws.onmessage = function (event) {  
7          const json = JSON.parse(event.data);  
8          try {  
9              if ((json.event = "data")) {  
10                 ...  
11             }  
12         } catch (err) {  
13             console.log(err);  
14         }  
15     };  
16 }
```

Once the application using React is built, it is added to the /webapps folder in Tomcat, the catalina server is launched with this application. The graphi-

cal interface can be accessed through the URL: `http://10.2.1.33/monitoring`, considering that the Application Server runs on the container with address 10.2.1.33. The overall connections to this server are then shown in Figure 8.

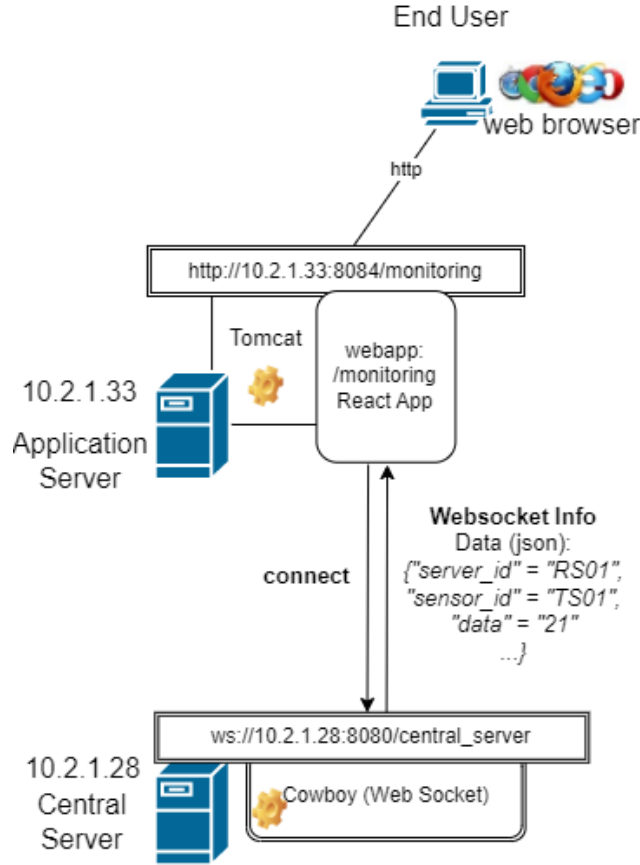


Figure 8: Central Server internal message flow scheme.

3.4 Sensor node

As a way to simulate sensor nodes, two simple applications were created, as a data source for the platform testing. The first approach uses Java and sends messages to only to Region 1, using two different sensor IDs (the data generated corresponds to temperature and humidity readings, both generated randomly); The second approach uses Python and sends messages to both

Region 1 and Region 2, and can emulate the behavior of up to eight sensor IDs (two temperature and two humidity data sources for each Region). The Java class "SensorNode" runs an infinite loop that simulates sending sensor data to a regional server over HTTP at a specified interval. The "sendSensorData" creates a connection to the regional server, sets the request method to POST, and sets the request body to the generated sensor data, then sends the request to the server and retrieves the response code and response body from the server. The Python code used a `run_test` method, where the number of sensors simulated and the target Region can be configured, as well as the duration of the run.

4 Deployment and testing

4.1 Back end network testing

To ensure that the message-passing system developed for the monitoring system was providing connectivity between all Erlang servers deployed a test was conducted on the back portion of the platform. This section then evaluates the access interface, provided by Regional Servers, and the data stream management, provided by the Central Server.

Using two Containers in the network, two independent Regional Servers were deployed. The Container with IP address 10.2.1.27 was used for representing Region 1, accessed by the resource `http://10.2.1.27/regional_server` and Container with IP 10.2.1.34 was used for Region 2, with the resource exposed in the URL `http://10.2.1.34/regional_server`. On the platform internal network interface, the nodes have the following Erlang node name: `regional_server@Distributed2022` and `regional_server2@Distributed2022` in Containers 10.2.1.27 and 10.2.1.34 respectively.

For the central Server, deployed on the Container with IP 10.2.1.28, two Erlang nodes were created, one with node name `erl_comm_interface@central` (the atom 'central' is configured in Containers 10.2.1.27 and 10.2.1.34 `/etc/hosts` file as IP 10.2.1.28), and one with node name `central_server@Distributed2022`

(created by Cowboy). Once the Regional Servers connect and transmit the sensor readings to the communication interface, the web socket will receive the frames and produce the json format strings to be streamed using the info function. To test the reception of such frames, the Postman software was used to capture the packets streamed by connecting to web socket at URL ws://10.2.1.28:8080/central_server. The network route tested is illustrated in Figure 9 (highlighted with the red arrows):

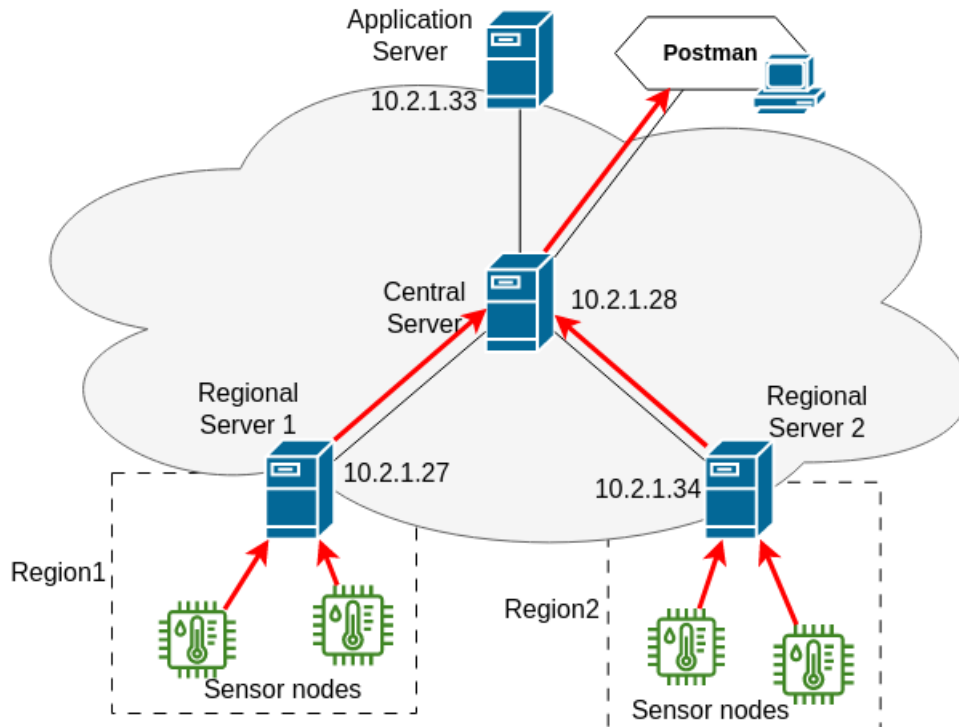


Figure 9: Back End data path testing.

Once the Servers were running the sensor node programs were executed providing data frames to the network. Captures from the Python execution and are shown in Figure 10 from the internal Erlang Regional servers are shown in Figure 11.

In the printed messages at the Regional Server, it is possible to see the when Cowboy calls the REST interface handler (printed as "message received..."), when the custom POST handler is called (printed as "POST handler..."), and

```
C:\Windows\py.exe
Regional Server Reply: Regional Server Echo
Sensor: TB02 Sent Reading: 54
Regional Server Reply: Regional Server Echo
Sensor: HB02 Sent Reading: 87
Regional Server Reply: Regional Server Echo
Sensor: TA01 Sent Reading: 78
Regional Server Reply: Regional Server Echo
Sensor: HA01 Sent Reading: 87
Regional Server Reply: Regional Server Echo
Sensor: TB02 Sent Reading: 53
Regional Server Reply: Regional Server Echo
Sensor: HB02 Sent Reading: 99
Regional Server Reply: Regional Server Echo
Sensor: TA01 Sent Reading: 75
Regional Server Reply: Regional Server Echo
Sensor: HA01 Sent Reading: 97
Regional Server Reply: Regional Server Echo
Sensor: TB02 Sent Reading: 54
Regional Server Reply: Regional Server Echo
Sensor: HB02 Sent Reading: 98
Regional Server Reply: Regional Server Echo
Sensor: TA01 Sent Reading: 77
Regional Server Reply: Regional Server Echo
Sensor: HA01 Sent Reading: 91
Regional Server Reply: Regional Server Echo
Sensor: TB02 Sent Reading: 57
Regional Server Reply: Regional Server Echo
Sensor: HB02 Sent Reading: 93
Regional Server Reply: Regional Server Echo
```

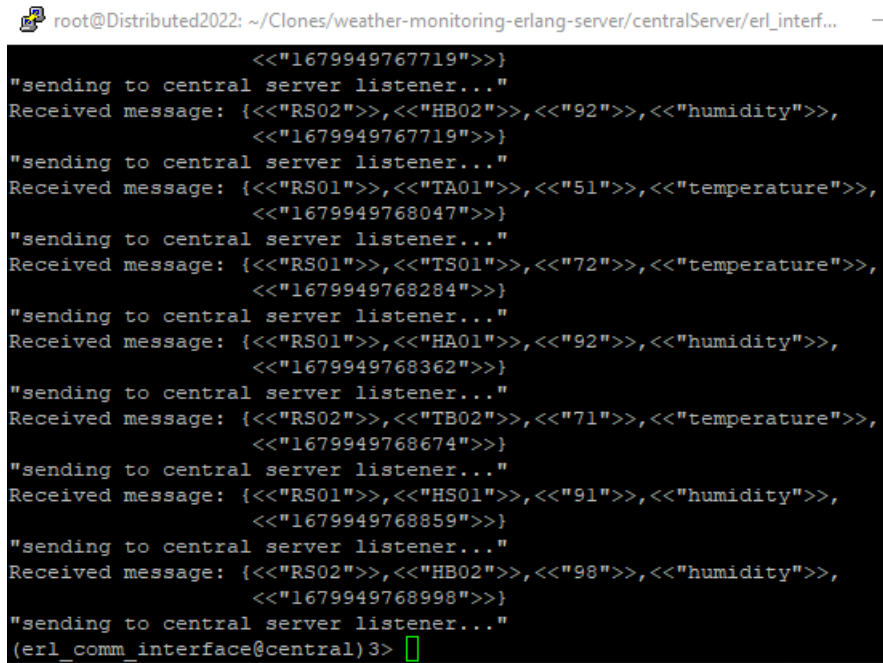
Figure 10: Screen capture from python3 terminal when executing the sensor emulation software.

```
root@Distributed2022: ~/Clones/weather-monitoring-erlang-server/regi
<<"HA01">>
"message received..."
"POST handler..."
"data from..."
<<"HS01">>
"message received..."
"POST handler..."
"data from..."
<<"TA01">>
"message received..."
"POST handler..."
"data from..."
<<"TS01">>
"message received..."
"POST handler..."
"data from..."
<<"HA01">>
"message received..."
"POST handler..."
"data from..."
<<"HS01">>
```

Figure 11: Screen capture from the Regional Server 1 terminal.

when the frame is parsed, acquiring the data from the sensor POST request (printed as "data from..." «"SENSOR ID"»).

Once the frames are received by the POST handler in the Regional Servers, the messages will be forwarded to the communication interface at the Central Server, with the Erlang node name `erl_comm_interface@central`. After the listener at this node receives the frames, it checks possible errors or malformed messages forwarding the correct messages to the Erlang listener at the web socket node. Captures of the terminals running the communication interface node are shown in Figure 12 and the Central Server node are shown in Figure 13.



```
root@Distributed2022: ~/Clones/weather-monitoring-erlang-server/centralServer/erl_interf...  
    <<"1679949767719">>}  
"sending to central server listener..."  
Received message: {<<"RS02">>,<<"HB02">>,<<"92">>,<<"humidity">>,  
    <<"1679949767719">>}  
"sending to central server listener..."  
Received message: {<<"RS01">>,<<"TA01">>,<<"51">>,<<"temperature">>,  
    <<"1679949768047">>}  
"sending to central server listener..."  
Received message: {<<"RS01">>,<<"TS01">>,<<"72">>,<<"temperature">>,  
    <<"1679949768284">>}  
"sending to central server listener..."  
Received message: {<<"RS01">>,<<"HA01">>,<<"92">>,<<"humidity">>,  
    <<"1679949768362">>}  
"sending to central server listener..."  
Received message: {<<"RS02">>,<<"TB02">>,<<"71">>,<<"temperature">>,  
    <<"1679949768674">>}  
"sending to central server listener..."  
Received message: {<<"RS01">>,<<"HS01">>,<<"91">>,<<"humidity">>,  
    <<"1679949768859">>}  
"sending to central server listener..."  
Received message: {<<"RS02">>,<<"HB02">>,<<"98">>,<<"humidity">>,  
    <<"1679949768998">>}  
"sending to central server listener..."  
(erl_comm_interface@central)3> █
```

Figure 12: Screen capture from the communication interface Erlang node terminal.

On the communication interface, all the individual fields of the sensor message are printed, as a way to verify the content sent into the web socket. On the Cowboy node running the web socket, the prints refer to the function call of the info handler, when the listener task sends the received frames. When the message "websocket stream..." is seen in the logs, it identifies when each

```
root@Distributed2022: ~/Clones/weather-monitoring-erlang-server/centralServer/
"sending to websocket info..."
"websocket stream..."
"websocket stream..."
"sending to websocket info..."
"websocket stream..."
"websocket stream..."
"sending to websocket info..."
"websocket stream..."
"websocket stream..."
"sending to websocket info..."
"websocket stream..."
"websocket stream..."
"sending to websocket info..."
"websocket stream..."
"websocket stream..."
"sending to websocket info..."
"websocket stream..."
"websocket stream..."
"sending to websocket info..."
"websocket stream..."
"websocket stream..."
"sending to websocket info..."
"websocket stream..."
"websocket stream..."
"sending to websocket info..."
"websocket stream..."
"websocket stream..."
```

Figure 13: Screen capture from the Central Server terminal.

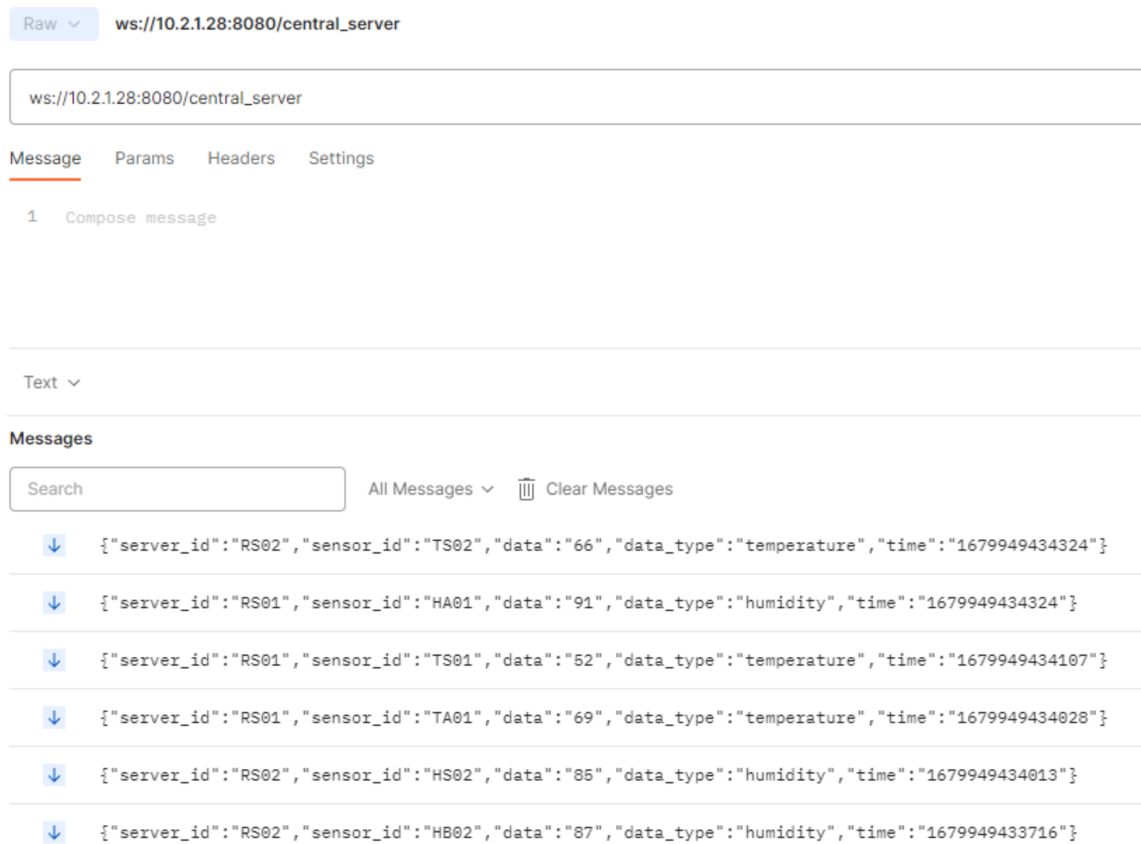


Figure 14: Screen capture from Postman connected to the Central Server web socket.

session created sends the json frame (in the example only two connections were established). The captured packets using Postman are shown in Figure 14.

4.2 Front end testing and platform deployment

With the back end connection tested and running, the final testing and demonstration of the weather monitoring system were executed. To run these tests, the Tomcat server is started and the Regional and Central Servers are initialized, establishing the full data path to be tested. Once the Application Server connects to the running web socket, the sensor node software is then executed, sending the simulated readings. The network section to be tested in this phase is shown in Figure 15.

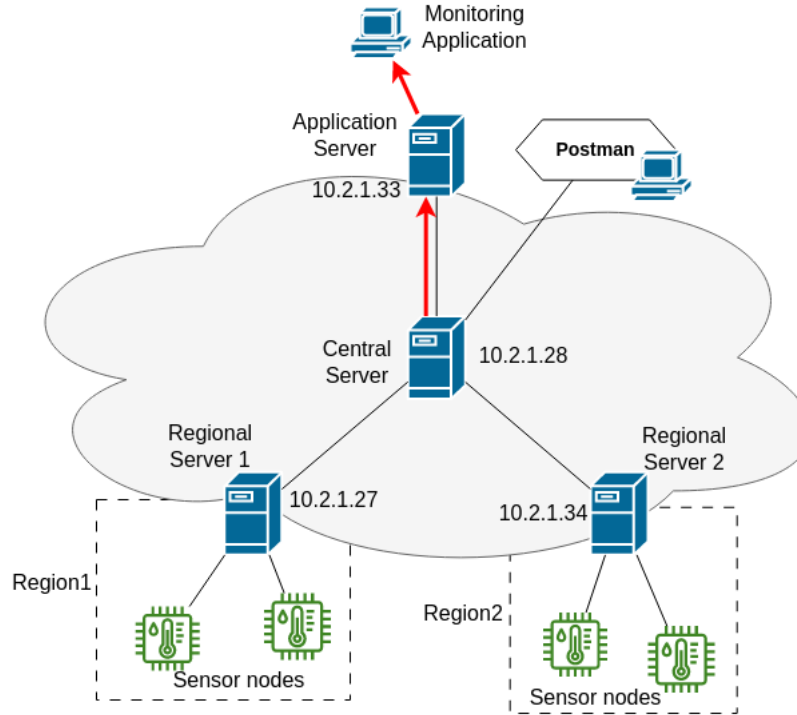


Figure 15: Front end testing scheme.

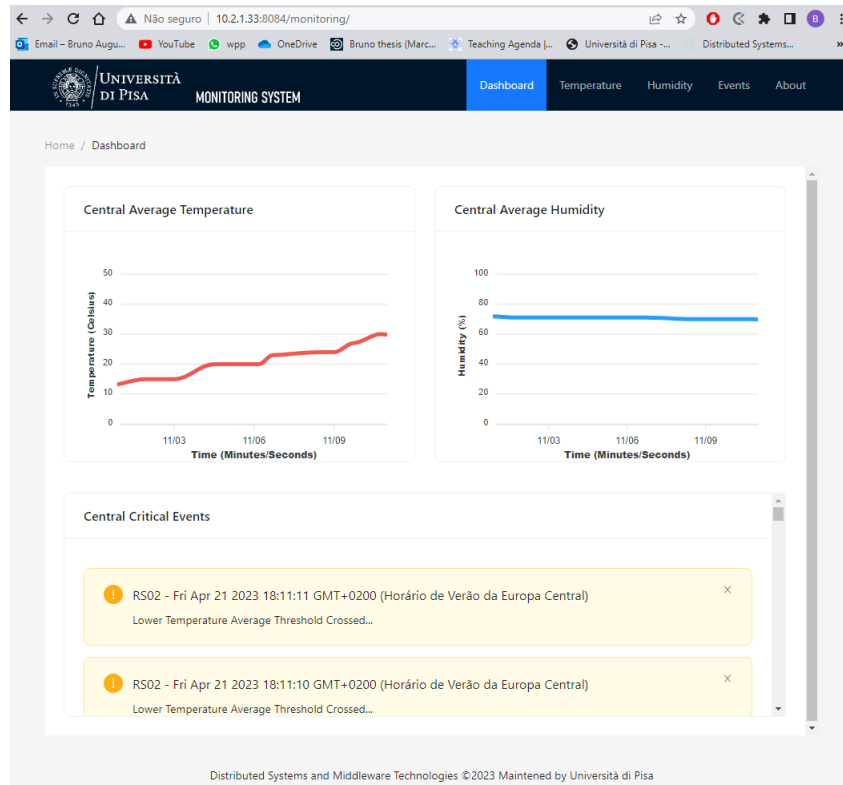


Figure 16: React js application - User interface Main Page.

From a remote computer, a web browser was launched, connecting to the Tomcat server resource /monitoring, where the graphical interface is running. Once the data stream is set (python code execution), it is possible to verify on the charts created the sensor readings, as well as the additional services provided (average temperature and humidity of each Region and an Event system that triggers when the values crosses a defined threshold). The Main page of the User interface is shown in Figure 16.

In Figure 17 and Figure 18 the readings from two sensors are shown on each Regional Server, demonstrating the information on each region (readings are provided every second in this demonstration, however, in a real application, the time values between readings are expected to be much larger, considering weather monitoring).

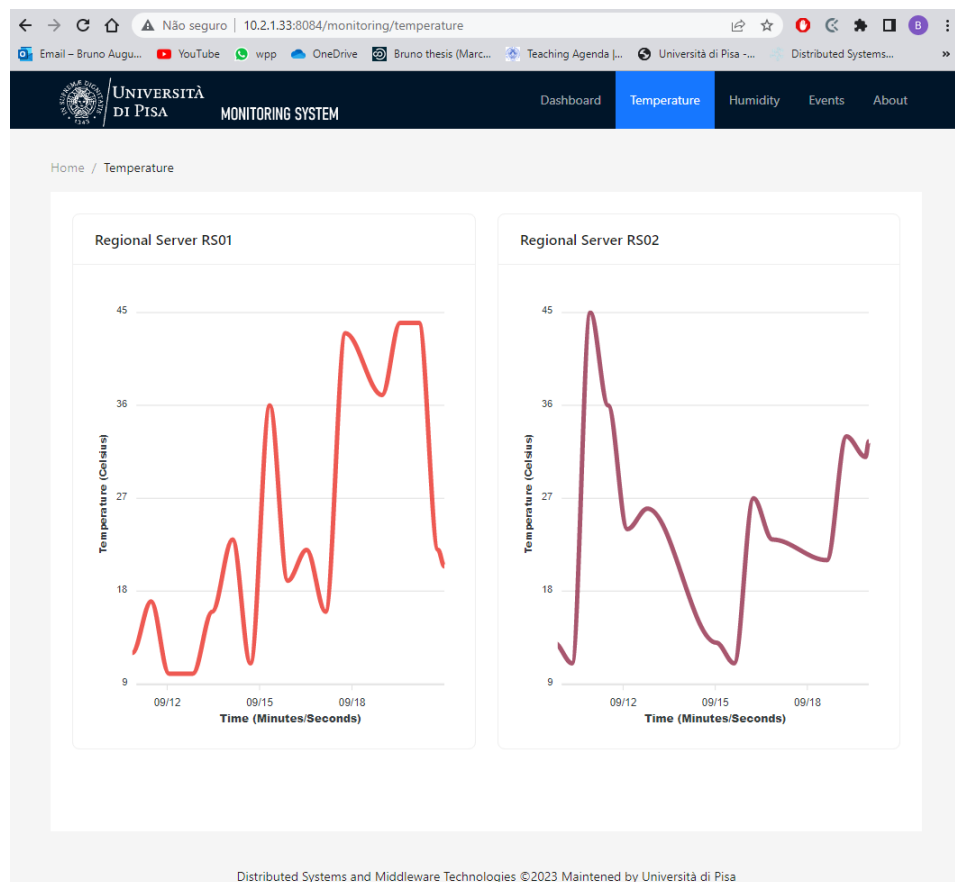


Figure 17: React js application - Temperature Monitoring Page.

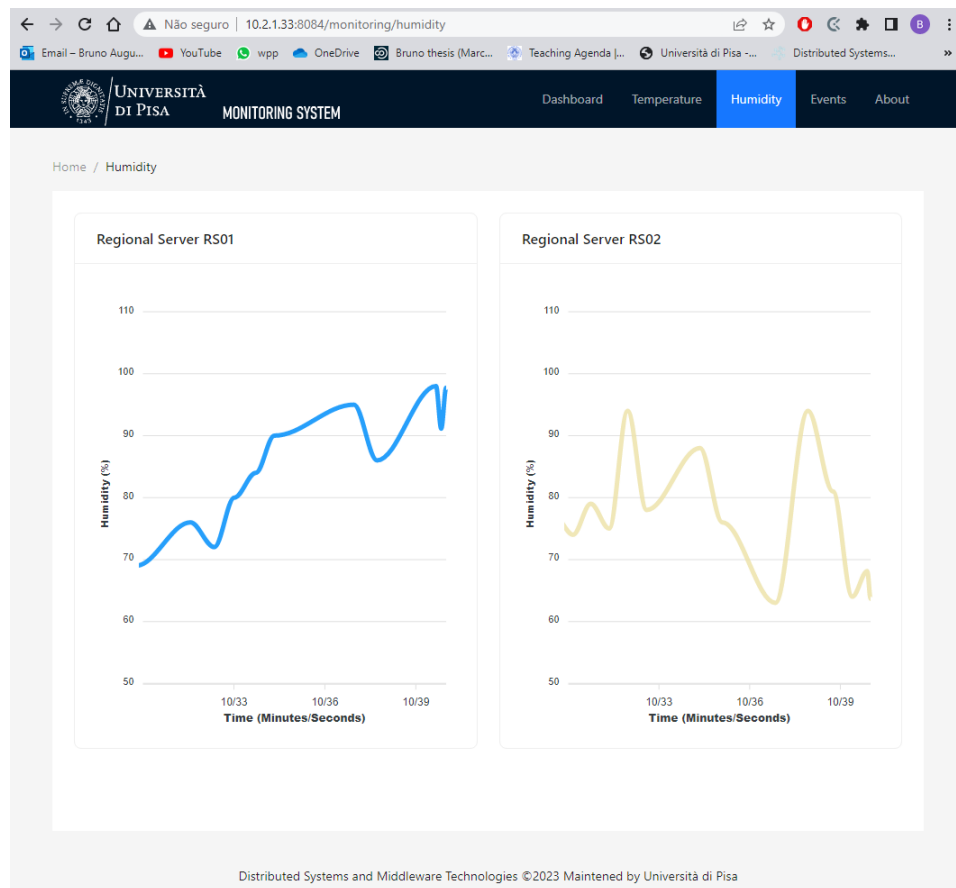


Figure 18: React js application - Humidity Monitoring Page.

Also, as a demonstration of the system scalability and flexibility, while the default sensors were generating data and transmitting it to the Region 2 server, an additional temperature sensor was created and added to the network. The readings from this extra sensor were then immediately added to the Region 2 temperature chart, allowing a quick comparison on the readings. This whole operation was performed without any reconfiguration of the servers, as the platform allows new sensor to be added automatically to the logs. Figure 19 shows the chart when the extra temperature sensor was activated.

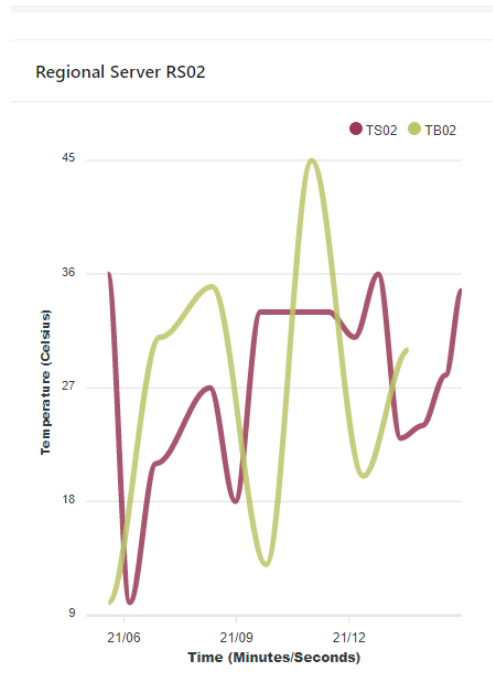


Figure 19: React js application - Temperature monitoring with multiple sensors per Region.

Finally, to test the system event handling, the simulated temperature readings were altered to be much higher than the average 30°C that is the default configuration. In this scenario, the Application server detects the high average value and generate warnings on the graphical interface, informing the time and which Region the event occurred. Figure 20 shows the Event logs.



Figure 20: React js application - Event log Page.

5 Conclusions

Considering all the interfaces and systems developed for the weather monitoring platform proposed, it is possible to conclude that all functional requirements were sufficiently implemented. The platform allows multiple independent sensors to transmit its readings to a centralized system using a distributed system that handles the frames. With that a user can retrieve information or many remote regions that have connectivity with the monitoring network, as it was demonstrated the system flexibility and scalability. Overall the message passing functions developed were proven to be an efficient way to handle this type of application, as in the tests a large amount of packets were being transmitted in the monitoring network. As for improvements in the platform a database system could be a very useful addition, considering that he users may want to retrieve and process the sensor readings for specific periods.