Practical Work 3 in Cloud Computing

Gustavo Freitas Cunha

Federal University of Minas Gerais (UFMG)
Belo Horizonte - MG - Brazil

gustavocunha@dcc.ufmg.br

1 Task 1: Approach to Maintaining State for Computing the Moving Average

To compute the moving average of CPU utilization, I utilized the `context.env` dictionary, which persists small amounts of data between successive invocations of the `handler` function. This dictionary is used to store the previous moving average value, which is then updated with each new measurement. On each function call, the function retrieves the last stored moving average from `context.env`, calculates the new average based on the current and previous data, and stores the updated value back in `context.env`. This approach ensures that the state necessary for the moving average computation is maintained across invocations, allowing for continuous and incremental updates without relying on external storage.

2 Task 2: Dashboard and Metrics

The monitoring dashboard focuses on visualizing key system metrics such as CPU usage, network egress, and memory cache utilization. CPU usage is tracked per core, displaying real-time averages over 60-second intervals. The network metric shows the percentage of data being sent out, while memory cache utilization is presented as a percentage. These metrics are collected from Redis in real time and plotted dynamically using Plotly. The dashboard allows users to monitor system performance at any given moment, making it useful for understanding resource allocation and potential bottlenecks. The graphs for each metric can be seen in Figures 1, 2, and 3.

From a technical perspective, the dashboard leverages Streamlit for rapid prototyping and rendering, while Redis serves as the data source, providing real-time updates. The sidebar, showed in Figure 4, offers user-configurable settings, allowing adjustments to the refresh rate and the length of the history displayed. This gives users flexibility in monitoring system performance over different time periods. The data is presented through interactive plots, ensuring that the information is easily accessible and digestible for continuous system analysis. The figures below illustrate these aspects: Figure 1 shows the average CPU usage per core, Figure 2 shows the network egress usage, and Figure 3 depicts memory cache usage.

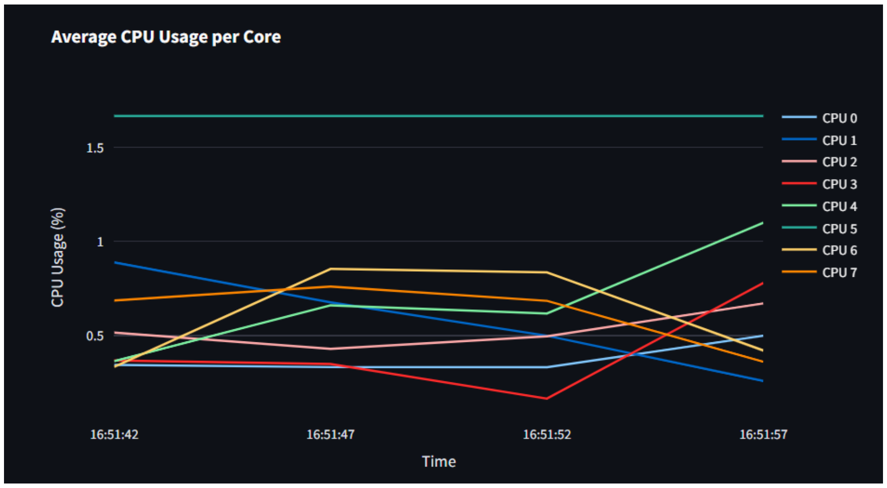


Figure 1: Average CPU Usage per Core

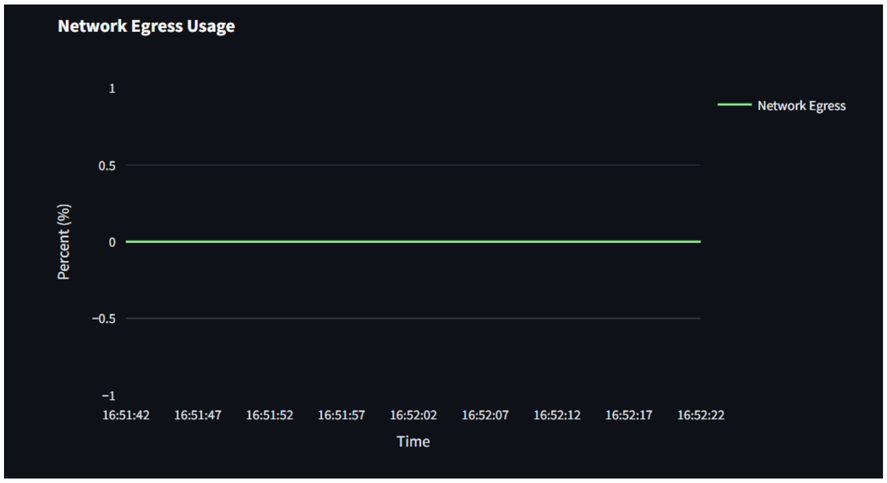


Figure 2: Network Egress Usage

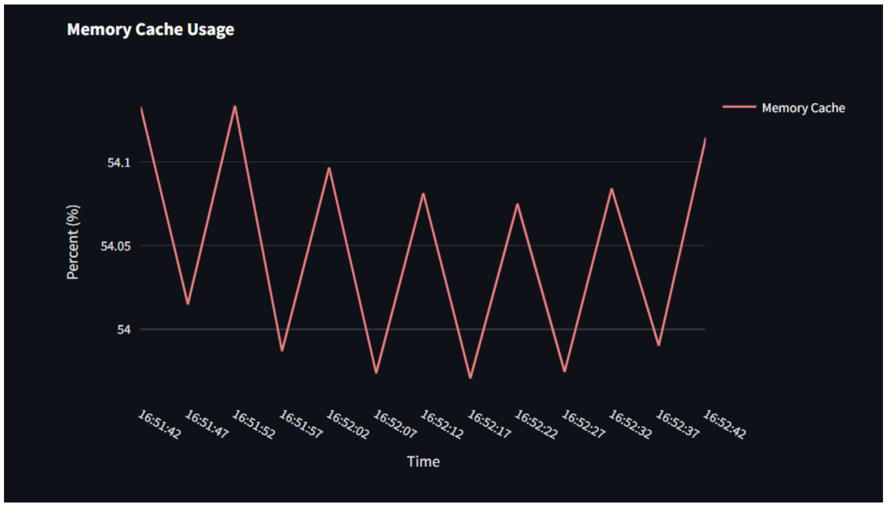
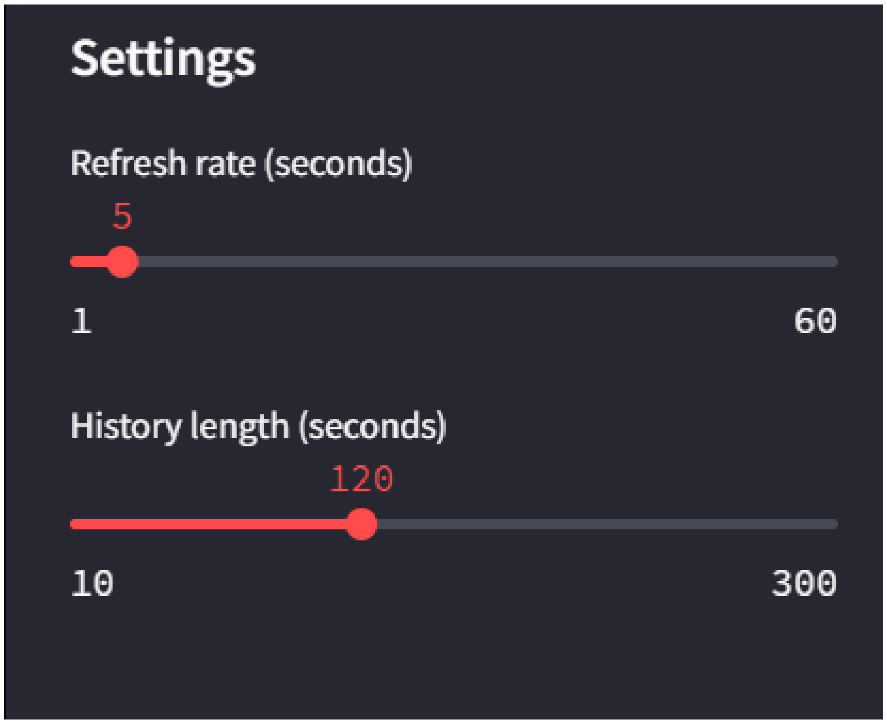


Figure 3: Memory Cache Usage


Settings

Refresh rate (seconds)

5

1 60

History length (seconds)

120

10 300

Figure 4: Dashboard's Sidebar

3 Task 3: Runtime

In my implementation of the serverless runtime, I sought to ensure both compatibility and extensibility. The core script responsible for running the serverless environment is `serverless_runtime.py`, which is located in the `runtime` folder. This file handles the main logic for loading the user-defined function, processing events from Redis, and storing the results back in Redis. To match the runtime provided by the instructors, I maintained the same environment variable configuration and Redis communication mechanisms, ensuring that the runtime behavior is consistent. The `serverless_runtime.py` also includes a key part for loading functions from either a single Python file or a ZIP archive, based on the `RUNTIME_SOURCE_TYPE` environment variable. This allows for additional flexibility, as the instructors' original runtime only supported loading from Python files. The environment variables like `REDIS_HOST`, `REDIS_PORT`, `REDIS_INPUT_KEY`, and `REDIS_OUTPUT_KEY` are used to configure Redis communication, similar to the instructors' deployment. For compatibility with the instructors' runtime, the deployment and configuration are similar, with adjustments only in terms of adding ZIP support for more complex user functions. These changes are reflected in the `deployment.yaml`, where necessary environment variables and paths are defined. Overall, my implementation remains compatible with the instructors' runtime, while enhancing its capabilities by supporting more complex function deployments, all within the same `runtime` folder structure.