System Data Aggregation for Linux-Based Applications

Adam Morrison

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 Cass Avenue, Lemont, IL 60439

Valparaiso University, 1700 Chapel Drive, Valparaiso, IN 46383

Dated: 4 August 2017

ABSTRACT

Single-board computers utilized in embedded applications often employ the Linux Operating System. This fact is true of the single-board computers implemented in Waggle Nodes, a component of the Array of Things project at Argonne National Laboratory. The Linux Operating System contains built-in command line functionality for viewing system metrics and diagnostic data relating to the computer. Yet, this information cannot be viewed in a single easily-accessible location. The research presented in this paper addresses the problem of aggregation of system data for Linux-based computer applications, specifically within Waggle Nodes. The method proposed to solve this problem required the construction of an Application Programming Interface (API) utilizing programming languages and Linux commands that could accomplish three goals. These goals were collection of diagnostic system data from a Linux-based single-board computer, transmittance of this data, and presentation of the information to a human in an easily readable, as well as easily accessible, format. These problems were solved by the end of the research period, and a functioning system metric aggregation API was developed. The integration of software and computer command techniques allowed the API to retrieve a Node's Linux Operating System metrics such as memory information and processor information, send data over a network to be stored in a database, and present the information gathered on a web page in a web browser.

I. INTRODUCTION

A. Project Scope

The origin of the necessity for an Application Programming Interface for Linux systems stems from the need to obtain more accurate data concerning cities and certain wildlife areas. The Array of Things project developed at Argonne National Laboratory was started to accomplish this task of urban sensing paired with data collection and analysis. The project consists of a network, seen in Figure 2, of modular sensor boxes, seen in Figure 1, that are spread throughout a city such as Chicago. These sensors, also called Nodes, shown in Figure 1, are placed on light poles and are able to measure, record, and transmit data relating to the surrounding environment. Examples of some of these measurable environmental variables include climate, weather, pedestrian traffic, air quality, and noise.[1]

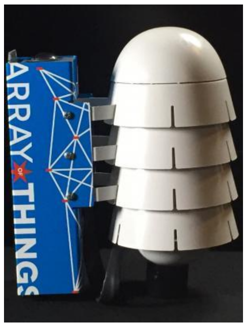


FIG. 1. A Waggle Node

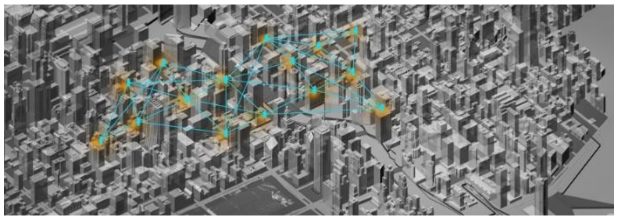


FIG. 2. An example of a network of Nodes in a city.

In order to be able to sense these environmental variables, a core structure was needed to control and run the Node while out in the field. The solution was to utilize two embedded single-board computers within the Node that would function as the “brain” of the sensor. These computers employ the Linux Operating System, a free and open source software that allows developers to control and change core functionality of the computer.[2] These single board computers, brand named ODROIDs, run Linux as their operating system and interface with the sensors and power control boards inside the Nodes. This software OS, coupled with hardware, is what allows the Nodes to have advanced functionality, such as image processing on location rather than off-site.

B. Project Goals

However, despite the extensive software functionality of Linux and advanced environmental sensing ability, the Waggle Nodes still lacked a platform for collecting data related to the computer hardware and software itself. In the event of a Node failure, a singular developer would be required to investigate and diagnose a Node individually which requires time and resources. Using Linux command line tools and knowledge of computer systems, one developer could take hours to diagnose a problem, in addition to the time needed to correct the issue. Although there are tools in place to find system metrics to diagnose problems, no comprehensive monitoring and recording of these metrics existed, and there was no easily accessible and readable aggregate list of these metrics. This paper focuses on the creation and implementation of a system software monitor checking important metrics relating to a Node at the hardware and software level and the aggregation of this Linux system data.

II. PROBLEM AND PROPOSED SOLUTION

The proposed solution to fulfill the need for a metric monitoring system was to create an Application Programming Interface (API) that could accomplish three tasks. These tasks were collection of diagnostic system data from an ODROID Linux single-board computer, transmittance of this data to a remote database for storage and analysis, and presentation of the information to a human user in an easily readable and accessible web page format. This proposed solution would improve the functionality of the Linux command line diagnostic tools for several reasons. The list of metrics related to the single-board computer would be in a central location, allowing information to become easily accessible and readable. Due to the web page format of the information, users with little experience in the field of computers could diagnose issues, thus reducing time and resources needed to find and fix problems. In a larger sense, the simplification of reduction of problems in Nodes opens the Array of Things project to focus on other areas of improvement, as well as continuing to serve the cities with important data. Thus, the proposed API solution would be a step in correcting issues with Nodes at a hardware and software level, but also issues within cities that can be solved by implementing these Nodes.

III. METHOD

A. Data Pipeline

As research was started and development began, the three goals of the project were initially grouped into one software system that could perform all of the project tasks. However, it was soon discovered that in order to accomplish the three goals of the research, the project necessitated the division of the tasks into three major sub-systems of software: data aggregation, data transmittance, and a web app. Each collection of software would control one of the project goals. These systems would work symbiotically, but also possess the ability to function separately, thus ensuring that if one subsystem were to fail, the other two would remain active. Linux Services, Bash code files that run background computer processes,[3] within the operating system would be able to run the API that contained all three subsystems, and would activate this API upon computer boot up, as shown by Box A and Box C in Figure 3. As seen in Figure 3, the flow of data would start with the data aggregation subsystem in Box B which would collect the computer metrics. The data transmittance subsystem, Box F in Figure 3, and web app subsystem, Box E in Figure 3, could then query the data aggregation subsystem for the computer metrics and use them according to their functionality. The data transmittance subsystem would send the information to be stored in a remote database, and the web app would show the information on a web page for the local user.

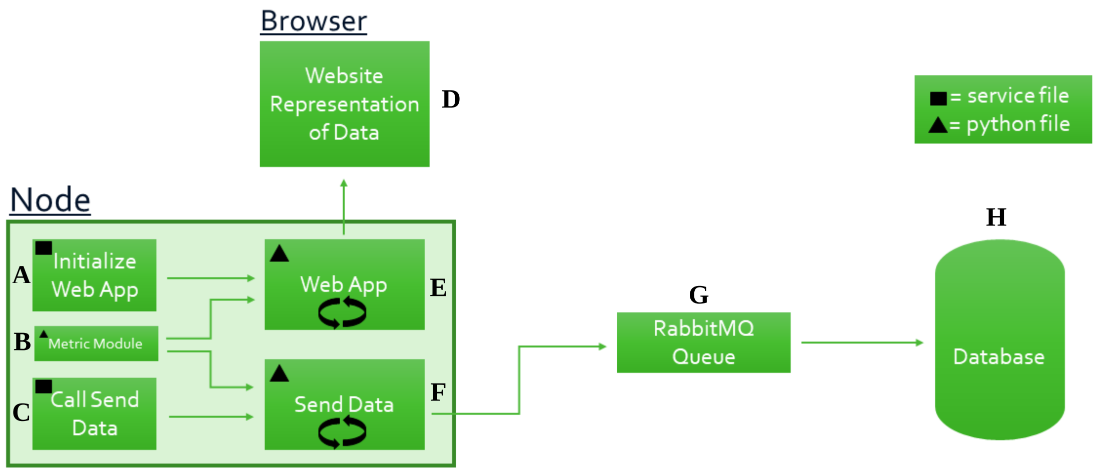


FIG. 3. A block diagram representing the data pipeline of the API.

B. Data Aggregation

As shown by Figure 4, the collection of data would begin in the metric retrieval software (Box B). This subsystem would query the operating system for various statistics and metrics regarding the single-board ODROID Linux computer. These metrics included: memory information, disk (hard drive) usage and information, processor information, uptime of the computer, connected USB devices, computer name, ID, kernel, operating system, architecture, and boot ID, running services, and uptime of services.

This subsystem contained a single file named metrics.py, seen represented by Box B in Figure 4, which as the file extension indicates, was written in the Python programming language. The file took the form of a Python module, which is a Python object with arbitrarily named attributes that one can bind, reference, and define functions, classes and variables within itself.[4] The metrics.py module contained functions that held the capability to query the computer for the aforementioned metrics. However, one function's sole purpose was to collect the data that the other functions would retrieve, and package the information in JSON format, which it would return when called by the data transmittance subsystem or the web app subsystem. In order to retrieve the metrics, metrics.py would employ two existing modules called subprocess and re (regular expression). The subprocess module could call Linux commands from the Python code, which simulated a user entering a command in to the Linux command line. This module allowed the metrics.py module to retrieve system information about the computer, such as processor information and memory information. In addition, the re module helped to parse the data that was retrieved by using the subprocess module. Using regular expressions on the retrieved data strings allowed patterns in the data to be found for processing and detection of specific, vital parts of the retrieved information.

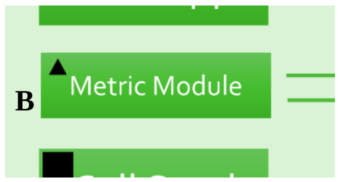


FIG. 4. A block diagram representing the data aggregation subsystem.

C. Data Transmittance

Since the data aggregation subsystem could be queried for information at any moment in time, it would only collect data when it was called by the other two subsystems. This process, in addition to creation of the data aggregation subsystem as a Python module, made the integration of data aggregation into data transmittance simple. The Python file sendData.py, as represented by Box F in Figure 5, would be able to import the metrics.py module represented by Box B in Figure 5 into its own code, then query a single function that would return a Python dictionary of the retrieved metrics in JSON format.

This data transmittance subsystem contains several pieces of software that allow the retrieved metrics to be stored in a database. This subsystem's dataflow begins when the service shown by Box C in Figure 5 begins the sendData.py file at computer boot up. The sendData.py file (Box F, Figure 5) would then run continuously, communicating and sending the retrieved metrics at a user-specified time interval over the network to a holding service called RabbitMQ, shown by Box G in Figure 5. This holding service would then retain the data until software written by another project member would retrieve the information, delete it from the queue, and store it in the remote database represented by Box H in Figure 5.

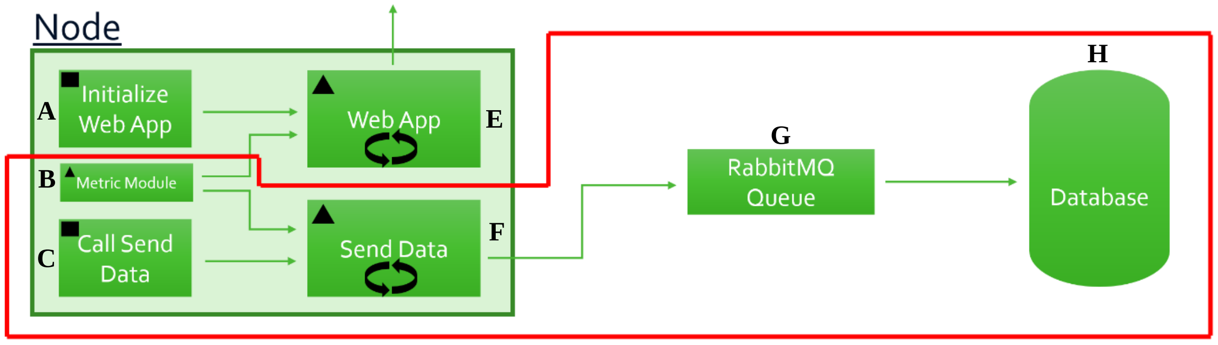


FIG. 5. A block diagram representing the flow of data through the data transmittance subsystem.

D. Web App

The subsystem that encompassed the web view of the aggregate data also consisted of multiple pieces of software. As seen in Figure 6, the service represented by Box A would enable the web application, represented by Box E, on computer boot up. The web application waggleApp.py was written in the Python programming language, using a micro-framework called Flask. When started, the Python file would begin a server on the localhost network, i.e., itself, and would then serve web pages containing metric data from the data aggregation subsystem, Box B in Figure 6, to a user connected to the network. HTML and CSS files were called from within waggleApp.py to render the web pages, and JavaScript was embedded within the HTML. Using a web browser on the local network, shown by Box D in Figure 6, a user could view the metrics gathered by the data aggregation subsystem on the served web pages from the web application waggleApp.py. An example web page that displays general information about the single-board computer in a Node is shown in Figure 7.

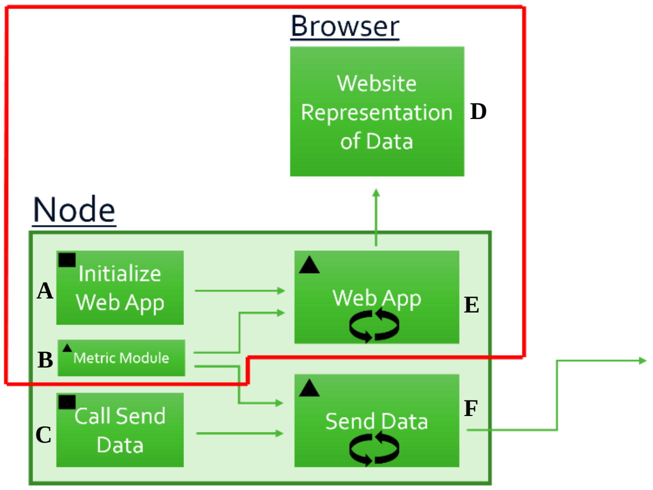
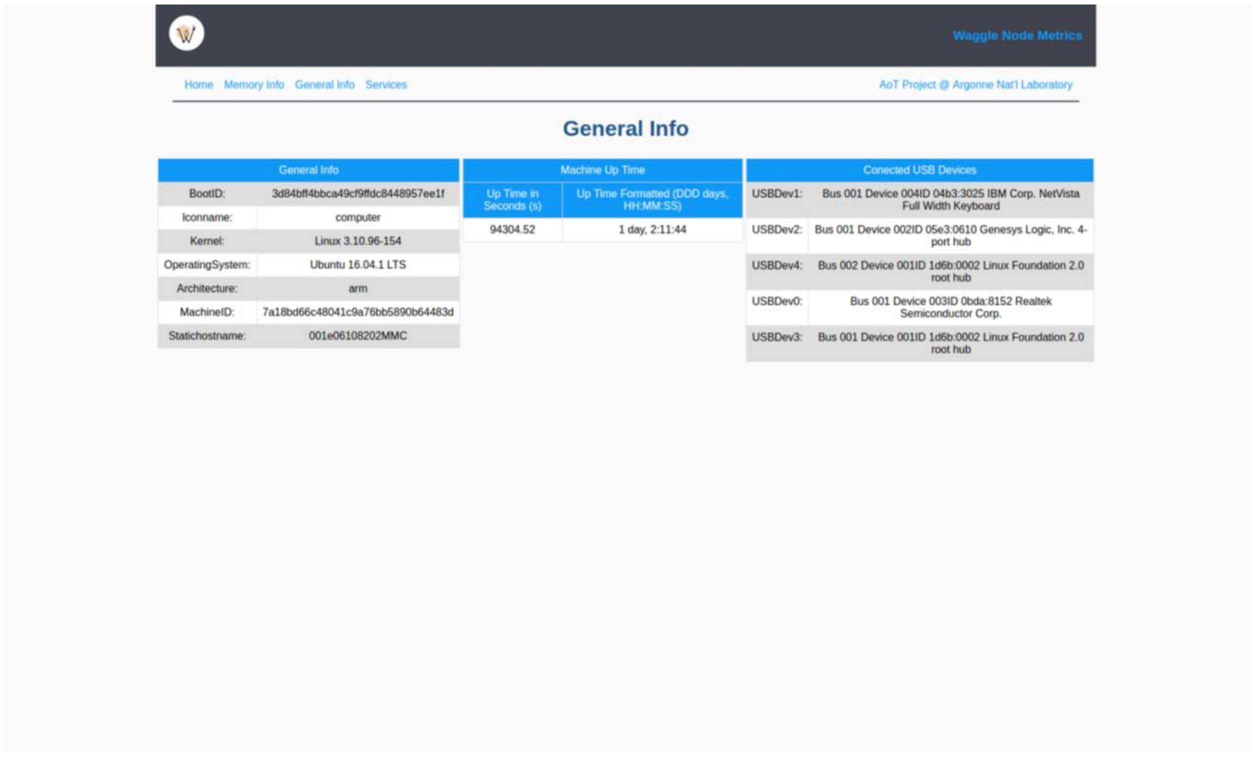


FIG. 6. A block diagram representing the flow of data through the web app subsystem.



General Info

General Info	
BootID:	3d84bff4bbca49cf9ffdc8448957ee1f
Iconname:	computer
Kernel:	Linux 3.10.96-154
OperatingSystem:	Ubuntu 16.04.1 LTS
Architecture:	arm
MachineID:	7a18bd66c48041c9a76bb5890b64483d
Statichostname:	001e06108202MMC

Machine Up Time	
Up Time in Seconds (s)	Up Time Formatted (DDD days, HH:MM:SS)
94304.52	1 day, 2:11:44

Conected USB Devices	
USBDev1:	Bus 001 Device 004ID 04b3:3025 IBM Corp. NetVista Full Width Keyboard
USBDev2:	Bus 001 Device 002ID 05e3:0610 Genesys Logic, Inc. 4-port hub
USBDev4:	Bus 002 Device 001ID 1d6b:0002 Linux Foundation 2.0 root hub
USBDev0:	Bus 001 Device 003ID 0bda:8152 Realtek Semiconductor Corp.
USBDev3:	Bus 001 Device 001ID 1d6b:0002 Linux Foundation 2.0 root hub

FIG. 7. An example web page of the web app subsystem displaying general Node information.

IV. Testing

Throughout the creation of the API, the three subsystems were tested on the single-board ODROID computers to ensure that the software being written and the implemented hardware were compatible. The testing procedure followed a three-step process: run, check, and reconfigure. After important code changes to the API's three subsystems, this testing process was followed. The new, updated subsystems would be run simultaneously on a desktop test station, and then again on an ODROID single-board computer. Running the new code on an ODROID computer simulated the code running on a Waggle Node since the ODROID is inside of a Node running the software. The new, running code would be inspected for errors or hardware compatibility issues. If no issues arose, new code would continue to be added to the API, and the testing process would restart. If issues were found, however, the testing would enter the reconfigure stage of testing where the errors were sought and corrected within the code, and the testing process would begin again. This constant, rigorous method of updating code and testing allowed the API to be constantly analyzed for errors or problems and corrected, and led to a successfully functioning API that accomplished the goals of the research project.

V. CONCLUSION

At the end of the 10-week research project period, the proposed API solution to the need for creation and implementation of a system software monitor checking important metrics relating to a Node at the hardware and software level and the aggregation of this Linux system data was successfully accomplished. The implemented code of the three subsystems of the API were able to run on a Node, and each succeeded in completing the task that it had been designed to manage. The data aggregation subsystem retrieved metrics from the operating system and made the information readily available to the other two subsystems. The data transmittance subsystem functioned correctly by sending the data retrieved from the data aggregation subsystem to a queue over the network, and thus to the remote database for storage. The web app served a local network web page view of the information from the data aggregation subsystem for testing or error diagnostics by a developer or end user.

The implications of this research project are both small-scale and large-scale. The constructed API simplifies the problem diagnostic process for developers and enables problems to be diagnosed and fixed more quickly. In addition, it creates an aggregated location for the metrics it gathers to be viewed. Due to the addition of the API to Waggle Nodes, the over-arching Array of Things project goal of measuring and recording city data can be reached much quicker. The development of this API will enhance the capabilities of Node developers, thus allowing the project to move forward at a faster pace, and thereby hastening the positive impact on the served cities.

VI. ACKNOWLEDGMENTS

I would like to thank the entire Waggle and Array of Things Teams for their tutelage, guidance, and instruction over the past two months of the research project. Thank you to Peter Beckman and Rajesh Sankaran for their direction and leadership of the project. Thank you to Peter Lane and William Catino for their help with hardware and software knowledge. Special thanks to Sean Shahkarami for his mentorship and dedication in offering solutions, guidance, and computer knowledge from which I learned immensely. I would also like to thank the Argonne National Laboratory, Department of Energy and National Science Foundation for the opportunity to work in this position and gain experience on a technical project at a research lab.

VII. REFERENCES

[1]R. Mitchum, "news.uchicago.edu/article/2016/08/29/chicago-becomes-first-city-launch-array-things"

[2]J. Wallen, "www.techrepublic.com/article/getting-up-to-speed-with-the-linux-desktop-operating-system/"

[3]The Linux Foundation, "www.linux.com/news/introduction-services-runlevels-and-rcd-scripts"

[4]Python Software Foundation, "https://docs.python.org/2/tutorial/modules.html"