## Embedded Middleware for Distributed Network of Raspberry Pi Devices

Project Report for TCSS 702

University of Washington, Tacoma

By

Siddharth Bhave

[svbhave@uw.edu](mailto:svbhave@uw.edu)

Faculty Advisor

Dr. Jie Sheng – University of Washington, Tacoma, WA, USA

Industry Advisor

Dr. Matthew Tolentino – Intel Corporation, WA, USA

Table of Contents

1. **Introduction3**
2. **Related Work5**
3. **Methodology6**

3.1 Initial Implementation6

3.2 Middleware Architecture and Data Flow7

3.3 Data Flow Division8

3.3.1 Streaming data from the sensors8

3.3.2 Providing data to one node9

**4. Experiments and Results10**

**5. Conclusion and Future Work13**

**6. Acknowledgements14**

**7. References14**

**Abstract**

Applications making use of embedded systems are anticipated to become extremely important as we advance towards realizing the vision of “Internet of Things”; with smart devices (Raspberry Pi), mobile phones, and compute anywhere paradigm. Principles of distributed systems play a pivotal role in this paradigm. For example, we implement a distributed network of low powered devices to accomplish various tasks autonomously driven by a distributed embedded system architecture where each of the devices can work on independent local data, (which is device specific) to perform a similar compute task simultaneously to achieve a common goal that is broad and systemic. This collaborative problem solving in the embedded setting is similar in concept to the big data paradigm now commonly proposed for commodity hardware and large databases. As the embedded devices go on to become more capable and powerful the two concepts will combine, but today they are worlds apart forming the motivation for our research. As put through in this project, a middleware layer is required to make the devices work collaboratively on local data within a network of embedded devices (such as Raspberry Pi devices). Currently no such middleware exists. The objective of this project was to design, devise and test a middleware layer which splits, distributes, computes and merges the compute task to accomplish a shared compute goal while performing the operations locally in a ‘shared nothing’ architecture.

1. **Introduction**

Raspberry Pi [1] is a very popular embedded device which is in use for many applications since last year. Raspberry Pi has the processing power comparable to 300 MHz Pentium 2 Intel architecture and 512 MB RAM [2]. It can act as a low powered wimpy node [3] in a distributed network, and a collection of such wimpy nodes can be used as a collaborative swarm to accomplish a certain task that can be distributable over a network.

Previously the data collected from an embedded device would be transferred to a local server and then to a bigger server where data processing is done. The results are then sent back as a feedback to the actuators. This is a time consuming process. Fast feedback for one system was possible but feedback for a wide network system of embedded devices the time efficiency was a big hurdle. The Raspberry Pi is a special kind of embedded device. It does not just provide us the General Purpose Input Output (GPIO) peripheral interfacing but it has a complete Linux package core. So it can act as a computing machine as well as an embedded device. This is a very rare feature that was unavailable until recently. This project tries to exploit this feature of the Raspberry Pi devices to bring the data and computing in embedded systems really close to each other.

Distributed processing allows a user to compute data task collaboratively on different nodes simultaneously. So instead of using powerful super computers we can use machines with lower specifications to process the data faster by avoiding network latency costs to centralize the data. Though the processors of these machines may be weak, they are cheap, low powered energy efficient and have software capabilities similar to a powerful computer. Distributed computing believes in “scaling out” and not “scaling up”. Instead of providing additional resources like CPU cores, RAM, memory to a single machine, multiple similar machines are used to achieve the same task. A *Microsoft Developers Network journal* article [4] discusses the applications of distributed embedded systems in the fields of environment monitoring, aerospace exploration, automation, mobile communication, health care, etc. in the future.

Big Data tries to address similar issues with a similar approach to distributed computing on a very large scale [5]. Big Data tools are much faster and efficient for a scenario with large data processing. Since recent years, in embedded systems the data from reactive sensors, visual sensors, monitoring devices and control parameters of the devices needs to be collected, transmitted, processed, analyzed and the meaningful extracts should be fed back to actuators and stored. The Raspberry Pi devices are capable enough to work with high quality sensor data. For example, 1080p video quality is supported by the graphical processing unit mounted on it. Current infrastructure is not efficient enough to support a network of such devices. So this is a new step in towards the intersection of Big Data and Embedded Systems and tap advantages of both these fields.

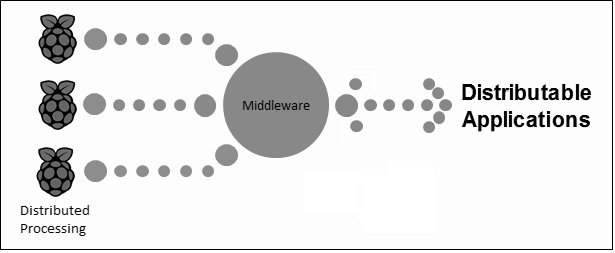


Figure 1: The Middleware Layer Arrangement

Implementing the concepts of Big Data over a distributed Raspberry Pi network can be done by designing a middleware for each Pi so all the tasks can be performed seamlessly. The middleware is the infrastructure that bridges the chip level applications and the software level distribution among the devices. Figure 1 shows a general block diagram of the middleware. The middleware will be responsible for scheduling the tasks, assigning the tasks to the nodes, transferring the data and communication among the nodes. Depending on the application the user will have to put minimal effort to establish the interconnection between the nodes.

Now consider these two scenarios: A) A treasure hunt game is organized where the participants have to find a particular thing, say a Red Rose. All the participants have a Smartphone and when they find a red rose they are to take a picture of it. One of the Smartphone is a master node. Now once say 3 red roses are found the hunt should end. There is an algorithm running on the Smartphone which segments the image and checks it to verify if the picture is a rose. When it is verified the phone node sends the message to the master node. The master node waits till the count is 3 and once that count is reached it passes a message to each phone to conclude the hunt and declare the winners. B) Suppose 3 Raspberry Pi devices are strapped on to a quad-rotor robot, which has a video camera input stream. The quad-rotor is trying to avoid obstacles by performing computer vision algorithms on the video input. This stream can be provided to each of the Raspberry Pi devices at an interval of 10 seconds. These devices will compute the algorithms on the devices and send back the signal to the quad-rotor. Both these scenarios require coordination and communication between the device nodes. A middleware facilitates this kind of cooperative computing and communication among the device nodes. In this project we have enabled the communication between different Raspberry Pi devices by devising a middleware layer to implement both the above described scenarios.

1. **Related Work**

In the past, efforts have been made to make a middleware application for embedded systems. The very first effort to explain the importance of the distributed embedded systems was in 2002 [6]. In this paper the importance of cooperative computing model was described for a distributed network of devices. This was followed by D.C. Schmidt et al. [7] explaining how useful the middleware layer is that splits, schedules and assigns tasks to different nodes. As the microcontrollers became more powerful they became more powerful as the system on chip features increased while the power required remained the same. This project’s design aspects will be drawn from the book *“Real-Time Systems: Design Principles for Distributed Embedded applications”* by H. Kopetz, which explains the system based concepts for distributed embedded applications [8].

Various kinds of middleware architectures were proposed by W. Wolf et al. giving us a guide to selecting or developing architectures for distributed system as per need [9]. A distributed network of embedded devices was applied as a low-powered web server in [10]. Mohaghegh et al. compare the existing system and the network of the microcontrollers to see the power consumption and performance. By using low powered architecture of embedded devices we can make a distributed network and split the workload in parallel computation. Most recently a system called *iLAND* [11] was developed to address the problems of deploying complex embedded systems into critical environmental conditions like in a machine industry. Small changes in conditions need to be recorded and announced across the network of embedded devices. This problem is addressed by developing a middleware to control all the devices and make them coordinate with each other. This middleware helps the devices to be reconfigured in real time. This kind of system is useful especially in machine equipment based industries.

As the amount of applications for distributed embedded systems increase the need to big data management increases. With the advent of big data tools the efficient management of the streamed sensor data was possible. Google’s MapReduce concept from 2004 is widely used by many industries [12]. The Hadoop distributed file system (HDFS) facilitates a distributed storage system for distributed nodes. One node is the master node called the ‘name node’. The HDFS architecture is the implementation of Map and Reduce over different nodes in a distributed system. A task is split into many parts and the Map and Reduce functions are computed over the parts on different nodes [13]. Recently many applications were developed with respect to distributed embedded systems using these tools. *Misco* is a MapReduce framework for mobile systems that addresses the problem of communication over a large number of devices in a network. MapReduce framework is used to handle the difficulties of communication and data distribution [14]. Another application of big data on embedded systems can be seen in a broadband network of embedded Linux based set top box devices. By implementing the Hadoop MapReduce framework on such devices the system can be made scalable and energy efficient for large scale data intensive applications. This system can be used to leverage the additional computation resources available in the network of set top box devices to make high quality transmission [15].

1. **Methodology**

This section explains the technical aspects of the project. First we discuss the initial implementation of the project and some specifications of the Raspberry Pi device. Then we will describe the in depth block diagram, features of the middleware layer and communication features between the nodes.

**3.1. Initial Implementation:**

This section discusses the hardware and software specifications of the project. The Raspberry Pi needs 5V of supply from a micro USB cable. This model B devices may need up to 700mA – 1200mA current. Many phone chargers as well as HDMI/USB to micro USB cables meet this requirement. Another option is supplying the Pi from a battery but the battery charging is a complexity in it. I am planning to use 3 Raspberry Pi devices so I will be using USB to micro USB power from the USB ports of a laptop. But if more devices are added on the network then I can use a wall socket to power a USB board where all the devices will be connected via USB to micro USB cable.

The Raspberry Pi is an ARM11, Linux based architecture. It has a slot for SD card where an SD card pre-loaded with a Linux Operating System (OS) can be inserted. The OS can be dumped on an SD card and loaded in the Pi. The recommended size for the device is 4GB but I will be using 32GB SD cards for large amounts of data. The Raspberry Pi community has plenty of support for different Linux OS versions except Ubuntu [17]. Currently the ARM11 that is the ARMv6 is not compatible with the Ubuntu version of Linux. An OS called Raspbian Wheezy which is a modified version of the Debian version of Linux can is used on the devices. It provides native compiling capabilities for languages like C,C++ and Python. Raspbian also provides driver compatibility with the General Purpose Input Output (GPIO) pins and Camera Serial Interface (CSI) for interfacing different sensors and actuators useful in embedded applications.

The Raspberry Pi device has two USB ports and an Ethernet port. We can use a wireless dongle or an Ethernet cable to provide internet connection to the device. But Ethernet has a much faster latency. Once the OS is dumped on the SD card and put on the device, we will need to access the Pi over SSH protocol easily. First the Pi has to be connected to a monitor screen via HDMI and to a keyboard and a mouse via USB ports. Once the OS is booted up, the network in which the Pi will connect has to be configured so that it automatically picks up that network once it has network access. So we have a simple python script that runs during initialization and sends the IP over email. If you have the IP address of the device then the SSH protocol (from a Terminal of Mac OS/Linux OS or putty on Windows OS) can be used to remotely access the Pi. So the requirement of a monitor screen a keyboard or a mouse will no longer be a necessity. Once we can access the Pi remotely we can program the devices from a single machine simultaneously.

We have used a MySQL webserver which is used as user interface (UI), where all the network information will be provided. It is used to monitor the status of each device. All the devices are connected to this webserver before staring a job and using the database enables them to retrieve the IP addresses of all the other devices that will participate in the process. This way there is no need to have a manually modified IP address lookup table.

**3.2. Middleware Architecture and Data flow:**

The middleware layer devised in this project enables easy communication between the nodes by minimum efforts by the user. It provides software applications for a system that are not readily available from the operating system. All the Raspberry Pi devices will act as nodes which are in a network. The middleware and the network protocols are the middle layer between the operating system and the application of the nodes. The middleware takes care of the operations specific to the embedded application in case of the Raspberry Pi device network.

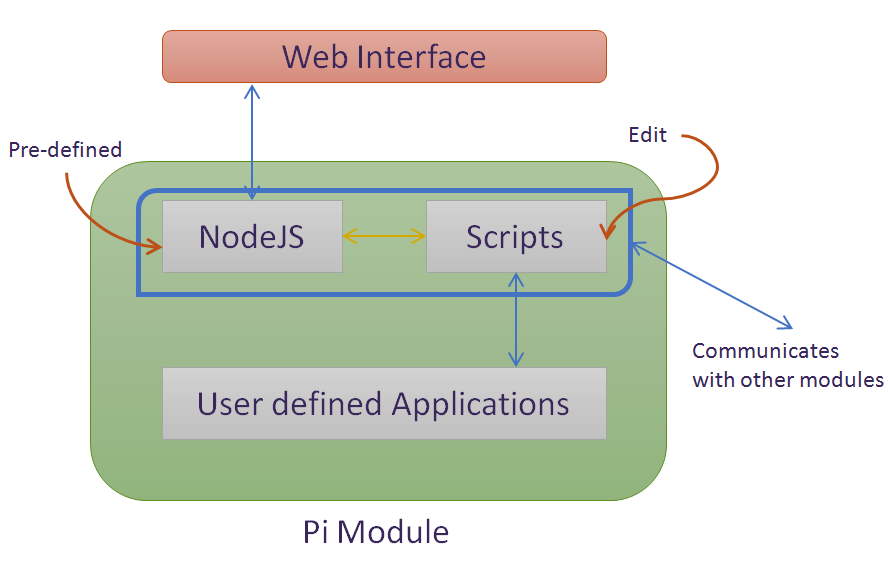


Figure 2: Middleware Layer Architecture

The blue box in the Pi Module in Figure 2 makes the middleware layer. It is used as a connection agent between the application, the webserver and other devices with similar software architecture. It consists of pre-defined Node.js connection scripts and bash/python scripts that enable communication over the network.

To initialize the system there are two necessary steps: making the IP of the device public and connecting it to the MySQL server. For the first script we provide a public DNS name to the Pi so that it can be seen over any network and the server can talk to it. We provided a DNS name from a free service and update the IP address on that DNS. For the second step we update the MySQL database server setup containing tables having the details of the Pi like the user, registering time, IP address and MAC address. There is a PHP script on the server that is called by the Pi to input the data in the database. We run a Node.js server that collects the necessary information on the Pi and calls the PHP script on the server. The PHP script decodes the Node.js call that is a JSON array and adds the data in the database. Now all the active devices that will be involved in the network are registered on the server. You can monitor the devices on the server.

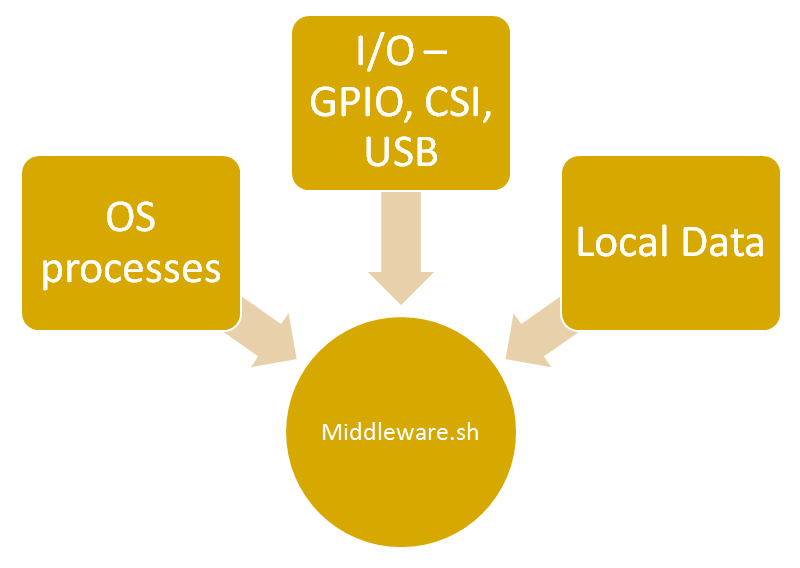


Figure 3: Middleware Layer Data Flow

At this point each device is unaware of the other active device in its vicinity. So just before beginning a job, we run another Node.js script that extracts the IP addresses of the other active devices in the vicinity and stores them in a text file. Figure 3 shows how the middleware layer script access data from the OS, the system I/O and local data. Once this is done the middleware.sh script begins the application code taken in as an argument. It connects with other devices by using the IP addresses in the text file through Python TCP server and client. The server starts performing a task. The other devices are listening through a Python TCP client server. And once they get a particular message or data they will perform different tasks defined by the user. In some applications it is necessary to have two threads running in parallel, one which is the master and one is the listener. Next we look at the applications for which this middleware will be useful.

**3.3. Application Grouping Based on Data Flow:**

The project’s applications can be divided into two different sections based on flow of the data. One is streaming data from the sensors and the other is supplying data to one node and distributing it.

* + 1. **Streaming data from the sensors:** In this type of applications each device has a sensor interfaced to it and all the devices are working to achieve the same goal. So we can see them as a distributed network of similar nodes. The goal may be to find or compute something in a data stream which is easily available to the devices. All the devices will be working on same process and when one of the devices finds something or finishes computing whatever it was computing then it lets the other devices know that the task is complete. So after the message, all the devices can move on to a different process. The example of the treasure hunt game described in section 1 is a good example for this. Figure 4 below explains the block diagram for this scenario.

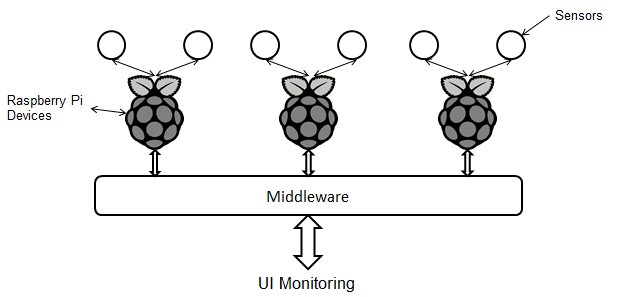


Figure 4: Streaming data from the sensors

Another scenario to exemplify this part can be as follows: A camera is interfaced with a Raspberry Pi and the system is placed at your doorstep. It is detecting if you have any intruders. The Pi is device is running a human figure detection algorithm called hog detection. So as soon as it detects the contour of a human body it can notify a Pi interfaced with a speaker placed in the bedroom to sound an alarm and notify the owner about the intruder. Another example of the usage of this system is in case of a fire at sensitive areas. If one Pi detects fire then it can notify other nodes that are interfaced with sprinklers to start the sprinklers and pacify the fire.

* + 1. **Providing data to one node:** In this type of applications one node can work as a master and receive input data. This data is then distributed among various devices to start computing. This data can be then combined together to give the results. The middleware layer on each device would manage the scheduling and distribution of the data over the network. Each node should receive connection messages before starting the computation. Scenarios for such application would be mergesort, word search, word count, etc. These examples have a large amount of data in the form of a list or array which can be distributed over the network to compute it separately on the Raspberry Pi devices. Figure 5 below explains the block diagram for this scenario.

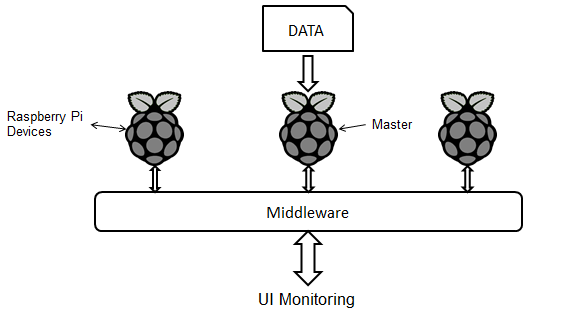


Figure 5: Providing data to one node

This application has a major drawback that the data is not present on the nodes. It is very expensive to transfer the data from one node to all the other nodes due to the network latency and bandwidth bottleneck. This kind of applications shall be focused to use the device network for batch processing rather than real time processing. Hadoop has this kind of batch processing structure. By doing this we will be exploiting the novel use of cheap, low powered devices for big data analysis. The overall (analytics/total cost of power) factor is an advantage in this case. Consider this case: in a forest environment monitoring network of embedded device nodes, sensors are gathering data from each device. The gathered sensor data is computed on each node separately and the summary of the computed data is stored in a master node. This data needs to be processed using some data analytics/data mining algorithms to know the temperature and humidity trends, the risk of forest fire, etc. Sufficient data collection over a period of time is required. To do this the master node will split the data and send to all the nodes to process them separately. After this again the results are collected back in the master node. As a result this becomes a stand-alone system that collects, computes and analyzes the data.

1. **Experiments and Results**

After the middleware scripts were written, we proceed with testing applications for the scenarios described in section 3.3. We will provide application code as an argument in the middleware script and tweak the scripts to carry on with the communication.

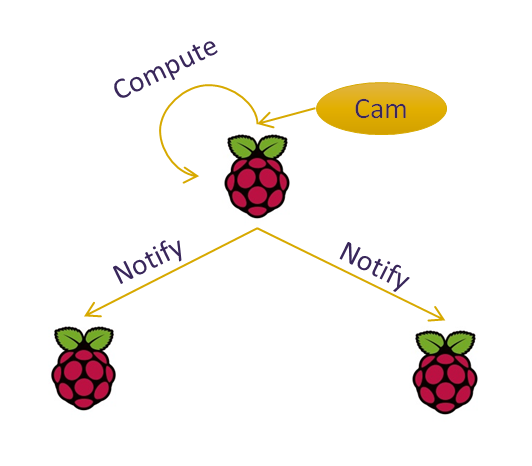


Figure 6: Streaming data from sensors

For the first application scenario, streaming data from sensors, we used OpenCV[18] face detection algorithm by interfacing the Raspberry Pi camera module with one of the devices while other devices listened for a notification. Once the Pi gets a notification from the camera interfaced Pi that a face was detected, it does some task locally like glow a LED. Figure 6 explains this process where one Pi computes the data locally and notifies the other devices.

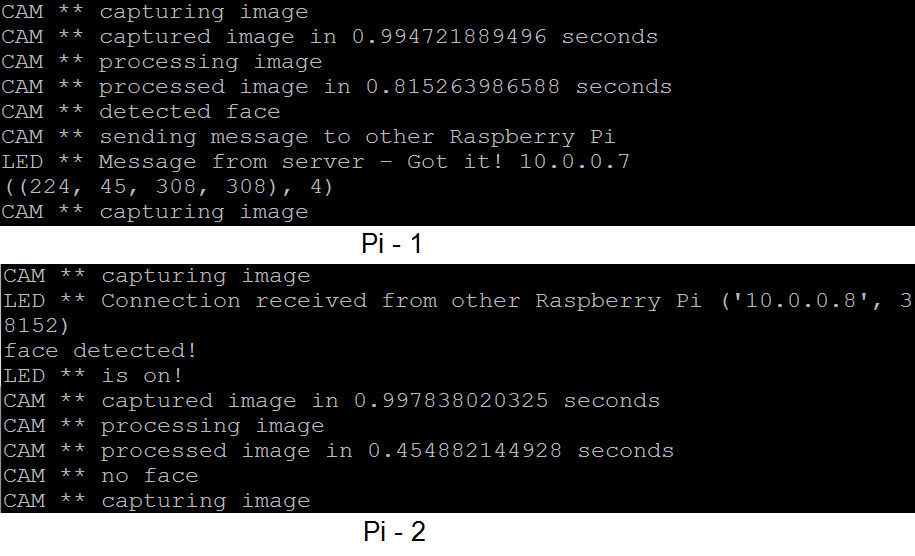


Figure 7: Two threads

Next, we ran the same example as two threads on each Pi. Each device has two threads running at a time, one thread listens for notifications and the other threads uses the interfaced camera and runs the OpenCV algorithm. So whenever a Pi detects a face it notifies the other devices that are also trying to detect faces. Figure 7 shows the two threads running in parallel on two nodes. The thread that is listening prints out "LED \*\*" and the thread that is capturing and processing the image prints out "CAM \*\*". As seen in the figure it takes about a second to capture an image and less than a second to process the image.

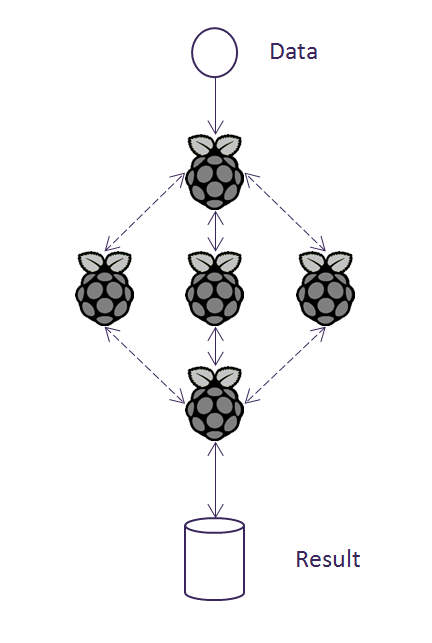


Figure 8: Providing data to one node

In the second application scenario data is provided to a single node and the data is distributed among the devices. The data chunks are processed individually after which the data is aggregated back to the master node as shown in figure 8. We carried out experiments with two applications, merge sort and word count. For merge sort we randomly create an array of 100000 integers and shuffle it. If only one node participates in the process then it takes about 20 seconds to sort and merge the array. If there are two participants then the time decreases to around 17 seconds. Here because the array is large the latency of sending the data across the nodes is compensated. If the data were smaller in size then it would have been cheaper to compute on a single node rather than sending it across the nodes. Similarly we compute word count from a text file containing around 200 lines using two nodes and it takes about 11 seconds to process.

We measured how much memory and CPU cycles are used by each process in the Raspbian OS by using the Linux "top" command. Table 1 shows the CPU usage and memory usage percentages. The merge sort master uses 35% CPU when it is waiting for results from the clients and 95% when it is processing the data while the client usage varies from 70% to 90%. The memory is used the most when a camera is involved. The image capturing and processing uses the most memory about 8% to 9%.

|  |  |  |
| --- | --- | --- |
| **Application** | **CPU Usage** | **Memory Usage** |
| Merge – Sort Master | 35 – 95 % | 2 % |
| Merge – Sort Client | 70 – 90 % | 2 – 3 % |
| Face Detector | 35 – 50 % | 8.5 % |
| Listener | 0 % | 1.2 % |
| Multithreaded | 35 – 50 % | 8 – 9 % |

Table 1: Performance of different applications

1. **Conclusion and Future Work**

The motivation for this project is - low cost and energy efficient computation. We exploit the memory and I/O interface of the Raspberry Pi to bring the most out of it via enabling distributing computing. We can see that the middleware devised here is compatible with a lot of platforms. We use different technologies like MySQL on the server side and Node.js, Python, C and bash scripting on the Raspberry Pi. The main base of the middleware is Python. The advantages of using Python are as follows - it compiles natively on the Pi, Python's compatibility with API for different languages and the easy API of threading and TCP communication. Both the scenarios explained in section 3 are widely applicable in the embedded world. There are many fields where this system is useful like Smart Homes, Environmental Systems, Automation and Avionics or for just low power computation.

In the future we would also like to test the system with various industrial benchmarks for distributed computing. It would be an ideal step in the future to compare the middleware performance and see how it stands in comparison to traditional systems. We would also like to modify the middleware to be able to decide when to use other nodes and when not to. It should be able to determine if the latency of distributing the data to other nodes would be advantageous or not.

Currently we were using three Raspberry Pi devices but in the future we would like to work with more. It would be interesting to compare the performance of systems with different varying number of devices. If the numbers of devices are large then providing power will be a bit difficult. So a power board can be used to provide power to all the devices. This power board will be powered by 110V wall output and divided into many USB ports of 5V where the devices can stick in. Another goal will be to explore other algorithms for implementing the communication in the network. Also analyze different techniques of networking among the devices.

All the code and documentation can be found in the Github repository - https://github.com/Sid-1020/raspberry-pi-middleware

1. **Acknowledgments**

I would like to thank my advisor Professor Jie Sheng to help me to come up with a proper project definition and implementing it. Our collaboration for this project was fruitful in the process of achieving the goals of the project. I would like to thank Professor Ankur Teredesai for discussing the possible project ideas and helping me to choose this project. I would also like to thank Professor Daniel Zimmerman and Professor Matthew Tolentino for giving valuable suggestions and comments while discussing this idea. And finally I would like to thank Professor Ankur Teredesai and Professor Robert Friedman for guidance on academic writing.

1. **References**

[1] (05/31/13). *Raspberry Pi*. Available: <http://en.wikipedia.org/wiki/Raspberry_Pi>.

[2] (06/02/13). *Raspberry Pi FAQs*. Available: <http://www.raspberrypi.org/faqs>.

[3] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: a fast array of wimpy nodes. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09). ACM, New York, NY, USA, 1-14. DOI=10.1145/1629575.1629577 <http://doi.acm.org/10.1145/1629575.1629577>.

[4] (06/02/13). *On Distributed Embedded Systems,*[*http://msdn.microsoft.com/en-us/library/dd129911.aspx*](http://msdn.microsoft.com/en-us/library/dd129911.aspx).

[5] P. Zikopoulos, C. Eaton, D. Deroos, T. Deutsch and G. Lapis, "Harness the Power of Big Data, The IBM Big Data Platform," pp. 1-34, 2013.

[6] C. Borcea, D. Iyer, P. Kang, A. Saxena and L. Iftode. Cooperative computing for distributed embedded systems. Presented at Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on. 2002, . DOI: 10.1109/ICDCS.2002.1022260.

[7] D. C. Schmidt. Middleware for real-time and embedded systems. *Commun ACM 45(6),* pp. 43-48. 2002. Available: <http://doi.acm.org/10.1145/508448.508472>. DOI: 10.1145/508448.508472.

[8] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications.* 2011.

[9] W. Wolf. Middleware architectures for distributed embedded systems. Presented at Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on. 2008, . DOI: 10.1109/ISORC.2008.31.

[10] N. Mohaghegh and M. Aboelaze. Using low-power embedded microcontrollers as web servers. Presented at Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on. 2012, . DOI: 10.1109/ISPA.2012.137.

[11] M. Garcia Valls, I. R. Lopez and L. F. Villar. iLAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. *Industrial Informatics, IEEE Transactions on 9(1),* pp. 228-236. 2013. . DOI: 10.1109/TII.2012.2198662.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun ACM 51(1),* pp. 107-113. 2008. Available: <http://doi.acm.org/10.1145/1327452.1327492>. DOI: 10.1145/1327452.1327492.

[13] K. Shvachko, H. Kuang, S. Radia and R. Chansler. The hadoop distributed file system. Presented at Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). 2010, Available: <http://dx.doi.org/10.1109/MSST.2010.5496972>. DOI: 10.1109/MSST.2010.5496972.

[14] YoungHoon Jung, R. Neill and L. P. Carloni. A broadband embedded computing system for MapReduce utilizing hadoop. Presented at Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on. 2012, . DOI: 10.1109/CloudCom.2012.6427483.

[15] A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen and V. H. Tuulos. Misco: A MapReduce framework for mobile systems. Presented at Proceedings of the 3rd International Conference on Pervasive Technologies Related to Assistive Environments. 2010, Available: <http://doi.acm.org/10.1145/1839294.1839332>. DOI: 10.1145/1839294.1839332.

[16] (06/02/13). *Distributed data processing with Hadoop*. Available: <http://www.ibm.com/developerworks/java/library/l-hadoop-3/index.html>.

[17] (06/02/2013). *Raspberry Pi Community Forum*. Available: <http://www.raspberrypi.org/phpBB3/>.

[18] (03/15/2014). OpenCV. Available: http://opencv.org/