## Project 4: Dynamically switch/provision clusters on Academic Cloud

## By Group 8 members - Nikhil Kulkarni & Vivek Kumar Singh

**Objective**: To implement a resource monitoring system that monitors and visualizes utilization in distributed set of nodes using the message broker middleware. This resource monitoring tool will be monitoring the overall average CPU usage and Memory usage and also the average CPU and memory usage of the MPI PageRank program written by us in Project #1.

**Introduction**: Resource monitoring is a crucial part of all distributed systems. In large scale computing environments it is important to understand the resource utilization and system behavior to manage the resources efficiently and detect failures and make the distributed system more robust and stable. Our aim is to monitor the CPU and Memory utilization of various nodes in the distributed system. The information from the various nodes is collected at the message broker and aggregated to display the overall system resource utilization. We will also be studying the resources utilized by a large job like the MPI PageRank program when deployed in a distributed environment like the Baremetal or Virtual clusters.

**Technical details**: The main pillars of this monitoring system are:

1. Monitoring Daemon.
2. Message broker Middleware.
3. Information display Front-End or the UI.
4. Details of PBS script.

We will walk through the functionality of all these sections and also understand how we implemented them in our project in this section.

1. **Monitoring Daemon**: The monitoring daemon basically does the core functionality of recording the Cpu and memory utilization from the system. How this really works is that when we run the daemon on a particular machine it records the system resource utilization and sends it to the message broker as a message. The message broker then further handles and uses the message further. We have implemented the monitoring daemon using the Sigar API. We have created a class called SystemsInformation which captures the system information and send it to the message broker. The SystemsInformation class uses the following functions of the Sigar API namely getCpuPerc(), getMem(), getUsedPercent(), getCombined(). To capture the CPU and Memory utilization of the MPI PageRank program we are using the functions fetchProcessCpuInfo() and fetchProcessMemInfo(). Let us understand what these four functions do in detail.
2. getCpuPerc(): This function returns a mapping of who is using what percentage of the CPU. We are particularly using the getCombined() function of the CpuPerc class to get the amount of Cpu resources used by the user.
3. getMem(): This function gives a mapping of the overall usage of the system memory. We are using the getUsedPercent() function to only get the total used memory in percentage.
4. fetchProcessCpuInfo(): This function fetches the CPU utilization of the MPI PageRank program. This function extracts the pid of the PageRank program through the PTQL(Process Table Query Language). [7]Hyperic SIGAR provides a mechanism to identify processes called Process Table Query Language. All operating systems assign a unique id (PID) to each running process. However, the PID is a random number that may also change at any point in time when a process is restarted. PTQL uses process attributes that will persist over time to identify a process. After the PID is fetched, it is passed as an argument to the sigar functions procCPU.getPercent() which gives the CPU usage of that particular process that is out MPI program.
5. fetchProcessMemInfo(): This function fetches the Memory utilization of the MPI PageRank program. This function also extracts the pid of the PageRank program through the PTQL. It then passes the pid as an argument to the sigar function getProcMem().getResident(). The getResident() function provides the resident memory utilized by the MPJ program. This value is then divided by the value obtained by sigar.getMem().getTotal() and multiplies it by 100 to find the percentage memory usage.

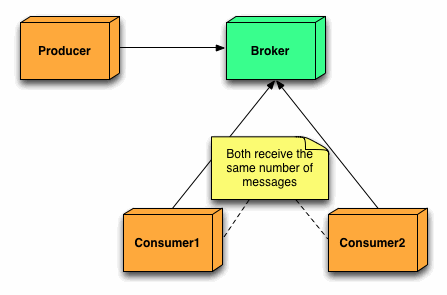
The daemon starts the timer for sending the information at only integer values of every second. For example, if the daemon starts at time 1335218774030 milliseconds. Then the timer would be started at 1335218774000 milliseconds from the epoch. This particularly helps when the two daemons start out of synch and we want the timers to send the messages in synchrony. Again one potential limitation is that, this technique relies on the times of the two nodes to be synchronized.

The timer function ‘scheduleAtFixedRate’ is used to ensure that the messages would be sent at 1 second interval on an average.

Since the consumer relies on the time stamp for recognizing the message it doesn’t matter it decouples the consumer from the identity of the nodes on which the daemons are running on.

The monitoring daemon uses these functions to send the system information to the message broker middleware. Now let us understand out implementation of the message broker middleware.

**2) Message Broker Middleware:**

[1]

Let us first understand the architecture of a message broker. The message broker architecture consists of a Producer, a Consumer and a broker. The message broker uses a Publish subscribe architecture. As per the above diagram one can see that the Producer produces some messages and publishes it on the broker. The messages can be subscribed by any consumer that is listening to the messages at the time when the messages are published by the producer on the broker. If no consumer is listening on the broker the messages will be lost. We are using the ActiveMQ broker for handling the messages and for creating the Producer and Consumer. Our implementation for the broker consists of the following classes:

1. ProducerTest.
2. FrontEnd.
3. ProducerTest: The ProducerTest class basically does the work of a Producer. It retrieves the system information by calling the daemon which is SystemsInformation, wraps it into a TextMessage and publishes under the Topic “G08\_xyz”. These messages are created at an interval of one second to provide a constant real time information of the system resource utilization. The ProducerTest also keeps a counter for Messages through which the consumer can know which message is being received at what time, which can further help in synchronization. The Producer is designed in such a way that it starts sending the overall average CPU and memory utilization of the system to the broker when it is started. Till then if the MPI PageRank program has not started the Producer will be sending the overall system information only. But as and when the PageRank starts it will start sending resource utilization of the MPI PageRank program also. However when the MPI PageRank program ends the Producer stops.
4. FrontEnd: The FrontEnd class does the work of Consumer. The messages that are published by the Producer on ActiveMQ broker are then received and interpreted by the FrontEnd class. The FrontEnd class helps subscribe to messages sent by the Producer to be able to retrieve the system information sent by the Producer.

**3.) Information Display Front- End or UI:**

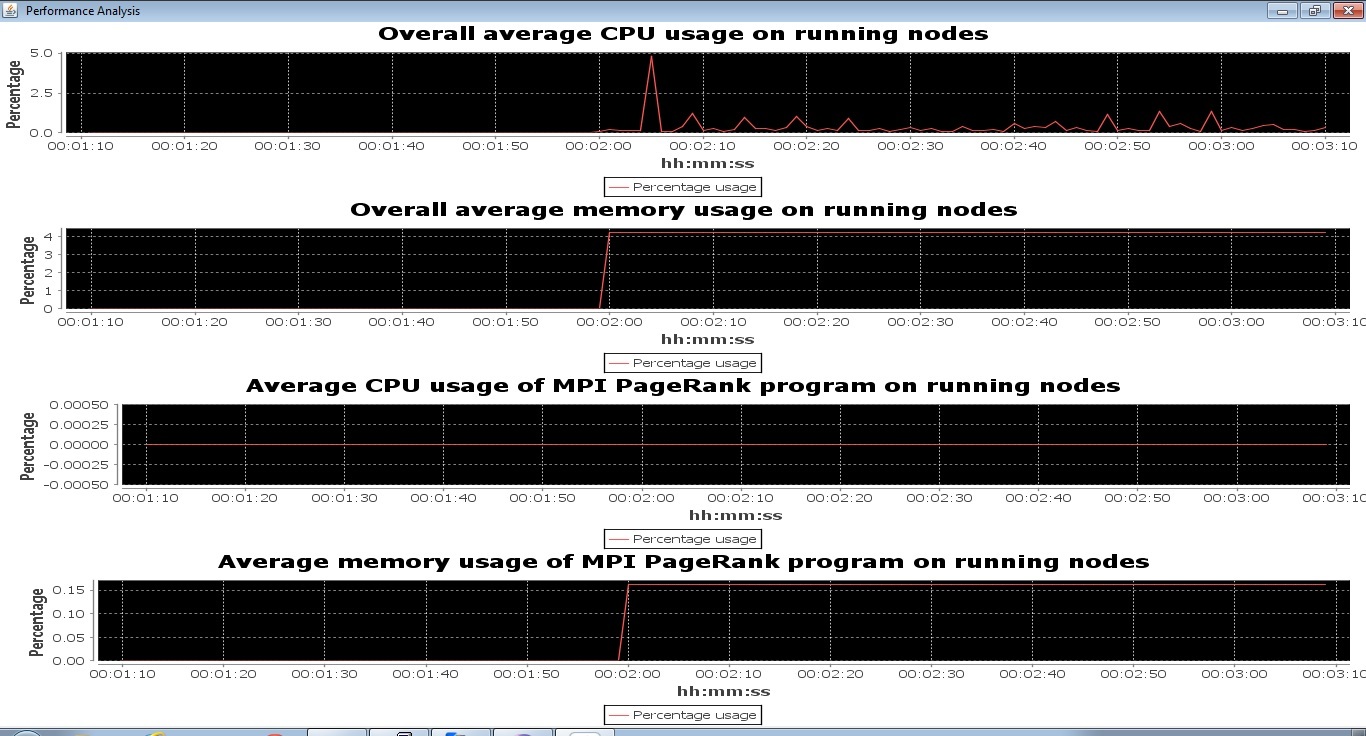
The front end of the application consists of two components; the user interface and a message consumer/receiver. The message receiver component of the front end implements a timer. The timer runs a thread on every second that performs a specified job. The job is to receive a message from the broker, check the order of the message and update the relevant UI charts.

Since, producer doesn’t use a queue at the broker; the data that is received by the consumer is the most recent.

The implementation involves a timer from Javax Swing package. At every second, the timer thread submits a job to the thread pool factory class. The thread factory reuses existing threads and spawns new threads if there are no threads available in the thread pool. The main advantage of this strategy is,

1. The timer thread will always be able to execute a new job at every second even if some job takes more than one second.
2. Better accuracy of the output.

The UI uses an ApplicationFrame that contains two scrollable JPanel s. These in turn house the Jfree charts that are created.

Following are the images of the UI during its operation.

Details of the PBS job submission script.

The basic structure of the Bare metal and virtual machine scripts is as follows:

1. Extract the unique node names in the machines file. This file will be used by MPJ for executing the MPJ program.
2. Start monitoring daemon on each node according to the nodes in the PBS\_NODEFILE. The pbsdsh command returns after issuing the command on each node. This allows the current script to continue. However, the monitoring daemons are still attached to the standard input and output. Hence, the daemons are not completely detached from the script. This ensures that, the daemons will be killed as soon as the script ends.
3. Start the mpj daemons on the current node and then on the second node and wait for some time for the daemons to start.
4. Issue the mpjrun command from the current node. At this point the shell does not return since the mpj program is still running and attached to the shell script on the current node.
5. As soon as the mpj program finished the shell script halts all the daemons on all the nodes in the machines file.
6. In case of VM script file, this algorithm runs after the virtual machines are reachable. And as soon as the mpj daemons halt after running the mpj program the virtual machines are also terminated.

**Following is the general flow of the application operation:**



**Interesting Findings:**

1. The fact that the code for retrieving system performance information and sending it requires a non-trivial amount of time affects the accuracy of frequency at which the data is sent. To better explain this idea let us consider the following diagram.



There are two timelines seen in this diagram. The first one shows the time intervals when we use a sleep inside a for loop or the timer from the swing library. The send one shows the ideal time intervals at which the messages need to be sent.

As can be seen clearly the, 7th message sent in the ‘actual sending interval’ time line is being sent at the 9th second of the ideal timeline. This introduces inaccuracy over time.

This problem was handled by using the ‘scheduleAtFixedRate’ Timer class function from the java.util package. This function automatically adjusts to the delay introduced by the task running time ensuring maximum accuracy.

1. If a persistent queue is used instead of just a topic, at the broker side then the consumer may receive outdated data from the broker before it starts receiving the latest data. This prompted us to use Topics instead of Queues at the broker side.

**Conclusion**:

We have successfully implemented a resource monitoring system for monitoring the resource utilization of a distributed system.

**Acknowledgements**:

We are thankful to Prof. Judy Qui, Stephen Wu and Ikhyun Park for their guidance and support.

**References:**

[1] <http://www.jakubkorab.net/2011/11/understanding-activemq-broker-networks.html>

2) <http://support.hyperic.com/display/SIGAR/Home>

3) <http://en.wikipedia.org/wiki/JFreeChart>

4) <http://www.vogella.de/articles/JFreeChart/article.html>

5) <https://cwiki.apache.org/NMS/>

6) [www.stackoverflow.com](http://www.stackoverflow.com)

7) <http://support.hyperic.com/display/SIGAR/PTQL>