## Project 3: Resource Monitoring System

## 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. As a start we are just planning to monitor the resources of a single local system.

**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. For a start we are just going to implement this monitoring system for our local machines which will be scaled at multiple nodes of a distributed system later.

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

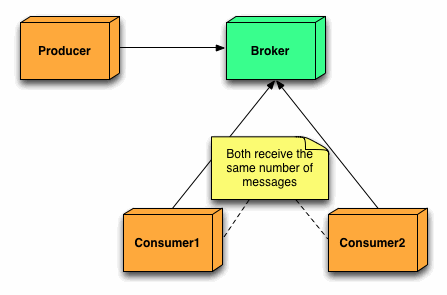
1. Monitoring Daemon.
2. Message broker Middleware.
3. Information display Front-End or the UI.

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 th 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(). 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.

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:

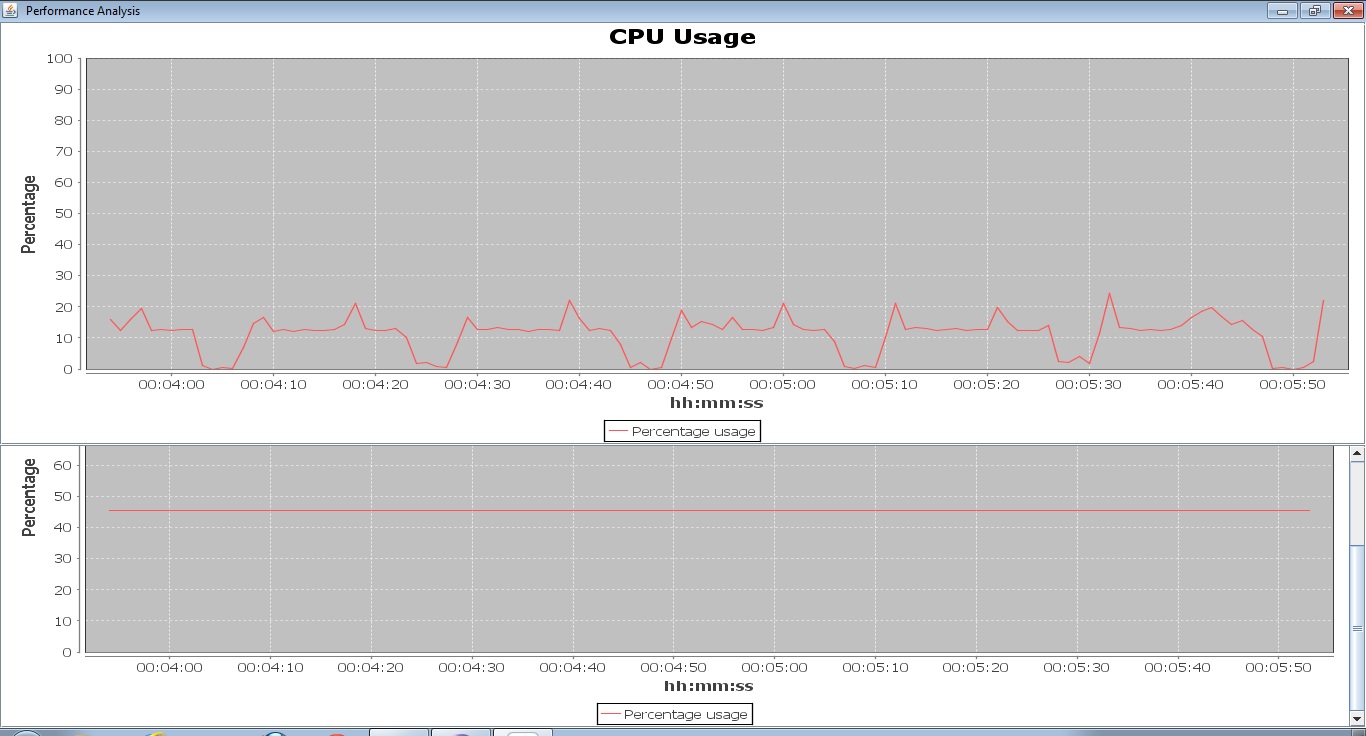
1. Producer.
2. FrontEnd.
3. Producer: The Producer class basically does the work of a Producer. It retrives the system information by calling the daemon which is SystemsInformation, wraps it into a TextMessage and publishes under the Topic “G08\_xyz”. These messages are created at an interval of one second to provide a constant real time informations of the system resource utilization. The Producer 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.
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.
5. Information Display Front- End or UI:

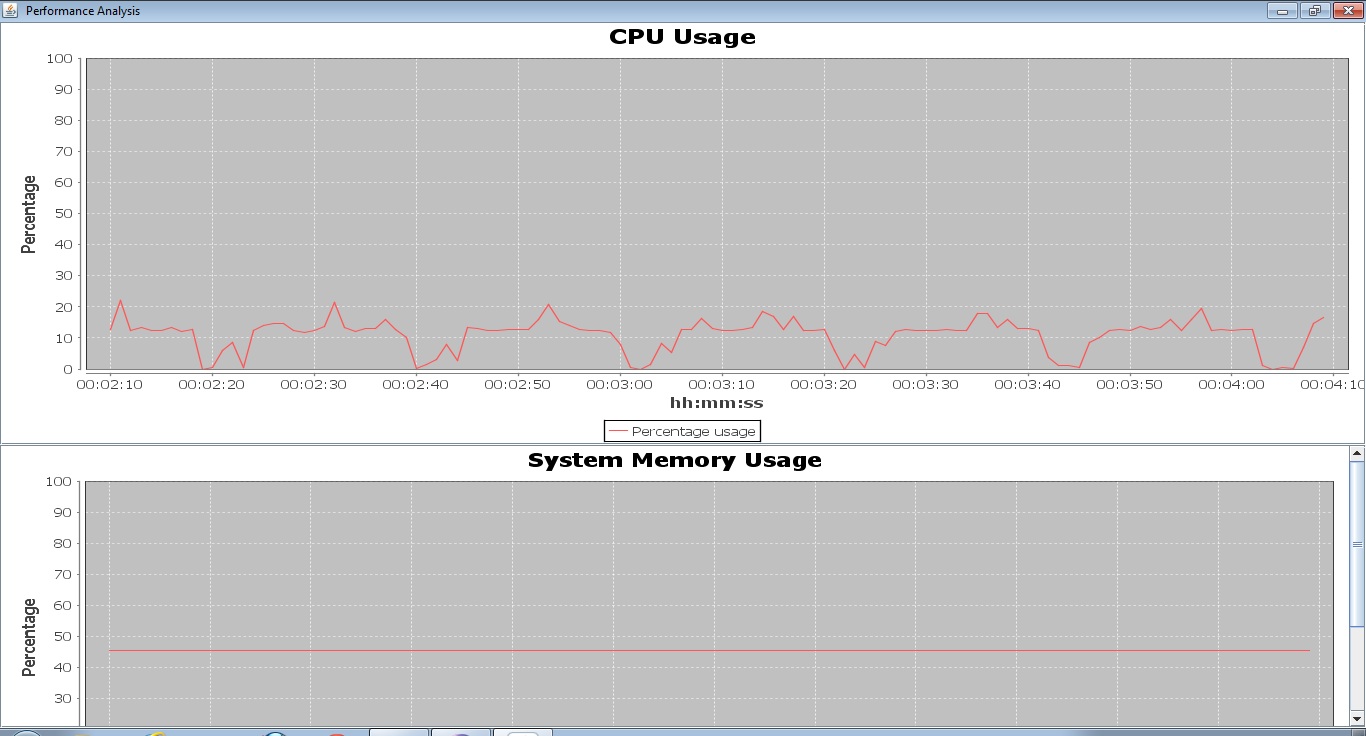
The front end of the application consists of two components; the user interface and a message consumer/receiver. The message receiver component of the front end implements a timer. The timer runs a thread on every second that performs a specified job. The job is to receive a message from the broker,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





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



**Conclusion**:

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

**Future work:**

We will be implementing the resource monitoring system for multiple nodes of a distributed system and displaying the aggregated information of all those nodes.

**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