# Project #2B Resource Monitoring System

# Goal

To implement a system that monitors and visualizes the resource utilization in a distributed set of nodes using message broker middleware.

# Deliverables

You are required to turn in the following items in a zip file **username\_Project2B.zip** via Canvas.

1. The completed source code of the Monitoring Daemon and monitoring front-end
2. Demo in class

Points will be reduced (maximum 1 point) if the filename or directory structure are different from instructed above.

# Evaluation

The total points for Project #2B is 40, where the distribution is as follows:

* 1. Completeness of your data collectors (20 point)
     1. Process data collector (8) points
     2. Node data collector (8) points
     3. Correct sending of JMS messages (4 points)
  2. Completeness of the data summarization and visualization (20 points)
     1. Correct message receiving (4 points)
     2. Correct summarization and graphs (16 points, i.e. 4 points for each graph)

# Introduction

Distributed resource monitoring is an important part of distributed systems. In large-scale computing environments, it is essential to understand the system behavior and resource utilization in order to manage the resources efficiently, detect failures, and optimize the distributed application performance. There exist several distributed monitoring solutions, such as Ganglia [1] and Nagios [2], to apply into a wide variety of distributed computing environments. Another example would be Amazon Cloudwatch [3], which allows users to monitor the resource utilization of their EC2 cloud resources. While these systems need to be comprehensive, care must be taken to ensure that they don’t disrupt or hamper the resources they are monitoring.

For this part of the project, you need to implement a system that monitors the CPU and memory utilization in a distributed set of nodes (two or more nodes which run on CS Machines). The system should support resource monitoring of Computer Science Linux Machines. Monitoring information needs to be collected and aggregated through the message broker and summarized to display the overall CPU and memory utilization percentages using graphs.

Also you need to be able to monitor the CPU utilization of a MPJ program. At this time it is sufficient to monitor MPJ or Hadoop. So there should be four graphs:

1. Overall CPU utilization of the nodes
2. Overall Memory utilization of the nodes
3. Process CPU utilization (MPJ)
4. Process memory utilization (MPJ)

The UI and the base functionality are provided with the project. You must implement the missing functionality.

**Note:**

For process monitoring you will need to run the MPJ PageRank on a CS machine environment and show the CPU and Memory utilization of the program.

# System architecture

There are three main components of this monitoring system: a **Message Broker**, **Monitoring Daemons running on nodes** and a **Monitoring UI**.

* **Message Broker:** a middleware that holds series of messages with specific topics/queues (using different functions from the ActiveMQ APIs), and waits for a Front-End Subscriber to pick the messages, i.e. ActiveMQ. AIs will set up instances of ActiveMQ to be used by the students. Students are advised to prefix their topics/queues with their username (eg: username\_xyz) to avoid conflicts when sharing the same brokers. Also students can run their own broker as well.
* **Monitoring Daemon:** a background process that runs on each compute node which captures and publishes the resource utilization information (CPU and Memory utilization required) to the Message Broker periodically. There are two components in each daemon: the node data collector collects the systems information and sends these to the message broker; the process data collector collects information for MPJ processes (if they are running) and sends them to the broker.
  + To collect the data student needs to use the Sigar [4] library API. The library is already included in the code we have provided. Students need to use the Sigar API methods to collect the information. Please refer to Sigar API and documentation for more details [1].
* **Client:** a client listens to the messages with specific topic(s) from Message Broker and puts them into a data collection. The aggregator runs periodically and summarizes this data so that it can be displayed in the graph. The summery data is put into the Graph Model by the aggregator. The client GUI is updated periodically with the data in the Graph Model.

These 3 components can run in 3 different machines. You can run multiple daemons to collect data from multiple machines.

Broker

Queue/Topic

NodeData Collector

ProcData Collector

Node DataCollector

ProcData Collector

Daemon 1

Daemon 2

DataCollection

Aggregator

GraphModel

UI

Client

**The system is configured through the monitor.properties file found in the bin directory.** Students can tweak the parameters in the monitor.properties if they need to.

You will need to implement the following class methods. The methods that you need to implement are clearly marked in the source code.

**Monitoring Daemon**

Java classes are located at src.main.java.com.resourcemonitor.daemon.

1. NodeDataCollector.java
   1. public MonitorData getMonitorData() : Get the CPU and Memory for the current node. [25-30 lines of code]

public MonitorData getMonitorData() {  
 MonitorData message = new MonitorData();

Mem mem = null;  
 CpuPerc cpuPerc = null;

/\* implement your code here

Get machine data using Sigar APIs. This function will be called time to time. This time is configured by sample.time in the monitor.properties.

\*/

return message;

}

1. ProcDataCollector.java
   1. public MonitorData getMonitorData() : Get the CPU and Memory for the MPJ [Process 15 -20 lines of code]

public MonitorData getMonitorData() {  
 MonitorData message = new MonitorData();

if (this.procIds == null) {

return null;

}  
 long mpiProcIds[] = this.procIds;

/\* implement your code here

1. Get processes’ CPU and Mem using Sigar APIs. This function will be called time to time. This time is configured by sample.time in the monitor.properties.

2. If the mpiProcIds are not null, go through them and collect performance data for these process IDs

3. Calculate average CPU and Mem usage on current node

\*/

return message;

}

* 1. private long[] getProcIds() : Get the process IDs for the MPJ Process, [25 -30 lines of code]

private long[] getProcIds() {  
 long list[];

/\* implement your code here to get the MPJ processes’ pIDs  
 e.g. to get the process ids for Google Chrome I can use the following command  
 ps ax | grep 'Chrome' | awk '{print $1}'  
 1. Call this command inside java program running under a Linux environment.  
 2. Parse the command result to a long list[] array.  
 \*/

return list;  
 }

**Communication broker to ActiveMQ**

Java classes are located at src.main.java.com.resourcemonitor.common

1. MonitorBroker.java

Please use a topic for communication. If you use a queue you may need to modify the interface we’ve provided (the receive part).

* 1. private void init(boolean sender) : Create the session, producer and consumer : [15 – 20 lines of code]

/\*\*

\* Initialize the connections. Create the session, producer and consumer

\* @throws MonitorException

\*/

private void init(boolean sender) throws MonitorException {  
 /\*\* implement your code

It’s similar to the ActiveMQ example in lab  
 1. Create topic/queue connection session  
 2. If it’s sender, create a producer. Otherwise, create a consumer

\*/

}

* 1. public void send (int workerId, MonitorData monitorMessage) : Create a JMS message from the data and send [15 -20 lines of code]

/\*\*

\* Create a JMS message using the Monitor Data and send using the producer

\*/

public void send(MonitorData data) {

/\* implement your code

Producer sends message to ActiveMQ broker  
 1. Construct/set message body by using javax.jms.MapMessage  
 2. Send the message

\*/

}

* 1. public void startReceive(final ReceiveHandler handler) : Create MonitorData from the JMS message [20 – 25 lines of code]

/\*\*

\* Receive a JMS message and convert it to MonitorData. After the monitor data is created call

\* the handler.onMessage(monitorData).

\* @param handler: the handler to call with the monitor message

\*/

public void startReceive(final ReceiveHandler handler) {  
 /\* implement your code

Consumer receives messages from ActiveMQ broker  
 1. Upon receiving any message, deserialize and fill them into MonitorData object  
 using handler.onMessage

\*/  
 }

**Front-end Graph interface client**

Java classes are located at src.main.java.com.resourcemonitor.client

1. Aggregator.java
   1. public void generate() : Generate the x and y values for the graphs by going through the messages in the DataCollection. :[70 -80 lines of code]

/\*\*

\* Run through the messages in data collection and create a summary from this data. This method is going to get called every few milliseconds configured by "aggregate.time" property. The middle of the current window can be found by "summary.getLastTime() + window". Students can calculate the average for each of the values you need to display in this window through the raw data collection.

\* Make sure you delete the data from the raw data collection that is already processed or that is below the current window. If they are deleted the data collection can grow. After the aggregate values are calculated you should add these values to the summary. In this method you may need to calculate values for multiple windows to compensate for the timing delays.

\*

\* \*\*\*\*\*\*\*\*\* Students should implement this method \*\*\*\*\*\*\*\*\*\*\*\*

\*/

public void generate() {

Map<Integer, ArrayList<MonitorData>> nodeMessageMap = dataCollection.getNodes();

Map<Integer, ArrayList<MonitorData>> procMessageMap = dataCollection.getProcs();

int procCount;

int count;

long time;

do {

double cpu = 0;

double mem = 0;

double procCpu = 0;

double procMem = 0;

/\* implement your code

The following average values calculation only aggregates the received messages without looking into their message weight; for instance, 5 messages collected in 10 seconds, 3 of them came from node A, 2 of them came from node B, the overall performance should be (0.6 \* Perf\_A + 0.4 \* Perf\_B). You could get the sender’s information from message.getWorkerId();

1. Calculate the average values for overall CPU and mem usage on all nodes  
 2. Calculate the average values for overall MPJ processes’ CPU and mem usage on all nodes

\*/

if (procCount == 0 && count == 0) {

summary.add(0, 0, 0, 0, System.currentTimeMillis());

} else {

summary.add(cpu, mem, procCpu, procMem, time);

}

} while (time < System.currentTimeMillis() - window);

}

# 6. Compile and Run

The instructions to compile and run the program can be found in the included README.txt. You need to install Apache Maven [5] for compiling the program. The installation instructions can be found in

<http://maven.apache.org/download.cgi>

Here are the general instructions for installing Maven

1. Download a Maven distribution
2. Unpack it.
3. Set the environment variable M2\_HOME to point to the maven distribution
4. Add the Maven distribution’s bin directory to the PATH environment variable
5. Source the environment

**How to install Maven**

# download and unzip maven

cd ~/

wget <http://www.eu.apache.org/dist/maven/maven-3/3.3.3/binaries/apache-maven-3.3.3-bin.tar.gz>

tar -zxvf apache-maven-3.3.3-bin.tar.gz

# set up the environment (change skamburu to your username)

vim ~/.bashrc

# add the following lines to ~/.bashrc, and then :wq!

export MAVEN3\_HOME=/u/skamburu/apache-maven-3.1.1

export PATH=$MAVEN3\_HOME/bin:$PATH:.

# save the ~/.bashrc with vim commands :wq

# source environment

source ~/.bashrc

**How to build the project**

Go to the Project2B directory and type ~/Project2B/

cd ~/Project2B/

mvn clean install assembly:single

This will build the jar file called resourcemonitor-jar-with-dependencies.jar inside target directory. **Copy the resourcemonitor-jar-with-dependencies.jar file to the bin directory.**

**Running ActiveMQ**

Before running the program you need to run an ActiveMQ broker instance. ActiveMQ is very easy to download and run on your local machine. For example here is the procedure I used on my local machine.

wget <http://archive.apache.org/dist/activemq/apache-activemq/5.9.0/apache-activemq-5.9.0-bin.tar.gz>

tar -zxvf apache-activemq-5.9.0-bin.tar.gz

cd apache-activemq-5.9.0/bin/

./activemq start

(Also you can use ActiveMQ 5.12.0, the latest version. These two versions have no difference in our projects.)

Now you can go to the web browser and type <http://localhost:8161/> to view the web console of ActiveMQ.

Running ActiveMQ locally will only work if you run the project locally. If you run the project in distributed mode in different machines (CS machines) your program may not work because of the port restrictions. A public ActiveMQ instance for you is as follows:

**A public ActiveMQ**

Configuration: IP at 156.56.93.171

Service monitoring at <http://156.56.93.171:8161/>

Students use [156.56.93.171:61616](http://156.56.93.171:61616/)to send and receive messages from it.

**Before you run the daemons or the client, you need to change the ActiveMQ URL configuration in the monitor.properties file to point to the ActiveMQ URL.** For example if you are running the ActiveMQ locally, your configuration should be

broker.url=tcp://localhost:61616

**How to run performance collector daemon(s)**

To start the daemon, you will run the daemon.sh file in the bin directory on CS Linux machines. When you run the daemon.sh you have to give a unique id as a parameter.

# run a daemon #1 in ssh terminal 1

cd ~/Project2B/bin/

daemon.sh 1

# run a daemon #2 in ssh terminal 2

cd ~/Project2B/bin/

daemon.sh 2

**How to run client GUI on your laptop or CS machines with physical access**

To start the client, you must change the field of “topic.name” inside the monitor.properties and run the client.sh (for Linux) / client.bat (for Windows) file in the bin directory under X Windows or Windows 7/8 environment which is GUI supported. In other words, plaintext terminal such as puTTy cannot show the client GUI!

# run a daemon #1 with physical login to X windows

$ client.sh

# for windows users, make sure the publisher daemons are running, then just double click on the client.bat

**How to run the example shown in lab**

After you have successfully compiled the code, please use the following commands to run the tester program shown in the lab.

cd ~/Project2B/bin

java -classpath ../target/resourcemonitor-jar-with-dependencies.jar -Djava.library.path="../lib" com.resourcemonitor.daemon.PerformanceExample

**References**

[1] Ganglia project: <http://ganglia.sourceforge.net/>

[2] Nagios project: <http://www.nagios.org/>

[3] Amazon CloudWatch: <http://aws.amazon.com/cloudwatch/>

[4] [Sigar Resource monitoring API](http://www.hyperic.com/support/docs/sigar/), <http://www.hyperic.com/products/sigar>, <http://sourceforge.net/projects/sigar/>, <http://www.hyperic.com/support/docs/sigar/>

[5] Maven project. <http://maven.apache.org/>