Dynamically Provision clusters on Academic Cloud

Purshottam Vishwakarma
School of Informatics and Computing
Indiana University, Bloomington
pvishwak@indiana.edu

Bitan Saha
School of Informatics and Computing
Indiana University, Bloomington
bsaha@indiana.edu

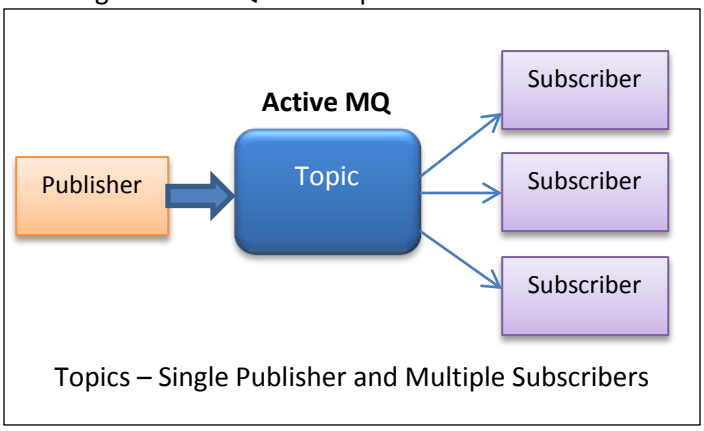
Introduction:

Dynamic Provisioning provides the ability and possibility to use on-demand resources in a shared academic Cloud environment; user of this system can simply send a request with specifying their needs to the resource manager [1] to obtain different kinds of computing resources, where the requested computing resources are deployed or instantiated on-the-fly. For example, users of FutureGrid can easily obtain a set of Bare Metal machines from Torque resource manager or boot up a set of Virtual Machines with using India-Eucalyptus.

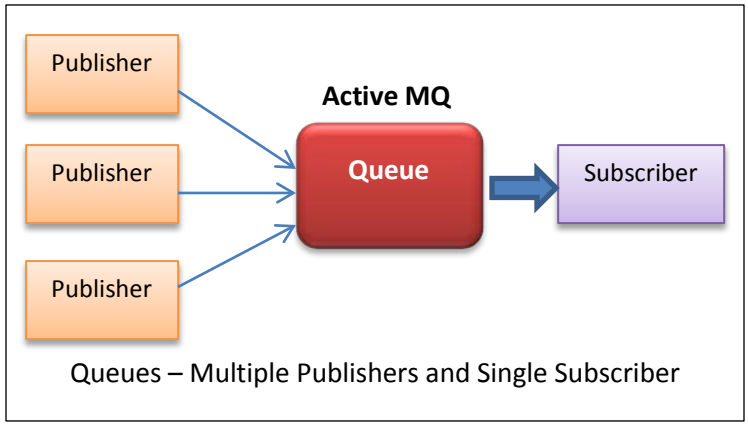
With the support of FutureGrid System Admin team, we have built a Dynamic Provisioning system which can switch between Bare Metal and Virtual Machine compute cluster environments utilizing XCat, Moab and Torque job scheduler. For this project, you need to understand the system architecture, and implement job scripts to run your MPI PageRank and Monitoring system using the provided dynamic infrastructure.

Techonology Used:

We have chosen "**Java**" as the programming language to implement both Monitoring system and the Daemon. We are using "**Active MQ**" in non-persistent mode as our Message broker.



Topics – Single Publisher and Multiple Subscribers



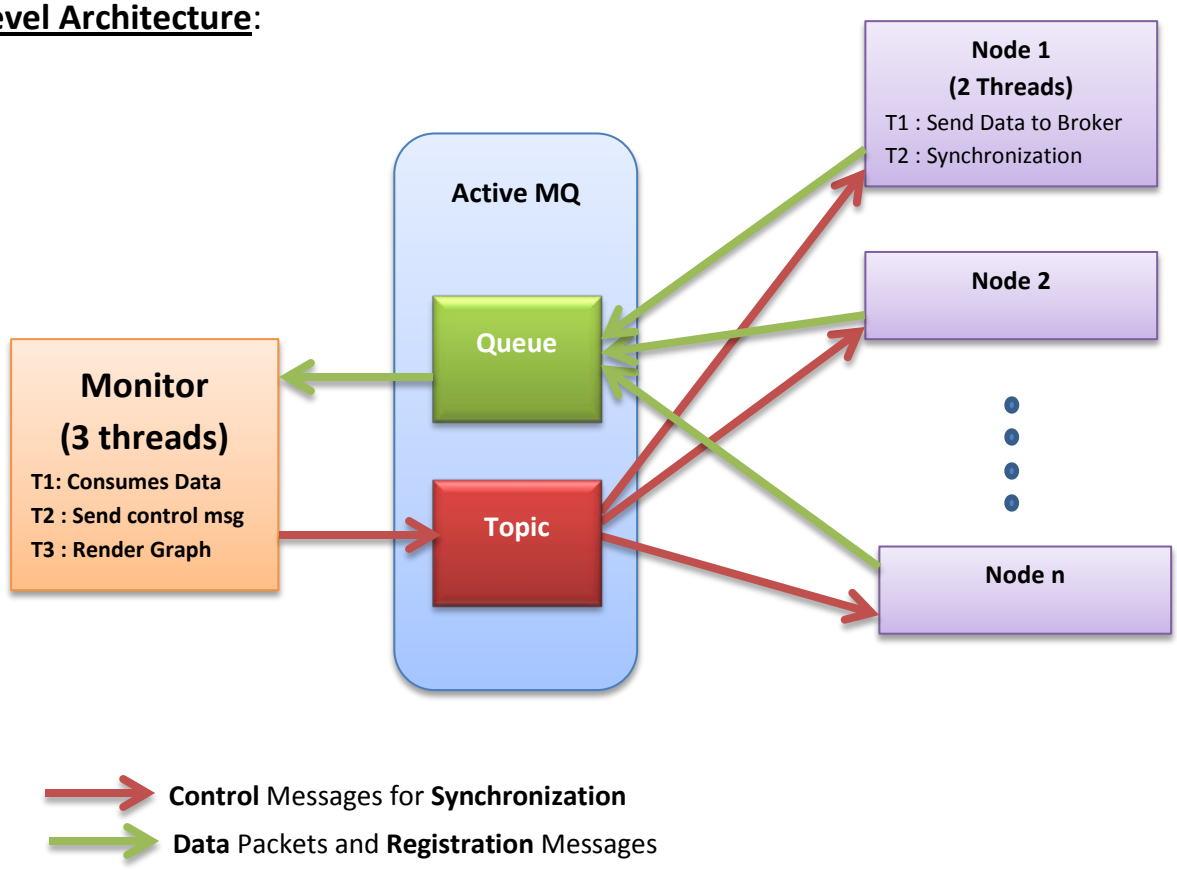
Queues – Multiple Publishers and Single Subscriber

We would be using both *topic* as well as *Queue* for communication in our implementation.

Topic is used for control messages (Synchronization) whereas *Queue* is used for transferring data and registration messages.

Our Monitoring system is designed to support n (n>= 1) number of nodes.

High Level Architecture:



- Each node runs a daemon to collect CPU and memory Usage and routes it to the monitor through the Queue for visual representation.
- Monitor uses Topic to send control messages.
- New Nodes also use Queue to send registration messages to the Monitor.
- Monitor keeps a buffer for each node. So for n nodes it will have n buffers.
- One separate thread in the monitor reads all the buffers and renders the graph. After the data has been read it is cleared from the buffer.

Registration & Synchronization:

- Below is the data packet structure. Each member in the structure is described below:

Data Packet Structure
Message ID
State ID
CPU value
Memory value
Daemon ID

- *Message ID*: This is an integer value used to identify the sequence number of the messages which is incremented with every data packet sent. The first data packet sent will have the message ID = 1 and subsequently incremented by 1.
- *State ID*: This is used to identify the Phase of data transfer. The state changes every time a new node registers i.e. all the nodes will start sending data starting with message id 1 and this new state id.
- *CPU Value*: This is the CPU usage at a given instance.
- *Memory Value*: This is the Memory usage at a given instance.
- *Daemon ID*: This is the combination of system name and IP address to identify each node uniquely.

- Whenever a new node joins the monitoring system, it sends a **registration** message to the Monitor through the Queue.
- Once the monitor receives the registration message it clears the buffer of each Node and sends a synchronization message with a new **State ID** to all the Nodes including the newly added node to *reset their Message ids to 1*.
- Hereon all the registered nodes will use this newly generated **State ID** along with message ID starting from 1 to send the data messages. This indicates a new phase of data transfer between Monitor and the Nodes. If the monitor receives a data packet with old state id, those data packets are discarded. This mechanism is used to synchronize all the nodes.

Auto Correction:

- We have incorporated *auto correction* feature in order to deal with the *lost data packets*. As an example, suppose there are 3 nodes in the system. Monitor will maintain 3 separate buffers for each node. The message received from each of the nodes are stored in the buffer corresponding to each node as shown the below diagram. Now suppose the monitor doesn't receive Message ids 999 and 1000 from Node 3. Note that we are using Active MQ with *non-persistent* mode. Hence requesting the broker to send the lost data packets is out of scope.

Node 1 Buffer	Node 2 Buffer	Node 3 Buffer
1005	1005	1005
1004	1004	1004
1003	1003	1003
1002	1002	1002
1001	1001	1001
~~1000~~	~~1000~~	Lost Packet
~~999~~	~~999~~	Lost Packet
998	998	998
997	997	997

- In such a scenario, we will discard the message id 999 and 1000 from all other node buffers. This entails we display the data from message id **1001** after **998**.

System Architecture

Based on the information received from the monitoring infrastructure, users will programmatically switch/re-provision their nodes to another environment (eg: from Linux to Linux VM's). Figure 1 shows the interactions between components within this system.

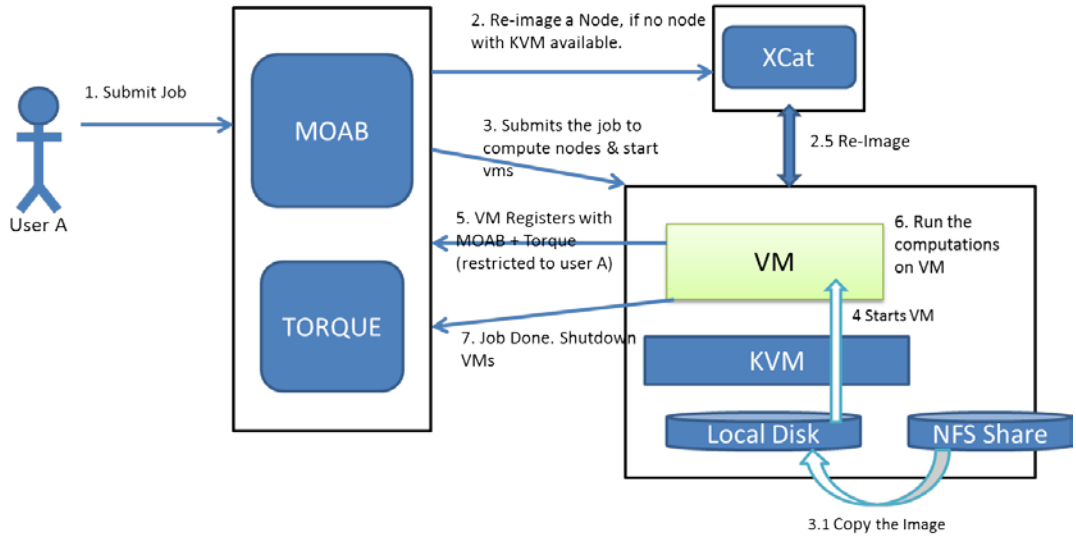


Figure 1 User interactions with Dynamic provisioning system

We have a set of nodes with the above infrastructure reserved for this call under a special job queue (b534) in india.futuregrid.org. Since the number of nodes is limited and the user jobs will be going through the queue, it's advised that you start working on this assignment soon, to avoid any contention of resources towards the assignment deadline.

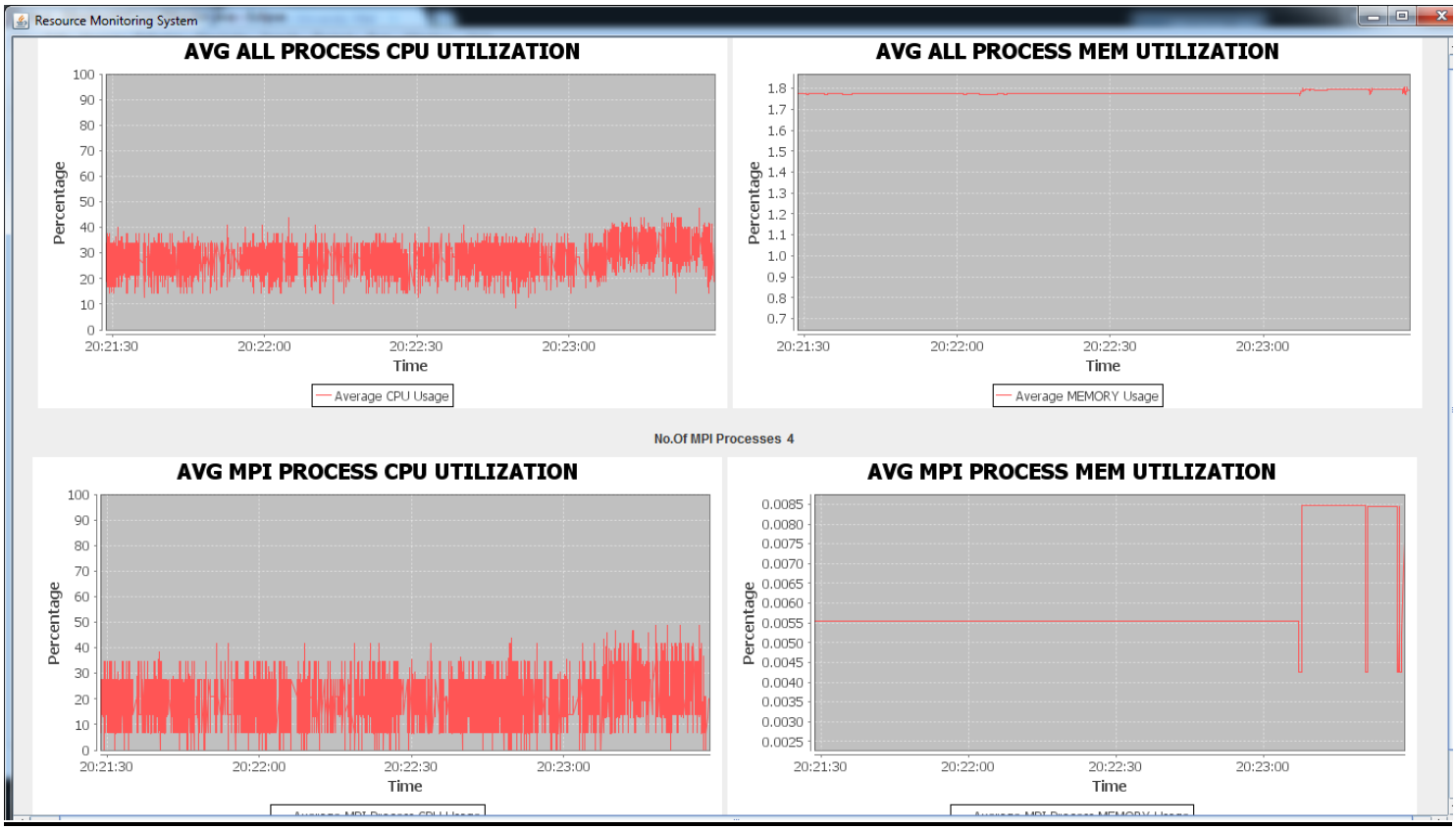
Below sections show two main types of jobs.

Scripts for BareMetal:

Following steps were followed to run the scripts on BareMetal.

1. Acquire the PBS nodes. Here we acquired i97 and i98 nodes which were assigned to us.
2. Output $PBS_NODEFILE contents (unique) to **nodes** file. In the nodes file we will have unique node ids. i.e. i97 and i98.
3. Loop each node in the nodes file.
    a. If the node is the head node i.e. the first node in the file, start the daemon without ssh.
    b. Else start the daemon using ssh (ssh $LINE "cd $HOME/B534/P2/DeamonCode;./startDeamon.sh &" &).
4. Load the mpi module using the command "module load openmpi"
5. Run the mpi program using *mpirun*.
6. Loop each node in the nodes file.
    a. Close the daemons using ssh.
    b. For head node close the daemon without ssh.
7. End bare metal job.

In the below Snapshot for the Bare Metal, you can see that the monitor registers the CPU utilization for the MPI Processes when number of processes = 4. The Graph also shows the number of processes running at that time which is 4 at the time the snapshot was taken.



Scripts for VMs:

Following steps were followed to run the scripts on VMs.

1. Acquire the PBS nodes. Here we acquired i97 and i98 nodes which were assigned to us.
2. Shutdown the VMs if there are any already running VMs and wait for 60 seconds to shut down.
3. Start the VMs using the script *start_vms*.
4. Output $VM_NODEFILE contents (IP addresses) to **nodes_vm** file. In the nodes_vm file we will have the IP Addresses of the Virtual Machines.
5. Loop each IP addresses in the nodes_vm file..
    a. Start the daemon using ssh (ssh $LINE "./startDeamon.sh &" &).
6. Start the mpi program using ssh on the Headnode using the below command.
   ssh $HEADNODE "module load openmpi;mpirun --mca btl_tcp_if_exclude lo,eth1 -hostfile nodes_vm -np 6 mpi_main -i pagerank.input.2M -n 500 -t 0.000001 >> mpi.out"
7. Loop each node in the nodes_vm file.
    a. Close the daemons using ssh (ssh $LINE "killall -e java -u pvishwak &" &).
8. End job.

In the below Snapshot for the VM, you can see that as soon as the MPI processes start 10 seconds after starting the daemon, the monitor registers the CPU utilization for the MPI Processes. The Graph also shows the number of processes running at that time which is 6 the time the snapshot was taken.

Reference

[1] TORQUE Resource Manager http://www.clusterresources.com/products/torque-resource-manager.php
[2] KVM Hypervisor http://www.linux-kvm.org/page/Main_Page
[3] libvirt: The virtualization API http://www.libvirt.org/
[4] Torque Qsub: http://www.clusterresources.com/torquedocs21/commands/qsub.shtml#I
[5] Torque Job submission: http://www.clusterresources.com/torquedocs/2.1jobsubmission.shtml

Acknowledgement

We would like to acknowledge the efforts of our Instructor Dr. Judy Qui and Associate Instructor Stephen for helping us in accomplishing this project on time.