CS423 Spring 2015
MP4: Dynamic Load Balancer
Due April 27th at 9:00 am 2015

1. Goals and Overview

1. In this MP you will design a Dynamic Load Balancer architecture for a Distributed System
2. You will learn and implement system monitors, adaptation and load balancing schemes to improve the system utilization and performance of a distributed application under limited bandwidth, memory and transient CPU load constraints.

2. Development Setup

For this MP you will develop a Dynamic Load Balancer for Linux. You will work entirely in the Linux user-space using Virtual Machines. The development setup will simulate a distributed infrastructure in which a local small factor device (e.g. cellphone or tablet) delegates some of its processing load to a remote higher-end computer.

For this MP you can choose between **Java, Python and C/C++.** You will use the provided VM as the remote higher-end computer (i.e. Remote Node) and also you will be provided with an additional Virtual Machine for your group to use as the local small factor device (i.e. Local Node). Your group's local node is named as **sp15-cs423-gxx-s.cs.illinois.edu**, where xx is your group number. Say you are group 1, your local node will be **sp15-cs423-g01-s.cs.illinois.edu**. You can turn on/off both VMs using the vSphere web client. Two users have been created for you: A regular user cs423 with sudo access (password: tallbutter85), and a TA-only user cs423ta. **Please log in your small VM and change the password as soon as possible. DO NOT change the cs423ta password.**

Finally, you are encouraged to discuss design ideas and bugs in Piazza. Piazza is a great tool for collective learning. However, please refrain from posting large amounts of code. Two or three lines of code is fine. High-level pseudo-code is also fine.

3. Introduction

The availability of low-cost microprocessor and the increased bandwidth of network communications have favored the development of distributed systems. However, one common problem of these systems is that in many cases some nodes remain underutilized while other nodes remain highly utilized. Maintaining even utilization in these systems is useful to increase system efficiency, performance and reduce power consumption.

One common technique to achieve even utilization across nodes, is the use of dynamic load balancing techniques that actively transfer the load from highly utilized nodes to lower utilization nodes.

This type of algorithm usually has 4 main components:

1) **Transfer policy**: Determines when a node should initiate a load transfer

2) **Selection policy**: Determines which jobs will be transferred.

3) **Location policy**: Determines which node is a suitable partner to transfer the load to or from. Please note that for our MP this policy is trivial, because there are only two nodes in the system.

4) **State Information policy**: Determines how often the state information about each node should be shared with other nodes.

4. Problem Description

In this MP, you will develop an architecture that implements a **dynamic load balance algorithm**. You will decide a suitable transfer, selection, location and information policies, evaluate and justify your decisions.

You will be required to implement a minimal set of components and functionality and you will have the opportunity to further augment and improve your design. You are encouraged to evaluate the performance impact of your decisions and to present your results as part of your demonstration.

The system architecture is composed of 2 nodes: a limited CPU Mobile node and a high-end Server node. Both of these two nodes work together to compute a vector addition task. **You need to malloc a double vectors A with 1024 * 1024 * 32 elements.** For vector A, initialize all elements with 1.111111. For each element of vector A, your task is to do the following computation (**DO NOT optimize the following computation**):

```
for( int j=0; j < 1000; j++){
        A[i] += 1.111111;
}
```

Note that the workload here is fully parallelizable; Your system should divide the workload in smaller chunks, called jobs. Each job is independent of each other and each job should roughly take a similar time to complete. It's up to you to decide the size of the jobs during the implementation. However, initially there should be at least 512 jobs. **During the transfer of the jobs, you MUST transfer the real data within vector A. You cannot simply tell the other node that please compute vector addition from, say, index 1000 to 2000. You must transfer A[1000-2000] to the other node to finish the above computation.**

Our system will have three phases:

1) **Bootstrap Phase**: Originally, the Local Node will contain all the workload and it must transfer half of the workload to the Remote Node. This will be called the bootstrap process. You **must** print out enough information (say, the job ID) on screen so that we can tell you are indeed transferring the jobs.

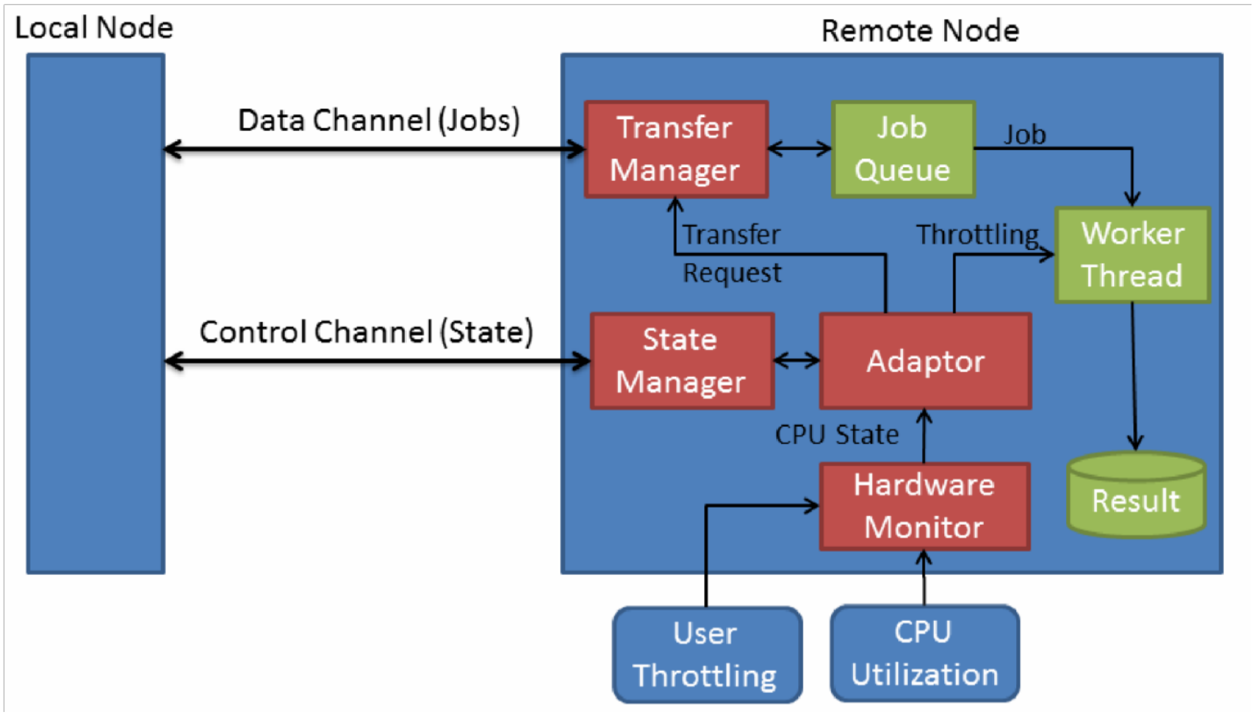
2) **Processing Phase**: After the bootstrap phase, each node divides the half workload into chunks of data called jobs; each node will have a queue with roughly the same number of jobs. At this

point, each node will start processing each of these jobs one by one and storing the result locally. Our dynamic load balancer will only work during this processing phase (i.e. by applying the transfer, selection, location and state information policies).

3) **Aggregation Phase**: After all the jobs are successfully processed, our system must aggregate the result into a single node (either local or remote) and display the result.

5. Implementation Overview

In this section, we present the minimal architecture that you must implement. Below you will find a figure showing the components of this architecture:



- **Job Queue**: It is responsible of **storing the pending jobs and dispatching them to the Work Thread**.

- **Worker Thread**: The Worker Thread is responsible of **processing each job and locally storing the result. The Worker Thread must react to Throttling**. This means, the worker thread must limit its utilization to a percentage value provided by the Adaptor/User. For example, a throttling value of 70% means that during each 100 milliseconds, the Worker Thread must be sleeping for 30 milliseconds and processing the job for 70 milliseconds. During the sleeping period, the worker thread must not spin on a variable. You must use adequate system mechanisms to put the thread to sleep.

- **Hardware Monitor**: The Hardware monitor is responsible for collecting information about the hardware. You are required to **collect information about the CPU utilization** of the system.

However, as noted in the next Section you can also collect Network Bandwidth and other system information used by any of the policies of the Load Balancer (Transfer Policy, Information policy and Selection policy). As part of the hardware monitor, **you must implement an interface that allows the user to dynamically specify a Throttling value to use to limit the Worker Thread during the execution**. During the demo, we'll try different throttling values (like 0.1, 0.25, 0.5, 0.75) to limit the usage of either the local node or the remote node (or both).

- **Transfer Manager**: The Transfer Manager is responsible of performing a load transfer upon request of the adaptor. It must **move jobs from the Job Queue and send them to remote node**. It must also receive any jobs sent by the remote node and place them in the local Job Queue. You can use any protocol that you choose (e.g. TCP, UDP, HTTP, etc.). **You MUST print out enough info on screen (like job ID) when transferring and receiving jobs, so that during the demo we can tell that the jobs are indeed being transferred between the two nodes**.

- **State Manager**: The State Manager is responsible of **transferring system state to the remote node**, including, the number of jobs pending in the queue, current local throttling values, and all the information collected by the Hardware Monitor. For this component, you will need to choose an Information Policy. This policy will dictate how often the collected information is exchanged including time intervals or events. Careful design of this policy is important due to the performance, stability and overhead tradeoffs.

- **Adaptor**: The Adaptor is responsible for **applying the Transfer and Selection policies**. It must use information provided by the State Manager, and the Hardware Monitor and decide whether a load balance transfer must occur. Please note that the most basic policy that you can implement is to consider the Queue Length or the **Estimated Completion Time** (based on the throttling and the queue length). A more sophisticated policy should use the Queue Length or the Estimated Completion Time and also other factors.

  As part of the implementation of your transfer policy, you must choose between sender-initiated transfers, receiver-initiated transfers or symmetric initiated transfers. Each of these have different tradeoffs in terms of performance, overhead and stability.

6. Further Improvements

In this MP, you will be awarded extra points for further improving your code over the basic architecture. There are several issues that have room for improvement, and also, several design decisions that must be validated. In this section, we discuss some of these ideas:

- You can analyze the impact of choosing between sender-initiated transfers, receiver-initiated transfers or symmetric initiated transfers. You should show graphs and highlight the tradeoffs in your final report (5 pts).

- You can extend the Hardware Monitor to consider Bandwidth and Delay. Assume the communication between the Local Node and the Remote Node is limited to **10 Mbit/s** with average delay of **54ms (Std. Dev. 32ms) and average packet loss of 0.2%.** Use this information as part of your Transfer Policy. Limited bandwidth can make load balance a very costly operation and further degrade the performance. (5pts)

- You can compress the data when sending it over the internet to save time and bandwidth. (10 pts)

- You can implement a Graphical User Interface that shows the progress of your job and also clearly shows when your system initiates a transfer and changes a throttling value. (15 pts)

- You can increase the number of Worker Threads to exploit the concurrency of the system (5 pts)

7. Software Engineering

As in previous MPs, your code should include comments where appropriate. It is not a good idea to repeat what the function does using pseudo-code, but instead, provide a high-level overview of the function including any preconditions and post-conditions of the algorithm. Some functions might have as few as one line comments, while some others might have a longer paragraph. Also, your code must follow good programming practices, including small functions and good naming conventions.

For this MP you should choose adequate data structures to use. C++ and Java provide a large number of data structures with various performance characteristics. During the interview, you should justify your data structure design decisions.

8. Hand-In

All MPs will be collected using compass2g. You need to submit a single group_ID_MP4.zip (replace ID with your group number. If the filename is incorrect, -10 pts) file through compass2g. Only one of the group members need to do the submission. You need to provide all your source code (*.c and *.h, including the given source code) and a Makefile that compiles both your kernel. Do not include binaries or other automatically generated files.

Finally, you must write a document briefly describing your implementation and design decisions. **You must include all the analysis that influenced your design decisions including any experimental evaluation that you might have performed.**

In addition to the submission, you should be able to demo your source code and answer questions during an interview on Monday with the graders. This interview will be a major part of the grading. All the members of the group must be present. This interview will be a major part of the grading. Instructions for how to sign-up for the slot interviews will be posted at a later time on Piazza.

9. Grading Criteria

In this MP, you will be awarded extra credit for implementing additional features for your code. However, you are required to implement the minimal set of features described by the proposed architecture in this Handout.

Criterion	Points
• Bootstrap Phase	10
• Aggregation Phase	10
• Adaptor including Selection and Transfer Policy	15
• Transfer Manager	10
• State Manager	10
• Hardware Monitor	10
• Job Queue	10
• Efficient use of Locks Synchronization Primitives and Conditional Variables	5
• Code Compiles Successfully	5
• Code Follows Good Software Engineering Principles	5
• Documentation	10
• Bonus Points - Further Improvements	Up to 40
TOTAL	**140 / 100**

10. References

[1] N.G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems", presented at IEEE Computer, 1992, pp.33-44.