Abstract

Implementation and development of a Dynamic Load Balancer for a distributed system.

Introduction

Although we read through and implemented the MP4 spec step-by-step, there were quite a few places where our implementation deviated. Those deviations, as well as a brief overview of the project and the important observations we had, will be outlined in this document.

Environment Setup

Testing was done on a single laptop communicating with a local virtual machine. I also created a git repository of the root folder and uploaded it to share with the team.

Workload Description

The task we decided to use as our workload was to create a color histogram from a collection of images. The master node first opens the images and the color values of each pixel are read into a buffer. These ints are then divided into same-sized arrays and placed in a queue. Workers will then pull from the queue, determine the color contents of each pixel and store the result. The result is then printed out once all jobs are finished.

Implementation

2.1. Job Queue

We used a Linked Blocking Queue to implement our job queue. In addition to being thread safe, the Linked Blocking Queue supports operations that wait for the queue to become non-empty when retrieving an element, and waits for space to become available in the queue when storing an element. It seemed an obvious choice to use for our multi-threaded program, especially in the context of a Producer/Consumer. When the jobs in the JobQueue are processed, they are stored in a ResultMap, which is a thread-safe hashmap. Both the JobQueue and the ResultMap function well when being thrown around by multiple threads.

2.2. Worker Thread and Throttling

The throttling for our worker threads is not controlled by the adapter as the spec suggests but instead gets the current throttling value from the State Monitor. The Worker Threads time how long each job takes to finish, calculate the amount of time they need to sleep, and set a wake-up timer for that amount of time before going to sleep. By implementing our workers this way, we can ensure that jobs are finished properly without interruptions and that threads would not be put to sleep while holding a lock. The math behind sleep time calculation is: workTime * ((1.0f - throttle) / throttle)

2.3 Hardware Monitor

Our Hardware Monitor was a simple class that consisted of two things: a field where the current throttling value is stored and a timer that would initiate a read of the CPU usage. To read the CPU usage, we parsed sysout from a common (os x and linux) utility called sar.

2.4 Transfer Manager

The transfer manager connects the two nodes through one port and follows a general TCP-like scheme with SYN and ACK packets. The two nodes, when requesting or responding, use the "_syn"/"_ack" notation after message strings to signal start and completion of tasks. The transfer manager simply sends jobs as well as results between the nodes.

2.5 State Manager

The state manager connects the two nodes on a separate port from the transfer manager for scalability and mutual exclusion reasons, though this is probably not necessary. It only sends packets of throttle, cpu usage, jobs left, state, scaling, and job time values from the master to the slave. These values can then be accessed from the other threads.

2.6 Adapter

The adapter was made a part of our main thread. After the workload has been divided, it prompts the Transfer Manager to send half of the load over to the remote node and waits for it to finish before starting the worker threads. While the jobs are being processed, the adapter in the local node periodically checks the State Managers of the local and remote nodes and decided whether or not to initiate a transfer. Only the master node can initiate transfers (the Client).

2.7 Transfer Policy

The transfer policy uses two interesting formulas for calculating the optimal number of jobs to transfer, and in which direction. Each node has a job count and a scaling factor (to estimate how long a job will take). If you want these to remain balanced, you can subtract/add a value i to each job count and solve for it, i.e. $(a-i)x = (b+i)y$. Solving for i gives the optimal number of jobs to send not taking into account concurrent work processing times or network transfer times. To smooth this out, we used a system of equations to solve for a corrected transfer count given the time it takes to process one job, and the time it takes to send one job over the network. $(t/p)n_t = n_p$ and $n_t + n_p = n$, where n_t is the number of nodes to transfer and n_p is the number of nodes to keep and process, given n optimal nodes to transfer from the last step. Solving for n_t yields $n / ((t/p) + 1)$.

2.8 Observations

We tested out several transfer policies, but we could not find anything too solid. At the moment, our transfer policy relies on a scaling factor composed of CPU usage and throttling, but this relies on the two communicating machines having similar CPUs and ignores the rest of the system state. We figured that we could use the job processing time to scale instead, but for our jobs this varied too much to make smooth transfers. However, it is very apparent that when we throttle one side or the other, we can get steady transfers in mostly one direction.

2.9 GUI

For fun's sake, we also added a GUI which allows you to set the throttling on a node and view the overall progress on the Master node. At one time, we also had an image being loaded into the GUI upon completion of all of the jobs, to represent the resulting hashmap histogram.