Dynamic Load Balancing With Central Monitoring of Distributed Job Processing System

P. Srinivasa Rao
Computer Science
YPR College of Engineering &Technology, A.P, India

V.P.C Rao, PhD.
Computer Science
St. Peter's Engineering College, A.P, India

A.Govardhan, PhD.
Computer Science
JNT University Hyderabad
A.P, India

ABSTRACT

This paper presents a Dynamic load balancing with a centralized monitoring capability. The purpose of using a centralized monitoring feature was based on the idea that the computation in a environment may be distributed, but the status of each task or job must be available at a central location for monitoring and better scheduling. This also allows better management of the jobs. The framework also addresses the inherent need for uniform load distribution by allowing the dispatcher to check against the status of the processors before a job is dispatched for processing. This eliminates the need for processors to be burdened with the task of re-routing the job when they discover that they cannot process the received job. The basic requirement of assigning a priority and processing as per priority is built into the framework. As a proof of concept, we simulate the framework with a Java and JMS compliant OpenMQ based monitor, dispatchers, processors and a centralized database. The framework will have the capability to scale horizontally as well as vertically.

General Terms

Distributed Job Processing, Load Balancing,Parallel Processing.

Keywords

Keywords - Distributed, Job Processing, Priority,Load Balancing , Monitoring, Recovery.

1. INTRODUCTION

Many of the Job processing systems available today are commercial systems that use proprietary technology (hardware / software) for performing the tasks. Quite prevalent are those with Mainframe systems. Such systems do have complex monitoring and control software. But these are less flexible, tightly coupled with other associated software and/or hardware, thus limiting the scalability to the extent the platform supports. Added to this is the cost overhead of upgrading the system when more computational power is needed.

Major limitations of such systems being:

1. Mainframes are proprietary systems.
2. Applications are not portable across multiple platforms.
3. Interfacing with heterogeneous systems is always a cumbersome work.
4. Difficult to upgrade or introduce new and / or better technologies.
5. Cost associated with technology upgrades.

With the easy availability of network access and the computer hardware price falling every quarter with increasing processing power, it should be possible to utilize the unused computing power of a vast majority of personal computers and servers for distributed computation. This will greatly improve overall response to Job processing requests; effectively utilize the unused computing power. The proposed framework addresses exactly these points.

Some of the benefits of this framework are:

1. A central monitoring component that provides a global view of all the Jobs under processing.
2. Dispatchers can get a global view of the availability of processors.
3. Dispatchers can choose alternate Processors if the target processor is loaded.
4. Processors can only process jobs and need not worry about re-scheduling.
5. Processors can be easily added and/or removed dynamically.

2. APPROACH

In this article, we will discuss about the approach and feasible implementations of a centralized monitoring system for Job scheduling and processing network. In such a system, there can be:

a. One or more monitors.

b. One or more processors.

c. One or mode Dispatchers.

The components (i.e. Dispatcher, Processor and Monitor) communicate over persistent message queues. Using a persistent message queue solves the problem of sequencing of messages and avoids problems of messages being lost when the network fails of systems crash. The status of a job is maintained in the persistence layer, a database. For simulation purposes, we are going to use MySQL Community Edition Server as the database.

To start with, let us consider the basic requirement of monitoring of a job scheduling system. The capabilities should include the following:

1. Processors should be able to report their availability.
2. Processors should be able to report their current load.
3. Every component (Dispatcher, Processor etc.) that handles a Job should be able to report the status of the Job.

4. The status should be updated and be available at a central location.

3. DESIGN

Let us consider feasibility of implementation of such a monitoring system. The following are various components within the system.

3.1 Job Dispatcher

This is the component that accepts the job requests (manual or otherwise), validates them and places the jobs in the Job Queue for processing. The dispatcher also records all the requests in the Database.

3.2 Job Processor

The processor is the component that picks up a job request from the queue, processes it. As shown in the diagram, the processor also reports the progress and status of job processing to the monitor. If a job is a long running job, progress information is sent at periodic intervals to the monitor. The Job processor also needs to report its health status back to the monitor. This is achieved through an independent thread in the job processor. Irrespective of whether a job processing is being done or not, the Heartbeat thread sends out the information about the availability and readiness of the processor. This helps monitor and dispatcher take intelligence actions on various aspects (describer later).

3.3 Job Monitor

This component is responsible for monitoring the status messages and updates the database. The component watches the progress messages and Heartbeat messages from various processors and saves the status in the database. This information also acts as feedback to the Job Dispatchers to take some decision at the time of dispatching the job to a target processor.

3.4 Dispatch Queue

This is the message queue that stores the job requests dispatched until a processor picks them up for processing. Note that, for reliable job processing system, this Queue should have persistence capability, so that, in case of system failures, the requests lying in the queue are not lost.

3.5 Progress / Status Queue

These are the message queues that store the job status sent by either dispatcher or processor. The monitor continuously monitors this queue for Job Status as well as Processor status messages. The information should include the current load, job status etc. This information is gathered by the Monitor and made available to the Job Dispatcher. The Job Dispatcher can then take intelligent decision based on this information to decide if a new job is to be dispatched to a target Job Processor or al alternate processor.

3.6 Database / Persistence

This is the most critical component in the entire system. All the information about the Job, the Processors, the state of processing and the availability of processors are maintained at a central database. This helps the job dispatching tasks little intelligent (as described later) and helps near real-time reporting of the progress of the job processing as well as health of the entire system.

The proposed system also takes into account important design aspects that greatly enhance the Job processing. They are:

a. Processor Affinity

b. Priority Thread Pool

3.7 Processor Affinity

The proposed system provides for defining a target processor for a given job. While it does not restrict any processor from picking up a job from queue for processing, the provision to specify a target processor helps design special purpose processors for specialized jobs. The dispatcher can read the Job definition to check if it can be dispatched to any processor of its choice or any specific processor(s). The job designer, of course, should be aware of the special purpose processors available and their capabilities. Once identified, the Jobs can be defined as such. The dispatched can then choose the appropriate processor while dispatching the job.

3.8 Thread Pool

We introduce here another important component in our design. The processor is designed to

As can be observed, we have not considered the traditional Sender/Receiver paradigm of design. The advantages of our Dispatcher / Processor design over the Sender / Receiver model is explained below. But, before that, let us understand the inefficiencies present in Sender / Receiver model.

In the traditional Sender / Receiver model (we will call each such component as a node), there is only one node that acts as either a Sender or a Receiver. Based on the current load level crossing the threshold values, either Sender changes itself to a Receiver or Receiver changes itself to a Sender. In order that such a system works correctly, all these nodes need to have the knowledge of all other nodes. This model has the following disadvantages:

a. The node is loaded with the responsibility of processing as well as job dispatching.

b. When a node receives a job and is 100% loaded, it needs to query the status of other nodes to find out if any other node is less loaded and can process this.

c. There is no clear distinction between types of different processors and each node is treated same. This means, there can be no processor affinity. Processor affinity, in a complex distributed network of processors, may be a desirable factor for specialized jobs.

d. When every node requests the status of other nodes periodically, the network overhead increases many fold.

e. In a complex network, having multiple sub-networks, configuring each node for locating other nodes is a complex task.

f. Assuming, the nodes use broadcast to announce their status, this also causes enormous load on the network.

g. Quite a good amount of time is wasted at each node to query other nodes. This time could have been utilized for processing the job.

How does our new model address these concerns? Here are the advantages.

a. There is a clear distinction between Sender (Dispatcher) and Receiver (Processor). Processors only do processing of the Jobs dispatched to them and report the status. When a job is about to be dispatched, the Dispatcher analyses the status of all the processors and takes the intelligent decision about the best processor available.

b. Processors report status to a central monitor at a configurable interval and there is only one way communication. Dispatcher need to query the central persistence (database) to check the status. This avoids nodes sending a request for status and other nodes responding with the status. This is a huge saving on the network usage.

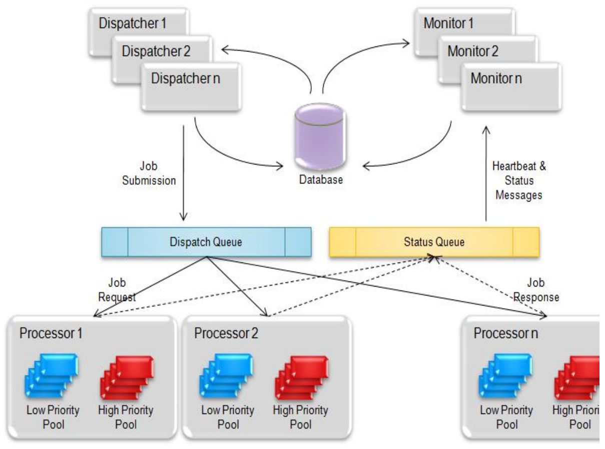
c. Any number of processors can be added and/or removed dynamically to the system without the need for configuration anywhere. Thus, the system has the ability to easily scale horizontally.

d. Each processor maintains it's internal Thread Pool based on the priority. The pool size is configurable. Thus, on a high end server, the same processor can be configured to handle more loads. This allows the system to easily scale vertically.

e. Each Processor can be assigned an ID and thus, Processor affinity of a Job can be defined.

f. Using a standard Message Queue with persistence, helps the system retain the messages during a crash and subsequent recovery.

g. Processors utilize the time only for processing and need not have to be burdened with the decision of re-distributing the job when they are loaded. This situation will not arise because the dispatcher would have considered the load situation and distributed the job to the best processor which can immediately pick up the job for processing (assuming not all processors are 100% loaded). The job is dispatched only once. This minimizes wait of the jobs as well as network delays.



The **Figure 1** represents a Job Processing network with monitoring capability. The dispatcher reads the Job definition, identifies its target processor. The target processor can be any processor or a specific processor. Having identified, it checks if the target processor is available and its current load. If the Job can be handled by multiple processors, the dispatcher finds the processor that is least loaded. This information is available in the database at a central location. After the potential processor is identified, the dispatcher sends the request to the target processor through the Dispatch Queue. Any number of dispatchers can be invoked from any location without having any conflicts. When a job is submitted, a unique identifier is assigned to it and the information is logged into the database.

The Job Processor monitors the Dispatch queue for new jobs. The processor also has two thread pools for processing jobs. The two pools are Low Priority pool and High Priority pool. The design is flexible enough to have any number of pools for any number of priority levels. Once a Job is available, the processor checks its priority and puts the job into the corresponding internal processing pool. The threads pool manager then takes the job and starts processing.

The Processor also has a Heartbeat thread that sends out heartbeat message to the Monitor at regular intervals. The heartbeat message includes the target processor id, the current load for respective priority pools and heartbeat interval. At the end of processing, the status of the job is also communicated to Monitor. Both Heartbeat messages and Job status messages are sent through the Status queue.

The Monitor regularly checks for messages from status queue. If a Job status message is received, it updates the status of the job in the database. If a heartbeat message is received, it updates the processor status in the database. The Processor status is thus kept current through the heartbeat message. Therefore, when a dispatcher is about to dispatch a job, it can easily check if the target processor is available and its current load and take appropriate decision to choose the right processor for the job.

The most important aspect of this design is the plug and play nature of the components (i.e. Dispatcher, Processor, Monitor). Such a system can be implemented over a vast wide area network having many smaller sub-networks. Any number of processors can be added independent of each other. Similarly any number of Dispatchers and Monitors can be added. While a single monitor is sufficient to handle load for hundreds of Processors, for handling failures, multiple monitors may be started.

Another important component in the entire system is the Database. The database is the central persistence that maintains the information about the Processors, their availability, Job definition, Job Execution status etc. It is recommended that a relational database like Oracle or SQL Server or MySQL be used for large Job processing systems. For simulation purposes, we used a MySQL database.

Since the Database is the central component that holds such critical data, it is also a probable single point of failure. This means, if the database system is down, the entire Job Processing system comes to a grinding halt. However, Clustering and Failover Recovery technologies available today can be used quite effectively to address such failovers scenarios.

All other components, using industry standard message queues can be easily replicated to address failover requirement without the need for any additional technology implementation.

4. SIMULATION and ANALYSIS

For simulating our design, we implemented a Java based job processing system with multiple processors, monitors and dispatchers. As part of this experiment, we defined a Job that compute 400000 prime numbers. Thus, the Job processing time was allowed to take whatever time it takes to compute. The wait time, process time and Total time of the jobs were monitored.

Table 1: Results

Priority	Wait Time	Processing Time	Total Time
Sender	231217	112141	343358
Proposed Algorithm	193129	109345	302474

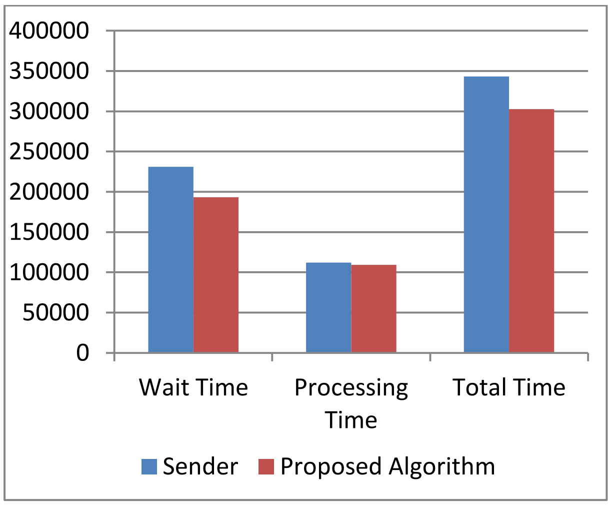


Figure 2. Results.

The architecture can be easily implemented in a network of processors. The processors need not necessarily be of identical capability in nature. The health and load of the entire processor network is available to any component in the network. The dispatchers can utilize this information for efficient routing.

The algorithm used to determine the least loaded processor for dispatching a Job request is given below.

```
TargetNode = RequestNode
READ Alternate Targets From DATABASE
FOR EACH Alternate Target
  IF Target IS NOT AVAILABLE Continue
  IF Target Load Is Minimum
    TargetNode = Target
    BREAK
  END IF
END FOR
IF NO Target IS RUNNING
    ABORT JOB
ENDIF
MARK TARGET FOR JOB AS TargetNode
DISPATCH To TargetNode
```

5. COMPARISON

The results were compared with the data collected through an implementation of Sender initiated algorithm. The network overhead in the Sender Initiated Algorithm was quite enormous and it increased the waiting time of the jobs in case of sender initiated algorithm. The results show that at-least 11% improvement in the total processing time in our proposed approach.

So far, in most of the systems implemented, the mechanism and protocol of communication between senders and receivers are not explained in detailed manner. This may lead to ambiguity in defining the overhead associated with the Sender initiated algorithms and/or Receiver initiated algorithms. The approach described here eliminates that ambiguity and also eliminates the overhead of processors participating in routing of jobs. This also keeps the architecture and implementation of such a system simple, dynamically scalable and flexible. This is an important aspect of requirement of a large array of networked Job processing systems.

6. CONCLUSION

The architecture presented here is a quite flexible and adaptive Job Processing network with a Central Monitor. This avoids every processor (or sender/receiver) having to be concerned about identifying the load of other processors and routing the job requests. This minimizes the processing overhead on the processors and communication/network overhead on the network.

The architecture can further be enhanced to include recovery of Jobs under processing at the time of a processor crash. This implementation will make the architecture a completely safe and reliable Job Processing network.

7. GLOSSARY

Word	Meaning
MQ	Message Queue
JMS	Java Messaging Specification
Active MQ	Industry standard, free Messaging System

8. REFERENCES

[1] Ambika Prasad Mohanty (Senior Consultant, Infotech Enterprises Ltd.), P Srinivasa Rao (Professor in CSC, Principal, YPR College of Engineering & Technology), Dr A Govardhan (Professor in CSC, Principal, JNTUH College of Engineering), Dr P C Rao (Professor in CSC, Principal, Holy Mary Institute of Technology & Science), Framework for a Scalable Distributed Job Processing System.

[2] Ambika Prasad Mohanty (Senior Consultant, Infotech Enterprises Ltd.), P Srinivasa Rao (Professor in CSC, Principal, YPR College of Engineering & Technology), Dr A Govardhan (Professor in CSC, Principal, JNTUH College of Engineering), Dr P C Rao (Professor in CSC, Principal, Holy Mary Institute of Technology & Science), A Distributed Monitoring System for Jobs Processing.

[3] J. H. Abawajy, S. P. Dandamudi, "Parallel Job Scheduling on Multi-cluster Computing Systems," Cluster Computing, IEEE International Conference on,

pp. 11, Fifth IEEE International Conference on Cluster Computing (CLUSTER'03), 2003.

[4] Dahan, S.; Philippe, L.; Nicod, J.-M., The Distributed Spanning Tree Structure, Parallel and Distributed Systems, IEEE Transactions on Volume 20, Issue 12, Dec. 2009 Page(s):1738 – 1751

[5] David P. Bunde1, and Vitus J. Leung, Scheduling restart able jobs with short test runs, Ojaswirajanya Thebe1, 14th Workshop on Job Scheduling Strategies for Parallel Processing held in conjunction with IPDPS 2009, Rome, Italy, May 29, 2009

[6] Norman Bobroff, Richard Coppinger, Liana Fong, Seetharami Seelam, and Jing Xu, Scalability analysis of job scheduling using virtual nodes, 14th Workshop on Job Scheduling Strategies for Parallel Processing held in conjunction with IPDPS 2009, Rome, Italy, May 29, 2009

[7] Y-T.Wang and R.J.T.Morris. Load Sharing in Distributed Systems. IEEE Trans. Computers, Vol. C-34, No. 3, 1985, pp. 204-215

[8] Pravanjan Choudhury, P. P. Chakrabarti, Rajeev Kumar Sr., "Online Scheduling of Dynamic Task Graphs with Communication and Contention for Multiprocessors," IEEE Transactions on Parallel and Distributed Systems, 17 Mar. 2011. IEEE computer Society Digital Library. IEEE Computer Society

[9] Sunita Bansal, and Chittaranjan Hota, Priority - based Job Scheduling in Distributed Systems, in Third International Conference (ICISTM 2009), Ghaziabad, INDIA, Sartaj Sahani et al. (Eds.), Information Systems and Technology Management, Communications in Computer and Information Science Series, Vol 31, pp. 110-118, Springer-Verlag Berlin Heidelberg, March 2009.

[10] Y-T.Wang and R.J.T.Morris. Load Sharing in Distributed Systems. IEEE Trans. Computers, Vol. C-34, No. 3, 1985, pp. 204-215.

[11] Nazleeni Samiha Haron , Anang Hudaya , Muhamad Amin , Mohd Hilmi Hasan , Izzatdin Abdul Aziz , Wirdhayu Mohd Wahid, “Time Comparative Simulator for Distributed Process Scheduling Algorithms”

[12] Java Message Service (JMS): http://java.sun.com/products/jms/