Priority based 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 proposes a framework for implementing a Priority Based Job processing system that has the capability to specify the priority of a job at the time of submission, execute the job at as per the priority at the job processor end. The framework will scale horizontally as well as vertically without any change in the components. This feature is achieved through simple configuration. The advantages of such a framework over other implementations is that we are using a global queue where in the processors can be dynamically added or removed without affecting the overall processing of jobs. This makes it flexible enough for any processor to handle any type of job without any restriction. While, it is still possible to impose restrictions on specific processors handling special type of jobs that is implemented as a configuration option and does not in any way impose restrictions on the processors. Therefore, new types of processors can seamlessly added to the entire network of processors without affecting the existing processors.

General Terms
Distributed Job Processing, Load Balancing,Parallel Processing.

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

1. INTRODUCTION
There are several Job/Batch processing systems implemented over the past decades, majority of them being on Mainframe systems. While such systems have met the functional requirements, the usage of them has been mostly restricted to offline / batch mode operations. Few of such systems are Payroll Processing systems, Back office systems in a financial institution for interest computation etc. Such systems, having lengthy processing time, processing large volumes of data, never had the requirement for a real-time computation or priority based computation.

Several batch / job processing systems do exist today, but they may not have priority based scheduling implementation. Some of the obvious problems are:

1. Some of the system may not have the capability to define a priority.
2. Not many levels of priorities are supported.
3. No clarity on how priority is managed.
4. It is not clear which component manages priority (Dispatcher or Processor)

While such challenges do exist today, given the technology available today, it is not difficult to implement a Job Processing system that supports priority based scheduling. There are several challenges in implementing such a system. So, how do we build such a system?

Some of the critical challenges are:

1. Defining a priority.
2. Handling of the priority by the processor.
3. Technology support available in implementing such a mechanism.

2. APPROACH
In this article, we will discuss about an approach and feasible implementations of a priority based job processing system. It is assumed that there is more than one processor available, but not necessarily online, in the system. By 'not necessarily online', we mean that a system is capable of processing the job, but is currently not available and will be available in the near future. Also, since the entire system has the reporting capability, is does have its own persistence, possibly through a local or remote database. So, the status of a job is maintained in the persistence, a database. The reporting can be done from the data available in this database.

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

1. Should clearly define the priority levels.
2. There should be a mechanism to assign priority to a Job.
3. The processor should be able to handle job processing requests based on the priority.

3. DESIGN
Let us consider feasibility of implementation of such a system. How can the priority handling be implemented?

Solution 1: The processors can be classified based on the priority. This means:

a. Each processor is identified with a priority; apart from its other attributes.
b. The dispatcher has the information about all the processors and their priority levels.
c. When a job is about to be submitted, the dispatcher identifies the job's priority, identifies the processor with the appropriate priority.
d. The dispatcher then checks if the processor is free to take up this job. It then dispatches the job to the appropriate processor.

e. Processor then takes up the job.

The solution has some advantages. The processors have pre-defined levels of priority. Thus, each processor deals with only the job that matches its own priority. Implementing this is simple too.

While this is quite a feasible solution, this approach has few drawbacks. Assigning priority to the processors means that we need to have pre-configured processors with respective priority levels. But, what if few of the processors go down and are not available? This will result in queue getting built up and jobs lying in the queue.

What if we design a processor that handles priority jobs of all levels? Based on the load it can take up any job and process it. Our second approach discusses exactly this implementation.

Solution 2: Let us consider that all the processors are capable of handling any job and of any priority.

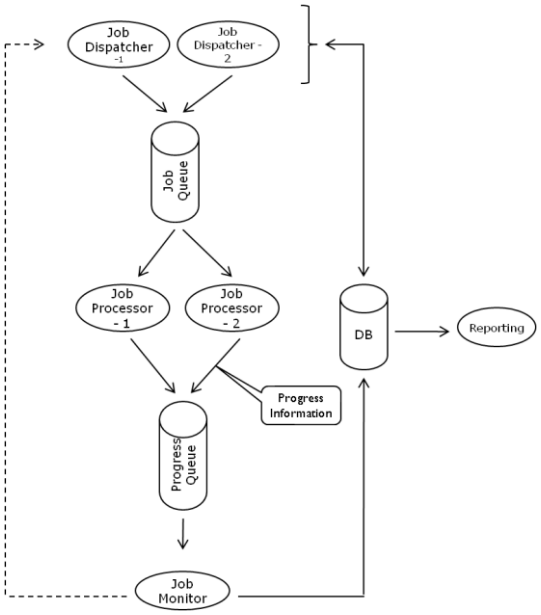

Fig1: Job Processing Flow Chart

In such a scenario, the system will have the following features:

a. Dispatcher can dispatch a job to the request queue, without bothering about the priority.
b. The processor is capable of handling jobs of any priority.
c. The processor internally, maintains independent thread-pools for different priority jobs.
d. Based on the priority, the processor assigns the job to appropriate pool.
e. The threads in a given pool have pre-defined priority, i.e. they are allocated CPU time based on the priority number assigned to them.

The solution 2 proposed above appears simple and feasible. Let us discuss in detail about how such a system can be implemented.

Job Dispatcher – This is the component that accepts the job requests from the external systems, validates them and places the jobs in the Job Queue for processing. The dispatcher also records all the requests in the Database.

Job 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. (Not all queues support persistence. For example, native message queue implementations in Unix systems do not support persistence. However, commercial message queues like Microsoft Message Queue, Active MQ, JBoss MQ etc support persistence.)

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. 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. 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.

The Figure 2 depicts the internal components of a typical Job Processor. The Job Processor primarily contains two threads. The first one is the main processing thread. This thread is responsible for handling the job processing request. For long-running jobs, this thread may also send out regular progress messages to the monitor through the Progress queue. While this is not mandatory, by suitable design of the progress message protocol, real time progress of a long-running job can easily be monitored. The second thread is the heartbeat thread. The responsibility of this thread is to send out messages to the monitor indicating that the Job Processor is active. The information also can include the current load, expected time to complete etc.

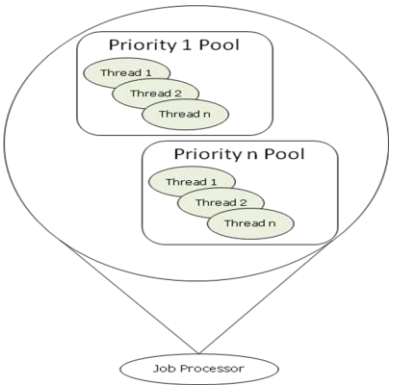

Figure 2. Job Processor with Priority Thread Pool

Progress / Status Queues – The progress or status queue is the message queue that receives messages with the information about the Jobs currently under processing. The processors, at periodic intervals, post these messages. The

monitor receives these messages and updates the persistence database accordingly. The final status of the job is also communicated to the monitor by the processor through this queue.

The Figure 3 shows the flow chart of the Job Processing steps that takes care of identifying the Job priority and assigns to the appropriate processing thread. The processor runs, it continues to send the progress messages at regular intervals. Thus, the Job dispatch components are always aware how many processors are active with their corresponding load and how many processors are not available.

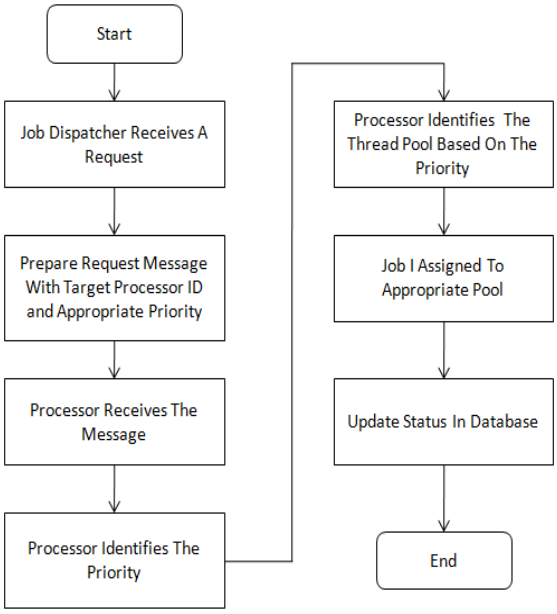

Figure 3: Job Processing System

For the implementation of a prototype, we used the Java environment for designing a priority pool based Job Processor. The core of the processor is the priority queues and the processor. Standard Java environment provides an internal queue named **LinkedBlockingQueue**. The declaration will be similar to the following.

```
LinkedBlockingQueue<Runnable> queue =

    new LinkedBlockingQueue<Runnable>();
```

The thread that needs to wait on this queue will be initialized as shown below.

```
ThreadPoolExecutor threadPool =

  new ThreadPoolExecutor(DEFAULT_POOL_SIZE,

                         MAX_POOL_SIZE,

                         keepAliveTime,

                        TimeUnit.SECONDS,
```

The DEFAULT_POL_SIZE and MAX_POOL_SIZE can be designed as per the application requirement, load to be supported and the resources available.

What is the advantage of the thread pool? The thread pool is internally managed through the respective queues internal to the processor. The processor can have multiple thread pools based on the priority. For example, a processor can have two thread pools, one for Low priority with 5 threads and the other for High priority with 10 threads. The biggest advantage of such a design is that when the processor is loaded with Jobs of one priority, other priority jobs can still be taken up. For example, if a Low priority job is submitted to a processor having 100% load with prior low priority jobs, then the newly arrived low priority job can be given to the high priority pool. Thus, there will be no starvation.

An instance of job execution component that actually handles the Job requests is assigned to this tool. As and when a Job request is received, the processor places the request in the appropriate Thread Pool based on the priority. A priority queue is always allocated to a job execution component. The component waits in a passive mode. As and when a request is placed in the internal priority queue, a new thread of job execution component is started up. The job execution component then picks up the processing request and executes the job. This is triggered by the code similar to given below.

```
TaskExecution task = new TaskExecution(newJob);

threadPool.execute(task);
```

where the 'task' is an instance of job execution client.

4. SIMULATION and ANALYSIS

Experiment 1

We implemented a Java based job processing system with various options. As part of this experiment, we defined a Job that compute 200000 prime numbers. Thus, the Job processing time was allowed to take whatever time it takes to compute.

1. The Job could take the priority as an attribute. The priority could be Low or High.
2. The dispatcher was able to dispatch the job to the alternate queues (i.e. first to Low priority queue, second to high priority queue, third to low priority queue and so on).
3. The processor was designed to have two independent thread pools, one for Low priority and the other was for high priority.
4. Each thread pool had the capacity to process 10 jobs concurrently, beyond which, jobs will wait in the queue.
5. When the Job processing was delegated to the appropriate thread, the thread priority was set to either low or high based on the Job's priority.
6. The job was configured to compute 200000 prime numbers.
7. A total of 40 Jobs were dispatched, 20 with low priority and 20 with high priority.

8. The wait time, processing time were measured for each job.
9. Finally, the average values were plotted as a graph as shown in **Figure 4**.

As can be observed, the total time for Low priority jobs are very high as compared to that for the high priority jobs when the number prime number computations are kept same at 2000000.

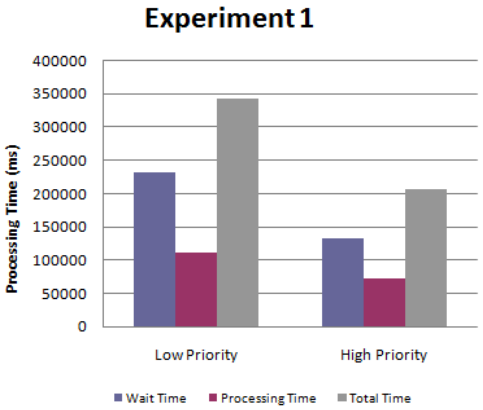

Figure 4 : Result of Experiment 1

Here is the data that was collected over several iterations and averaged out.

Priority	Wait Time (ms)	Processing Time (ms)	Total Time (ms)
Low Priority	231217	112141	343358
High Priority	133129	72955	206084

Table 1: Data collected as part of Experiment 1

Experiment 2

We then implemented a Java based job processing system with more options. As part of this experiment, we defined a Job that compute prime numbers for a fixed period of approximately 40 seconds. Thus, the Job processing time was fixed and we monitored how many prime numbers were computed.

1. The Job could take the priority as an attribute. The priority could be Low or High.
2. The dispatcher was able to dispatch the job to the alternate queues (i.e. first to Low priority queue, second to high priority queue, third to low priority queue and so on).
3. The processor was designed to have two independent thread pools, one for Low priority and the other was for high priority.
4. Each thread pool had the capacity to process 10 jobs concurrently, beyond which, jobs will wait in the queue.
5. When the Job processing was delegated to the appropriate thread, the thread priority was set to either low or high based on the Job's priority.
6. The job was configured to compute for approximately 40 seconds.
7. A total of 40 Jobs were dispatched, 20 with low priority and 20 with high priority.
8. The wait time, processing time and number of prime computations were measured for each job.
9. Finally, the average values were plotted as a graph as shown in Figure 5.

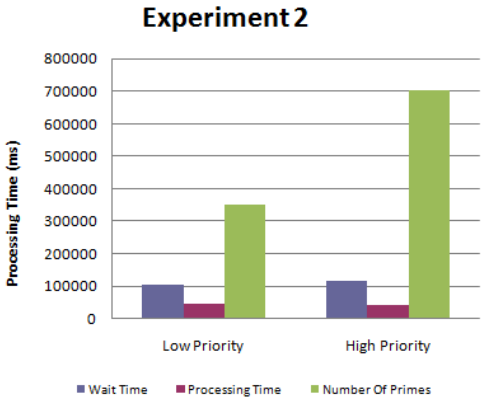

Figure 5: Result of Experiment 2

As can be observed, when the computation time was kept constant, the number of prime numbers computed was considerably higher (almost double).

Here is the data that was collected.

Priority	Wait Time (ms)	Processing Time (ms)	Number of Primes
Low Priority	104630	44014	349504
High Priority	116286	42012	701410

Table 2: Data collected as part of Experiment 2

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 figure below demonstrates at-least 12% improvement in the total processing time in our proposed approach.

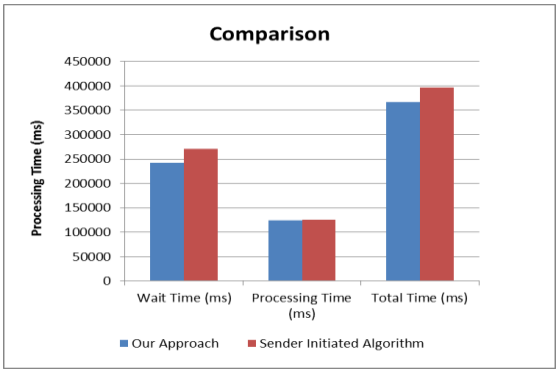

Figure 6: Comparison

6. CONCLUSION

The idea presented here uses the currently available technologies on Java platform to implement a priority based scalable, high performing, cost effective job processing system. This technique had the option to configure the processors and priority pools. The technology can be utilized to design enterprise level Job processing systems.

The proposed system can be enhanced further to include load balancing based on job priority and cost of routing and cost of processing. Such systems can be commercially quite viable in areas like web-based batch processing services, services provided over cloud computing platforms etc.

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