A Greedy Algorithm for Load Balancing Jobs with Deadlines in a Distributed Network

Ciprian. I. Paduraru
Department of Computer Science,
University of Bucharest
Bucharest, Romania

***Abstract*—One of the most challenging issues when dealing with distributed networks is the efficiency of jobs load balancing. This paper presents a novel algorithm for load balancing jobs that have a given deadline in a distributed network assuming central coordination. The algorithm uses a greedy strategy for global and local decision making: schedule a job as late as possible. It has an increased overhead over other well-known methods, but the load balancing policy provides a better fit for jobs.**

***Keywords*—scheduling; greedy; coordination; network**

I. Introduction

Distributed architecture of computers can represent the underlying of a web or network-based service used for processing user requests. In this context, the performance of the processing system is closely related to user experience and service availability, and can therefore play an important role in the success or failure of the respective service on the market. As sufficient hardware resources for processing a large number of requests are generally expensive, a good algorithm for the distribution of load - between the processing units in the distributed system - is necessary to save costs in addition to increase clients' satisfaction.

This paper proposes an algorithm for load balancing of jobs in a distributed network assuming central coordination. Jobs are non-preemptive, received by a single machine in the distributed network (master) and sent to workers. Each job has a given deadline which is assigned by the owner of the request. The master must decide if the job can be executed by one of the workers considering its deadline and an error window, and if it does, then who the best worker to execute it is. Once received by a worker, it must decide where on its own waiting list of jobs the new job should be added. The algorithm can work both for homogeneous and heterogeneous workers. It all depends on the ability of a worker to determine the execution time of a job. If workers can estimate how much time it will take to execute a given job within the considered error window then we can have heterogeneous machines in the distributed system. Various methods for doing such estimation are presented in [1]. One method is to assign to each job a length-class and test each worker in the system how much time it will take to execute each kind of length-class. There is a linear search overhead determining the best fit for a new job but it makes the load balancing better and provides better results. Also, we assume that there is a communication link between the master machine and each worker, and we can estimate the average communication time for each job. For simplicity, this communication time is included in the execution time of a job.

The rest of the paper is organized as follows: In Section 2 there is a discussion about research made on load balancing or scheduling algorithms with deadlines. Section 3 presents how the algorithm is designed and a pseudocode for its implementation. Results obtained from running a simulator over some test samples are given in Section 4. Conclusions are presented in the last section.

II. Related Work

At the time when this paper is written there is no paper dealing with load balancing tasks with deadlines in a distributed network with central coordination. However, there are various papers presenting techniques for load balancing / scheduling of tasks which are a point of inspiration and a possible comparison for the algorithm presented here.

Some theoretical aspects with high-importance for this paper are presented in [5]. A conclusion is that Earliest DeadLine First (EDF) policy is not optimal for non-preemptive tasks or when there are multiple processors in a system. [2] Presents a new algorithm for load balancing in grid architecture for fair scheduling. It addresses the fairness issues by using mean waiting time. It schedules the tasks by using fair completion time and reschedules them by using mean waiting time of each task to obtain load balance. In [3], there is a comparison between two important task schedulers such as EDF scheduler and Ant Colony Optimization Based (ACO) scheduler. Paper [4] presents a greedy algorithm for scheduling jobs with deadlines and profits with the main objective to maximize the profit, for a single processing unit.

III. Algorithm Design And Implementation

Jobs are received and sent further by the master machine. The algorithm doesn't move any job from a worker to another because each time when we give a job to a worker, we know that it can satisfy its deadline constraint. Also, as Section 1 states, jobs are non-preemptive. There are two separated views of the algorithm:

- Master view: responsible for assigning a new job to a worker if there is one available to execute this job satisfying its deadline constraint.
- Worker view: responsible for managing a data structure that holds jobs and extracting / executing these jobs.

A data type that defines a job can be defined as: *JobType*={timeToExecute, deadline, timeToStart, timeToEnd, dataContext}. *timeToExecute* is the time needed by an worker to execute the job, while *dataContext* is the data associated with the job execution. *timeToStart* represents the time when a job can start on a worker. *timeToEnd* is the computed value of timeToStart + timeToExecute.

A. MASTER VIEW

At this level, the main idea is to send a new job to the worker which can start it as late as possible but still satisfying the deadline constraint. It is a greedy solution which can keep workers available for earlier deadlines. Considering that function *GetTimeToStart* returns the time when a worker can start a job given as parameter (-1 is considered to be the return result for not being able to execute it and satisfy its deadline goal) then the pseudocode that master runs when a new job is received is presented below.

```
OnNewTaskArrived( JobType task)
  bestWorkerId = -1
  bestWorkerTime = 0
  foreach worker W do
  {
    Wtime = Controller[W]->GetTimeToStart()
    if  Wtime != -1 AND  bestWorkerTime < Wtime
    {
        bestWorkerTime = Wtime
        bestWorkerId = W
    }

    if bestWorkerId != -1
    {
        SendTask(job, bestWorkerId)
    }
    else
    {
        // Code for refused job
    }
  }
```

The *Controller* array is stored on master and represents the state of each worker – the data structure which stores informations about the currently assigned jobs for each client, excluding the “dataContext” field which is only needed by workers. This is actually logic part of worker’s view.

B. WORKER VIEW

There are two issues at worker’s view: the *GetTimeToStart* function implementation (called by the master and having its context data stored on the master in the *Controller* array) and how a worker manages its internal data structure to execute jobs.

Same greedy idea as in section 3.1 is used here: schedule a new job as late as possible (Figure 1). Workers are using a linked list to store the assigned jobs. In this linked list, jobs are sorted in ascending order by the value of field *timeToStart*. The value of this field for a new job should ideally be: deadline – timeToExecute, because the main objective is to promote free spaces for new jobs that have earlier deadlines. But if this is not possible due to existing jobs, then it should be set to the last gap found between assigned jobs that allows us to execute the new job and satisfy its deadline.

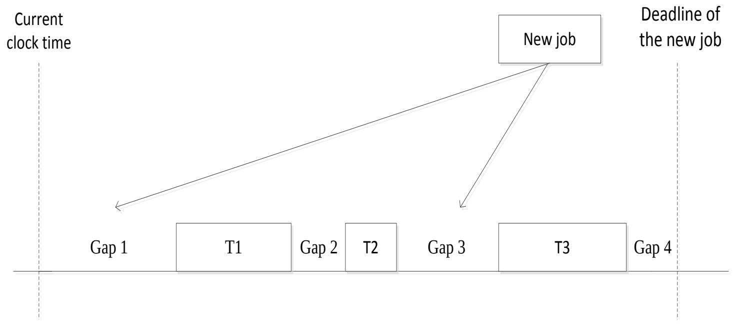

Fig. 1. Adding a new task to an existing list of jobs (T1, T2 and T3). Considering that the width of the rectangle represents the execution time of a job, then gaps with index 1 and 3 can fit the new job. The end of gap 3 will be preferred for the new job to schedule it as late as possible.

Pseudo-code for this operation is presented below. The

```
GetTimeToStart(job)
  // Step 1:
  // Check the back of the list first
  If mList.isEmpty()  OR
   mList.last().timeToEnd<=(job.deadlinejob.timeToExec)
  {
    job.timeToStart=job.deadline – job.timeToExec;
    mLastCachedPos = mList.end;
    return job.timeToStart;
  }

  // Step 2:
  // Check for gaps between tasks, starting from last to first
  foreach job T in mList (reverse order)
  {
    prevT = T->prev
    if  prevT == NULL continue

    deadlineImp=min(job.deadline,T.timeToStart)
    if prevT.timeToEnd+job.timeToExec<= deadlineImp
    {
      job.timeToStart = deadlineImp – job.timeToExec
      mLastCachedPos = position of T in mList
      return newJob.timeToStart
    }

 // Step 3:
 // Check for a gap between current clock time and first job
 //  begin
 Tfirst = mList.first()
 deadlineImp = min(job.deadline, Tfirst.timeToStart)
 if (clock() <= (deadlineImp – newJob.timeToExec))
 {
   mLastCachedPos = mList.first()
   job.timeToStart = Tfirst.timeToStart – job.timeToExecute
   return  job.timeToStart
 }
```

mList variable represents the linked list where the jobs are stored. *mList.end/mList.start* represents the last/first element in the list. If there is no other assigned job in the list or we can schedule the new job after the last one, then the ideal value for *timeToStart* will be deadline – timeToExecute. Otherwise, the algorithm tries then to fill the first gap found starting from the end of the list and going to its beginning. Each time we compare two consecutive elements in the list (T and *prevT)* and check if the new job can be added between the *timeToEnd* of prevT and the minimum between its deadline and the *timeToStart* of T. If we find such a position then we set the *timeToStart* as late as possible in this gap. Finally, if no gap was found yet, we try to add it in the gap starting from current clock time to the beginning of the first job in the list. The implementation of *GetTimeToStart* will also cache the *timeToStart*, *timeToEnd* and the position in the linked list where it should be added (mLastCachedPos). This information will be sent together with the job in the SendTask function to avoid doing the linear search twice.

On each worker there is a function which continuously polls for jobs in the jobs list and grabs them for execution. The trick is to allow grabbing and execution of the first job from the list even if the current clock time is less than its *timeToStart* field. Doing this will keep the machines busy. It is possible that some of the jobs with a deadline close to current clock time will be refused, but it has the same probability (assuming a normal distribution of jobs in time) that other new jobs will benefit from this.

The complexity of searching for the best place to add a new job inside a worker is linear in the number of existing jobs on that worker. The worst case happens where there are many jobs received in a short time interval while the jobs execution time is higher than the arrival time rate of new jobs.

IV. Simulation Results

To test the performance of the proposed algorithm, a simulation was made in order to see its behavior in comparison with other two load balancing algorithms. The first one sends the jobs in a round robin policy while the second one to the worker that can execute the job as late as possible. Both have just a simple policy at the worker's view: add the new job to the end of the queue if it can be executed before its deadline expires. Ideally, the load balancing should use the resources correctly by keeping the hardware busy most of the time and minimizing the number of refused jobs.

Final results were obtained by averaging a number of test samples which creates 1000 of jobs with a normal distribution of execution times between 10 and 200 milliseconds. The arrival time rate of new jobs was between 1 and 10 milliseconds. Deadline time extension of each job (time since a new job was received to when it should finish) was also chosen by a normal distribution in interval [10, 2000] milliseconds (considering the job execution time too). Samples where run on 24, 16, 12, 8 and 4 machines in a local network (workers) each having single hardware process dedicated for our job execution. The process of receiving and assigning a new job to a worker was done by a separate machine called master.

Figure 1. shows how many jobs where refused depending on the number of machines and the algorithm used. The results graph shows that the proposed algorithm is better than the other two methods, despite its overhead. The difference between it and the other two algorithms increases with the number of machines used. When using 16 machines, the number of jobs refused by the other two algorithms is with 49% higher than the proposed algorithm.

With 24 machines, the proposed algorithm succeeded to obtain 0 refused jobs while the other two solutions had 18/21 refused jobs. The samples used creates jobs in a short time interval (defined at the beginning of this section) to represent a worst case scenario for the proposed algorithm. When jobs have longer execution time and the arrival time has a different time distribution in the proposed algorithm can perform even better than this because there is less overhead spent on decision making.

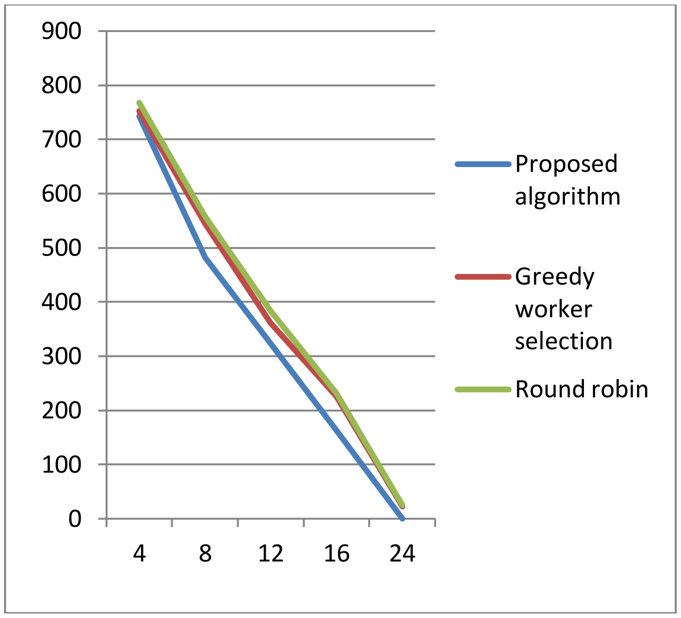

Fig. 2. The average number of refused jobs (1000 was the total number of jobs) in test samples by each algorithm.

V. Conclusion And Future Work

This paper presented a novel algorithm for load balancing jobs having deadlines in a distributed network with central coordination. By using two different greedy strategies, one from master view and the other from worker's view, the proposed algorithm provide an increased performance than the classical methods for load balancing jobs. Keeping the same hardware and being able to increase the performance with over 49%, as the results sections shows, represents an important issues for most of the web services on the market.

One important topic to study in continuation is to consider that each job also has a profit assigned and find a load balancing policy that maximize the profit instead of considering equal profits for jobs like the proposed algorithm does.

References

[1] Ciprian Paduraru, "A New Online Load Balancing Algorithm in Distributed Systems", 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, 2012

[2] U.Karthick Kumar, "A Dynamic Load Balancing Algorithm in Computational Grid Using Fair Scheduling", IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 5, No 1, September 2011

[3] M.Kaladevi and S.Sathiyabama, "A Comparative Study of Scheduling Algorithms for Real Time Task", International Journal of Advances in Science and Technology,

Vol. 1, No. 4, 2010

[4] Antonina Kolokolova, "A Greedy Algorithm for Scheduling Jobs with Deadlines and Profits", Scheduling case study, Lecture notes

[5] C. L. LIU and JAMES W. LAYLAND, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", Journal of the Associatlon ior Cornputmg Machinery, Vol. 20, No. I, January 1973

[6] Peter Brucker, "Scheduling Algorithms Fifth Edition", Springer, October 2006.

[7] Kirk Schloegel, George Karypis and Vipin Kumar, "A unified algorithm for load-balancing adaptive scientific simulations", Proceeding Supercomputing '00 Proceedings of the 2000 ACM/IEEE conference on Supercomputing Article No. 59 IEEE Computer Society Washington, DC, USA

[8] Abbas Karimi, Faraneh Zarafshan, Adznan b. Jantan, A.R. Ramli, M. Iqbal b.Saripan, "A New Fuzzy Approach for Dynamic Load Balancing Algorithm", (IJCSIS) International Journal of Computer Science and Information Security, Vol. 6, No. 1, 2009.

[9] Reinhard Lüling , Burkhard Monien, "A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance", Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, 1993.