INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL
ISSN 1841-9836, 9(6):749-757, December, 2014.

An Online Load Balancing Algorithm for a Hierarchical Ring Topology

C.I. Paduraru

Ciprian I. Paduraru
Computer Science Department
University of Bucharest
ciprian.paduraru2009@gmail.com

Abstract: Ring networks are an important topic to study because they have certain advantages over their direct network counterparts: easier to manage, better bandwidth, cheaper and wider communication paths. This paper proposes a new online load balancing algorithm for distributed real-time systems having a hierarchical ring as topology. The novelty of the algorithm lies in the goal it tries to achieve and the method used for load balancing. The main goal of the algorithm is to correctly utilize the computing resources in order to satisfy the average response time of clients. The secondary goal is to ensure fairness between the numbers of requests solved per client with respect to the average response time. A request from a client is moving through the network until a node considers that it can solve the request in the promised average time for that client or until it seems like the best opportunity to avoid any additional delays in solving it. A performance analysis and motivation for the proposed algorithm is given with respect to the goals it tries to achieve. The results show that the proposed algorithm satisfies its goals.
Keywords: ring; hierarchical; distributed; balancing; algorithm; fairness

1 Introduction

Today, the common approach for processing user requests sent to a web or network-based service is to handle them using a distributed architecture of computers. 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 client's satisfaction.
This paper presents a load balancing algorithm for hierarchical ring network. A hierarchical ring (Figure 1) is an alternative to 2D meshes or tori [5]. Hierarchical type was chosen to show the generality of the algorithm. Instead, we can have rings combined with other network topologies. In a ring network every node has exactly two neighbors: P_i is connected P_{i+1} to and P_{i-1}. In this paper, if we consider that n is the number of processors in the ring, then we assume that all additions on processors indices are done modulo n. In the hierarchical ring network considered, each sub-network has a leader which is responsible to store information and coordinate some activities. In the continuation, when we refer to a sub-network of a leader node then this includes only the direct nodes under the leader's level.

Requests are received by a web service which coincide with the leader node of the network - and are considered to have an estimated average time to complete. The goals of the presented algorithm are the most significant for services provided in the present. The main goal is to ensure a certain average response time given for each client, depending on what we call a user license. The user license can be interpreted as a contract between the service provider and the user, where parameters referring to the delivery of the service are specified. These include parameters

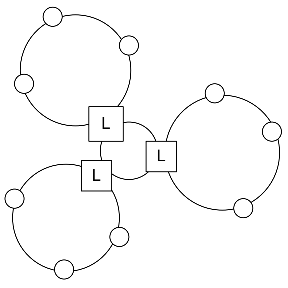

Figure 1: Two levels ring hierarchy.

relevant for load distribution, like the average response time of the system for certain types of requests. The secondary goal is to ensure fairness between the numbers of requests solved per client with respect to the average response time. A simulator has been created to demonstrate how the algorithm succeeds to satisfy the desired goals.
The rest of the paper is organized as follows: In Section 2 there is a discussion about research made on load balancing for ring topologies or other network types but appropriate to our goals. In Section 3 the design of the load balancing algorithm is discussed. If first starts with the assumptions made over the proposed algorithm then it describes the main ideas and pseudocode behind the decision making process. Section 4 shows the simulation results compared with a general load balancing for a hierarchical ring network . Conclusions are given in the last section.

2 Related work

A description of hierarchical ring networks is given in [5]. They are presented as an interesting alternative to popular direct networks such as 2D meshes or tori. Advantages of using them are also described here: simple router designs, wider communications paths and faster networks than their direct network counterparts. However the paper is not dealing with load balancing algorithms. Its a study to determine how large hierarchical ring networks can become before their performance deteriorates due to their bisection bandwidth constraints.
There are not many papers discussing about real time load balancing for ring networks. The most appropriate paper for our presentation is [1]. In comparison with [1], which is a general load balancer for rings, the new proposed load balancing algorithm has another two goals: satisfy the average response time specified in the owners license type and ensure some fairness between the requests with respect to their specified response times. Other papers, like [6], are performing a statical load balancing of requests on a ring network. Paper [2] presents a load balancing algorithm for distributed systems having the same two goals. However, the algorithms presented there are inapplicable to the ring networks. It uses the fact that nodes can communicate directly with a master and it would be too much overhead to simulate the same implementation algorithm on a ring. Both the requests and results are exchanged directly from master to workers.

3 Design of the algorithm

3.1 Assumptions

It is assumed that when a node finishes a request, the results are sent back to clients directly from that node. The algorithm allows for the system to be heterogeneous, workstations may differ in processing capacity. The processing time of a request is expressed as the "request's length" and can be predetermined. We assume that **GetEstTimeToCompute(request)** returns the estimated processing time of a request on any node and its time complexity is constant. One simple way to do this is to benchmark how fast each node can execute different requests length intervals, then group and store these results in a data structure on the node. It is considered that request processing is workstation independent (all types of requests can be processed by any of the nodes). Requests are independent (the order in which requests are processed does not affect the correctness of the result) and indivisible (can only be handled by a single worker at one moment). The communication time is not generally important for the algorithm. The reason is that while a request spends time moving through the network its priority increases.

3.2 High level implementation and the communication protocol

The main responsibilities of nodes are to take decisions, solve requests and communicate with neighbors. The communication and request's solving should run in different threads to avoid communication blocking. Requests are received by the leader of the ring and send further until a node can execute it in the required time or when that node is a good opportunity to save additional delay in response time. Nodes evaluates if a request can be executed by them or not depending on the time needed to complete all other requests waiting there and having a higher priority than the considered request. Also, in the case of requests that are close to their deadline (or already passed deadline), if they have a higher priority than all other requests waiting on a node then we choose that node to minimize the additional delays in the response time.
By using the above two conditions there is a possibility that all nodes to decline solving a new request. In this case measures need to taken in order to avoid affecting the performance of the system with the new request running too many times through the ring. The method used is to have a variable on each request that represents a bonus time considered when a node evaluates if it has enough time to execute the request. Each time the request goes back through the node that initiated it, the leader of the ring, this variable is incremented by some value determining the nodes to accept it faster.
In the continuation of this section these ideas are presented in more details. It starts with the high level operations and messages exchanged between nodes, then it continues to explain the implementation of data structure and decision making in more details using pseudocode and complexity analysis.

To send data between nodes, two functions are used: **Send**(data) used to send data to the next node on the same sub-network and **SendToSubNetwork(data)** which sends a message from a leader to its coordinated sub-network (this helps moving a message from a higher level network to a sub-network). Another important function is **IsLeaderNode** which has two prototypes. The first one doesn't have any parameters - tests if the node is the leader of a sub-ring - and the second one with a parameter representing a message - tests if the node is a leader and the one who created/added that message in its sub-network.
There are two types of messages used in the communication protocol: **Gather** and **Request**. A Request message is used for sending requests between nodes and contains the following: data context for request execution, the average response time specified in the owners license, timestamp when created, the time when should ideally finish and the current bonus time. The code below shows the high level implementation of the decision making when a Request message is received by a node. A node that is not the leader at the level where a request message is sent can either store the request for later execution or send it further in the same network level. Additionally, a leader node has the option to send the request down in its sub-network.
Users might also want to relax the conditions and not decrease the bandwidth performance with requests that are travelling the ring many times in order to find a node that accepts them

(BONUS_STEP variable is considered as input given by user). To make this possible, when a leader receives back a request that it previously sent to its sub-network, the bonus time variable on the request will be increased. An interesting property of this communication protocol is that if a request travels again back to the leader node of a sub-network because of the high workload, it can eventually get to another sub-network, if the leader evaluates that it is better to do so.

```
OnRequestReceived(request)
   if (CanExecuteRequest(request))
      AddRequest(request);
   else if (IsLeaderNode(request) AND CanExecuteOnMySubNetwork(request))
   {
      if (already received this request)
         request.bonusTime += BONUS_STEP
      SendToSubNetwork(request)
   }
   else
      Send(request);
```

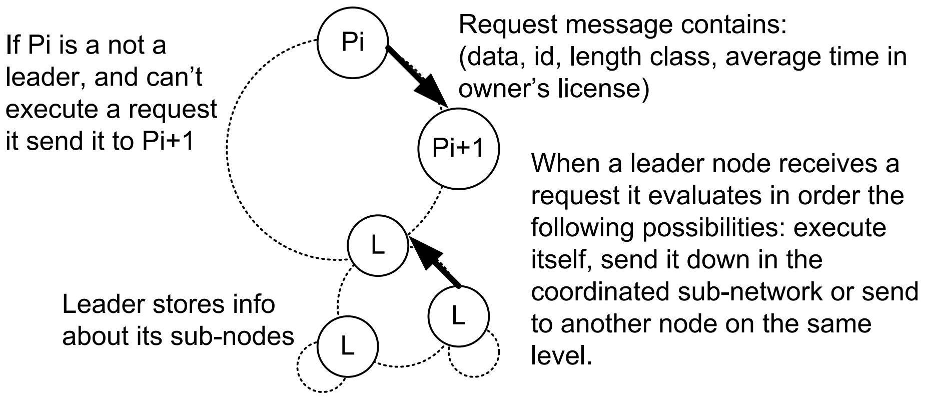
If Pi is a not a leader, and can't execute a request it send it to Pi+1
Pi
Request message contains: (data, id, length class, average time in owner's license)
Pi+1
When a leader node receives a request it evaluates in order the following possibilities: execute itself, send it down in the coordinated sub-network or send to another node on the same level.
L
L
L
Leader stores info about its sub-nodes

Figure 2: High level decision making for a new request.

A **Gather** message is initiated by leaders of each sub-network at fixed time periods and sent only to the nodes on the same level. The role of this message is to have a snapshot of the current load inside nodes. This information will be used to take a decision if a leader node should accept a request or not to be executed in its sub-network. The gather messages are asynchronous between different sub-networks. When a node receives a gather message, it adds its local load information (like how much load is in there) to the message and sends it further. When the message is received by the leader it updates its load information table. Figure 3 is representative for this flow.

The code below presents the action code of every node and the handler function for receiving Gather messages. *lastTimeGatherSent* is a variable where we store the timestamp of the last Gather message sending occured. *T* is threshould value set by the user, depending on how often he wants to send the Gather message. *Solve* function is supposed to run the effective job on the request. Function *ExtractNextRequest* selects the task with the highest priority from the local list of tasks.

```
OnUpdate()
   if (IsLeaderNode() AND (GetCurrentTime() - lastTimeGatherSent) > T)
```

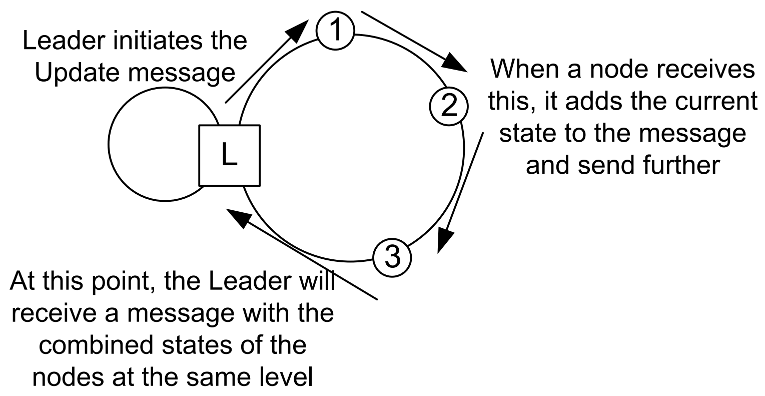
Leader initiates the Update message
1
2
3
L
When a node receives this, it adds the current state to the message and send further
At this point, the Leader will receive a message with the combined states of the nodes at the same level

Figure 3: An update message in the ring.

```
   {
      lastTimeGatherSent = GetCurrentTime()
      Gather msg
      SendToSubNetwork(msg)
   }
   request = ExtractNextRequest()
   if (request != null)
      Solve(request)

OnGatherReceived(msg)
   if (IsLeaderNode(msg))
      UpdateLocalState(msg)
   else
   {
      AddLocalState(msg)
      Send(msg)
   }
```

3.3 Decision making to execute a request locally on a node (leaf or leader)

To make computations easier, the average response times specified in the clients licenses are normalized. If $t_1, t_2, t_3, ..., t_n$ are the average response times and $t_{max} = \max t_i$, then $\overline{t_k} = \frac{t_k}{t_{max}}$. Requests are stored and evaluated based on their priorities. The priority of a request is defined as the waiting time for the request to be solved divided by the inverse of the normalized average response time specified in the owners license. If we denote with *WaitingTime(request)* the waiting time of the request to be solved, and *Owner(request)* the index of the owner license then the priority computation can be written as $\textit{Priority(request)} = \frac{\textit{WaitingTime(request)}}{\frac{1}{\overline{t_{Owner(request)}}}}$, where *WaitingTime(request)=CurrentTime()-request.createTime*. Priorities of requests waiting on a node are dynamic and could modify in time. This is a key point of the algorithm which gives the fairness between the clients with respect to their average response times. The formula used also helps in the case of requests that travel in the network for a long time. The priority of a request increases with its waiting time regardless of the license's specifications. If a requests travel a long time then it has a bigger priority and its chances to be added in a node are increased. Requests are stored in a node using a linked list. At each query of function CanExecuteRequest(request) the algorithm iterates over the existing items and sum up the time needed to compute all requests that have a priority higher than the new request. Using the bonus modifier and comparing the result with the average response time of the request's owner we can find out if the request can be executed on that node or not. Also, in the case of requests that are close to ideal execution deadline or should have been executed until now, we check if their priorities are higher than all other priorities waiting in the node. If this is valid then this node is good fit for the request because it will be the first request selected for execution, thus minimizing the delays in response time. A pseudocode for this is given below. *requestsList* is storing requests of a node. *request.bonus* represents the bonus time given by the leader, while *request.idealTimeOfFinish* is the precomputed time when the request should be solved in order to satisfy the owner's license specfication. T is a threshold value defined by user. It could be either the average time for moving data between consecutive nodes, the average time needed for processing it, or a heuristics combining these.

```
CanExecuteRequest(request)
   totalEstTime = 0;
   newEstTime = GetEstTimeToCompute(request) / request.bonus
   bestPriority = null;
   foreach req in the requestsList
   {
      if (Priority(req) > Priority(request))
      totalEstTime = totalEstTime + GetEstTimeToCompute(req);
      if (Priority(req) > bestPriority)
      bestPriority = Priority(req)
   }
   remainingTime = request.idealTimeOfFinish - (GetCurrentTime() + totalEstTime)
   isCloseToDeadline = (request.idealTimeOfFinish - GetCurrentTime()) <= T
   return (remainingTime >= 0 OR
            (isCloseToDeadline AND bestPriority < Priority(request)*request.bonus)
```

3.4 Decision making to execute a request on a sub-network

This type of decision is valid only for leader nodes. In order to make this possible the gather messages are sent in the sub-network in order to collect workload information. In an ideal case, a leader would know informations about all waiting requests in its sub-network nodes and run the same CanExecuteRequest function. But such a message would be too large creating a bandwidth and processing time overhead. A tradeoff solution between performance and quality of the decision result is to gather statistics on how much time the current requests would take to execute on different intervals of priorities. If P_{high} and P_{low} are estimated bounds of the priorities, then splitting on N equal intervals would result in $TimeSum_i$ storing the sum of times to solve the requests with priorities in interval $\left[P_{low} + \frac{P_{high}-P_{low}}{N} * i, P_{low} + \frac{P_{high}-P_{low}}{N} * (i+1) - 1\right]$. The tradeoff can be adjusted using variable N.

The gather message will contain the *TimeSum* array. When adding the local state to a gather message, a nodes responsibility is to iterate through all its waiting requests and for each one to add in the corresponding array index (the correct priority interval) the time needed by the node to solve it. The leader will keep the final *TimeSum* array and use it for decision making. To find out if a request can be executed by the leaders sub-network we need to sum up the values of all intervals of greater priority than the considered request. Then, we divide this sum to the number of nodes in the sub-network to find out the average time needed to finish all higher priority requests. The close to ideal deadline test is used here too, but this time we check if there are any values bigger than zero on intervals with greater priority than the considered request. Below is presented the pseudocode for adding a local state to the gather message and the decision making of a leader if it should accept or not a request in its sub-network. *GetPriorityInterval* does simple math to get the interval index from the priority of a request.

```
AddLocalState(message)
   foreach req in the requestsList
      message.TimeSum[GetPriorityInterval(req)] += GetEstTimeToCompute(req)

CanExecuteOnMySubNetwork(request)
   P = Priority(request)
   totalTime = 0;
   for i = P +1 to N
      totalTime += TimeSum[i]
   averageTotalTime = totalTime / NumNodesInSubNetwork
   remainingTime=(request.idealTimeOfFinish - (GetCurrentTime() + averageTotalTime))
   isCloseToDeadline = (request.idealTimeOfFinish - GetCurrentTime()) <= T
   return remainingTime>= 0 OR
           (isCloseToDeadline AND there is no TimeSum[k]>0 with k from P+1 to N)
```

The complexities of the operations used here are linear which can be good or bad depending on request's granularity. If there are generally very small requests to execute then this linear time might affect the global performance. An idea to solve this case would be to use a heap tree data structure (which provides logarithmic time for operations) and to rebuild the tree at different time intervals considering the newest priorities.

4 Simulation results

To demonstrate that the algorithm satisfies the proposed goals, a simulator in MPI has been created. The nodes are processes on different machines connected in a network. The test implies a total of 64 processes over 8 machines. Random requests where continuously generated with a normal distribution in length classes. The estimated times to compute the requests were between $[10, 500]$ milliseconds (depending on the computing power of the nodes). Same interval was used for the average response times in the clients licenses. Two relevant tests are used to show how the load balancer works. The results are compared to the results of the algorithm in [1].

Test 1: Check the response times with different workloads.
For this test, the simulator created random requests considering the total computing power of the system. Table 1 shows a comparison between both algorithms in terms of response time delays, given as a percentage value from the value promised in the owner's license. Final results were obtained by averaging multiple simulation results. In the proposed algorithm the average response time goal is satisfied, with important delays appearing just when the workload was too high.

Test 2: Check the fairness between requests with respect to the average response time when the available hardware resources are not enough for satisfying the requests. The simulator creates random requests to simulate a high workload then checks how many of them were solved per interval of average response time. The initial interval of average response times [10,500] was split in 5 intervals as the Table 2 shows. Ideally, the number of requests solved per each interval should be inverse proportional to the average value of the interval.

System workload	Average delays in response time - proposed algorithm	Average delays in response time algorithm in [1]
20 %	1.23 %	15 %
50 %	1.45 %	26.4 %
100 %	1.72 %	54.8 %
200 %	103.14 %	104.2 %

Table 1: Shows average response times with different system workloads.

Intervals of average response times	Average number of requests solved in proposed algorithm	Average number of requests solved in [1]
10-100	4233	1651
101-200	2397	1675
201-300	1683	1649
301-400	779	1693
401-500	541	1680

Table 2: Number of requests solved for different average response time intervals.

The results show that the second goal is satisfied too in the proposed algorithm, while in the other load balancer there is no fairness between the clients.

The proposed algorithm performs much better when comparing the maximum waiting times of requests thanks to the priority formula. Because of the overhead needed to satisfy the goals, the proposed algorithm had a throughput with 2.11% smaller than the reference algorithm. With a proper tuning of the BONUS_STEP, the number of intervals for splitting the local state data and the time to initiate a new Gather message the algorithm can obtain peak performance with minimizing the overhead. These variables should be tuned considering the granularity of the nodes and the available bandwidth. In the simulation, BONUS_STEP was equal to 2, the number of intervals was 5 and the time to initiate Gather messages was 300 milliseconds. As a recommendation, these variables values should actually represent a percentage value of real input data.

5 Conclusion

This paper presented a load balancing algorithm for distributed real-time systems which have a hierarchical ring topology. The algorithm has two proposed goals: satisfy the average response time if the computing power allows this and keep the fairness between clients with respect to the response times specified in their license. The results presented in Section 4 demonstrate that the algorithm satisfies the proposed goals.

Bibliography

[1] Oguz AKAY, Kayhan ERCIYES, *A Dynamic Load Balancing model for a distributed system*, Mathematical & Computational Applications, 8(3):353-350, 2003.

[2] Ciprian Paduraru, *A New Online Load Balancing Algorithm in Distributed Systems*, Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 14th edition, Pages:327-334, 2012.

[3] Andrew S. Tanenbaum, *Modern Operating Systems (3rd Edition)*, Prentice Hall, December 2007.

[4] Kwang Soo Cho, Un Gi Joo, Heyung Sub Lee, Bong Tae Kim, and Won Don Lee, *Efficient Load Balancing Algorithms for a Resilient Packet Ring Using Artificial Bee Colony*, Applications of Evolutionary Computation, LNCS, 6025:61-70, 2010.

[5] G. Ravindran and M. Stumm, *Hierarchical Ring Topologies and the Effect of their Bisection Bandwidth Constraints*, Proc. Intl. Conf.Parallel Processing, I:51-55, 1995.

[6] Perry Fizzano and Clifford Stein, *Scheduling on a Ring with Unit Capacity Links*, Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures, Pages:210-219, 1994.

[7] Johannes E. Gehrke , C. Greg Plaxton and Rajmohan Rajaraman, *Rapid Convergence of a Local Load Balancing Algorithm for Asyncronous Rings*, Distributed Algorithms, LNCS, 1320:81-95, 1997.

[8] Young-Soo Myung, Hu-Gon Kim, Dong-Wan Tcha, *Optimal Load Balancing on Sonet Bidirectional Rings*, Operations Research, 45(1):148-152, 1997.

[9] Dekel Tsur, *Improved scheduling in rings*, Journal of Parallel and Distributed Computing, 67(5):531-535, 2007.

[10] Amir Gourgy, Ted H. Szymanski, *Cooperative Token-Ring Scheduling For Input-Queued Switches*, Journal of Parallel and Distributed Computing, 58(3):351-364, 2009.

[11] Leonidas Georgiadis, Wojciech Szpankowski, Leandros Tassiulas, *A scheduling policy with maximal stability region for ring networks with spatial reuse*, Queueing Systems (Springer), 19(1-2):131-148, 1995.

[12] Joseph (Seffi) Naor, Adi Rosen, Gabriel Scalosub, *Online time-constrained scheduling in linear and ring networks*, Journal of Discrete Algorithms, 8(4):346-355, 2010.