Measuring the Cost of Online Load-Balancing in Distributed Range-Queriable Systems

Ioannis Konstantinou, Dimitrios Tsoumakos and Nectarios Koziris
Computing Systems Laboratory, School of ECE, National Technical University of Athens
Email:{ikons, dtsouma, nkoziris}@cslab.ece.ntua.gr

Abstract

Distributed systems such as Peer-to-Peer overlays have been shown to efficiently support the processing of range queries over large numbers of participating hosts. In such systems, uneven load allocation has to be effectively tackled in order to minimize overloaded peers and optimize their performance. In this work, we detect and analyze the two basic methodologies used to achieve load-balancing: Iterative key re-distribution between neighbors and node migration. Based on this analysis, we propose a hybrid method that adaptively utilizes these two extremes to achieve both fast and cost-effective load-balancing in distributed systems that support range queries. As a case study, we offer an implementation on top of a Skip Graph, where we validate our findings in a variety of workloads. Our experimental analysis shows that the hybrid method converges 10% faster than simple neighbor item exchanges and is more than 70% bandwidth efficient compared to simple node migrations.

1 Introduction

Data skew is a well-documented concern for a variety of applications. For instance, it has been widely observed that most Internet-scale applications, including P2P ones, exhibit highly skewed workloads [1]. Failing or departing nodes further reduce the availability of various content. Consequently, resources become scarce, servers get overloaded and throughput can diminish due to high workloads that can by themselves cause denial of service [2].

Data replication techniques is one commonly utilized solution to remedy these situations. Nevertheless, there are cases in which the requested resources cannot be arbitrarily replicated and retrieved across a distributed set of nodes. Distributed data-structures that support range-queries is such an example: The keys are stored in the network nodes so that a natural order is preserved and range-queries are efficiently handled. The interest in such structures is increasing, as they can be very useful in a variety of situations: On-line games [3], web servers [4], etc. In such cases, adaptive and on-line load-balancing schemes must be employed in order to avoid resource unavailability and improve performance in a variety of workloads.

In current bibliography, there exist a variety of methods that try to achieve efficient load balancing for such structures. Yet, only two different mechanisms are usually employed: *Node Migration* (hence *MIG*) and *Neighbor Item Exchange* (hence *NEIX*). These techniques represent two different approaches to handling the problem: *MIG* utilizes underloaded peers by placing them in overloaded areas of the network (see Figure 2, where node D is placed between nodes A, B sharing part of their load). *NEIX* balances load through iterative item exchanges between neighboring nodes (see Figure 1, where iterative key exchanges between (A,B), (B,C) and (C,D) node pairs produce a balanced load). The majority of proposed approaches utilize a version of these two schemes in order to finally balance load among peers each responsible for a given range of the data [5–7]. While they both achieve their goal, their effectiveness and cost greatly vary, making a method that utilizes only one of them inefficient for all cases.

Our contribution: In this paper, we formally identify these two different methodologies that, iteratively applied, perform load balancing on distributed range-partitioned data structures. An important result of our work is the observation that, through mere key exchanges the achieved result can be highly delayed, whereas using only node migrations the cost of updating the structure is immense. Based on this analysis, we describe *NEIXMIG*, a hybrid method that utilizes both *NEIX* and *MIG* in order to minimize overloaded peers and balance the load distribution among them: Load moves in a "wave-like" fashion from more to less loaded regions of the structure adaptively, using our version of the *NEIX* mechanism. When we locally identify highly overloaded regions, we activate *MIG*. Furthermore, we present a Skip Graph [8] implementation for both the two simple mechanisms as well as the hybrid method. We measure their behavior in a variety of skewed and dynamic workloads. Our results validate our analysis and show that our method can balance at low cost (70% less expen-

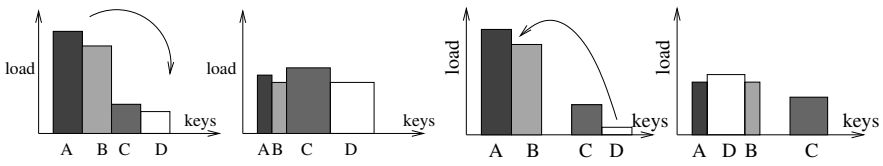

Figure 1. NEIX example.

Figure 2. MIG example.

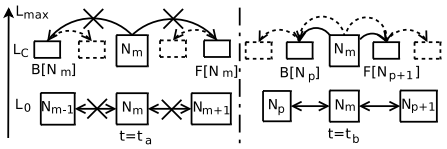

Figure 3. MIG Cost.

sive than *MIG*) and high convergence rate (10% faster than *NEIX*), adapts to changing workloads and is highly customizable.

2 Notation and Problem Setup

We consider the indexing and storing of M keys $(k_1, k_2, \ldots, k_M)$ in N nodes, where N≪M. We assume that a key represents an object or item, hence we shall use these terms interchangeably. Each server N_j stores and serves a range $r_j(t) = \{k_{u_j(t)}, \ldots, k_{v_j(t)}\}$ of the key space, $1 \leq u_j(t) \leq v_j(t) \leq M$. The partition boundaries are consecutive, i.e., $u_{j+1}(t) = v_j(t)$. As *item load* $l_i(t)$ of item k_i at time t, we define the number of user requests for this specific item over a specific time interval. Item load can be viewed as a portion of bandwidth (kb/sec) consumed on queries for this key. The *server load* $S_{N_j}(t)$ of node N_j at time t is the sum of the loads of the items that it stores: $S_{N_j}(t) = \sum_{i=u_j(t)}^{v_j(t)} l_i(t)$, j = 1 . . . N.

We are interested in keeping the natural ordering of the indexed keys, so as to facilitate the routing and answering of range queries. Each stored item has a different time-varying popularity. Users perform both exact match and range queries (where more than one node may be contacted in order for the correct answer to be computed). We assume that each node N_i, according to its capabilities sets a local load threshold, $thres_i$. When the load exceeds this value $S_{N_i}(t) > thres_i$, the node wishes to shed some of its load according to the load balancing algorithm that is implemented. Our goal is to transform the set of partition boundaries through consecutive either item exchanges or node migrations after some time so that $S_{N_i}(t') < thres_i, \forall i$. In addition, our goal is to achieve a balanced load distribution.

3 Load Balancing using Neighbor Item Exchange and Node Migration

Balancing is performed by transferring keys from overloaded peers (which we will refer to as *splitter* peers) to less loaded ones (*helper* peers). In the *NEIX* case, this is performed without any overlay maintenance cost, unlike with *MIG*: Distributed structures that support range queries perform routing in logarithmic time by maintaining a routing table list of $logN$ increasingly distant nodes (for an overlay of size N) placed in L_{max} levels (at the lowest level, L_0, each node holds the IDs of its immediate neighbors, etc). Figure 3 depicts the message exchanges that occur when node N_m leaves its place at time t_a and re-joins next to node N_p at time t_b. Solid lines represent routing links, whereas dotted ones represent the messages required for overlay maintenance. Every node contains $2L_{max}$ routing entries. For simplicity, we describe the procedure for a random level, L_c. Before N_m leaves its place, it removes every forward and backward link stored in tables $F[N_m]$ and $B[N_m]$ respectively (lines marked with an X). This triggers a number of message exchanges where nodes that were in $F[N_m]$ and $B[N_m]$ contact a number of distant nodes in order to fill their routing table "hole" (dotted lines on the left side of Figure 3). This operation is carried out for every old neighbor in $2L_{max}$ levels. When N_m re-joins between N_p and N_{p+1} (right side of Figure 3) at time t_b, it uses $F[N_{p+1}]$ as forward and $B[N_p]$ as backward links in each level and scans the structure to create its own routing table.

3.1 NEIX

In this section we describe the load exchange between neighboring nodes. For simplicity, we describe the situation where nodes are linearly placed (i.e., each node has two neighbors with consecutive ranges) and keys are transferred from the node to its forward neighbor. The transferring node sets a pointer $j = v_i$ and scans its range towards the backward direction. The procedure stops when sufficient number of items have been found so as to fulfill its request. Nevertheless, an obvious disadvantage of *NEIX* is that possibly many consecutive such operations may be needed in order to balance load inside large regions of loaded peers.

3.2 MIG

MIG is performed in three phases: In the probing phase, the overloaded node scans its routing table to find a distant pair of underloaded nodes that are not participating in another balancing operation to migrate next to it in the overlay. When this phase succeeds, the helper peer transfers its partition to its neighboring nodes, and empties its routing table info. In the final step, the helper peer places itself next to the overloaded peer, accepts a portion of its partition and creates a new routing table.

4 NEIXMIG

In this section we describe *NEIXMIG*, our proposed hybrid approach that is a *NEIX-MIG* combination. The goal of *NEIXMIG* is to balance load by adaptively choosing to utilize either *NEIX* or *MIG*. The rationale behind our method is that *MIG* is fast but costly, whereas *NEIX* is slow and cost

effective. Hence, we devise a scheme that, using only local knowledge, identifies conditions where *MIG* is necessary to speed up the the balancing process but is not excessively utilized. In short, when a large demand for *NEIX* operations is detected in a neighborhood, our method employs node migrations for faster load relief in that area.

NEIXMIG is performed in two phases: In the first phase, the overloaded peer tries to reserve a number of its neighbors and examines their current load status. If the locking succeeds, the algorithm moves to the balancing phase: If the node neighborhood is underloaded, the initiator starts an iterative procedure, which we will refer to as a *NEIX* wave, where portions of ranges are transfered from one locked node to its neighbor for all the participating nodes. In the opposite case, it initiates a *MIG* procedure. In detail:

Locking phase: When the current load of an idle node exceeds its self-imposed threshold $thres_i$, the node sends a *LockRequest* message to one of its neighbors. When the *LockRequest* succeeds by contacting an idle recipient, the recipient node forwards the *LockRequest* to another neighboring node, until a number of *hopcount* nodes have been successfully locked. Locked nodes continue to answer user queries but they do not participate in or initiate other balancing actions until they are unlocked by the wave initiator or a timeout has occurred. During each *LockRequest*, the contacted node calculates the *Potential Load* that would end up to it if the *NEIX* wave were performed.

Balancing phase: The last node of the Locking phase decides the balancing action: if its *Potential Load* is bigger than a maximum threshold, it aborts the locking wave and informs the initiator to perform a *MIG* operation, otherwise it allows the initiator to begin a *NEIX* wave.

To sum up, *NEIXMIG* first examines the neighborhood of an overloaded node: if its extra load can be absorbed by its neighbors it performs a cost-effective "wave-like" set of successive item exchanges. If this is not the case because, for instance, the entire neighborhood is overloaded, it initiates a fast but more expensive migration request.

5 Experimental Results

We now present a comprehensive simulation-based evaluation of our method on our own discrete event simulator written in Java. Starting off from a pure Skip Graph implementation, we incorporate our online balancing algorithms on top. By default, we assume a network size of 500 nodes, all of which are randomly chosen to initiate queries at any given time. During the start-up phase, each node stores and indexes an equal portion of the data, that is $\frac{M}{N}$ keys. We assume 50K keys exist in the system, thus each node is initially responsible for 100 keys.

Queries occur at rate $\lambda_r = 1req/sec$ with exponentially distributed inter-arrival times in a 4000 sec total simulation time (balancing is initiated at t=700sec). Each requester peer creates a range by choosing a starting and ending value according to a *zipfian* distribution, where the probability of a key i being asked is analogous to $i^{-\theta}$ or a *pulse* distribution, where a range of keys has a constant load and the rest of the keys are not requested. In the following, we plan to demonstrate the effectiveness of our protocol to minimize overloaded peers and create a load-balanced image of the system. In order to give a metric that summarizes the quality of load balancing achieved, we use the *Gini coefficient*, as it has been proposed in [9].

5.1 *NEIX* and *MIG* Performance

In the first set of experiments, we plan to identify the cost and speed of applying *MIG* and *NEIX* in a number of different workloads. To apply *NEIX*, we use *NEIXMIG* without the *MIG* operation. As our input workload we use a *pulse* with a constant height of 500 reqs/sec and we vary its width from 10% to 40%. In every case, nodes set their $thres$ value to 350reqs/sec. This $thres$ value can also be seen as corresponding to 350kb/sec bandwidth allocation, assuming that, for each request, 1kb of data is transmitted.

In Figure 4 we present the completion time, the number of exchanged messages and items of each algorithm for the applied workloads. In the first graph, we notice that *MIG* is not affected by the width of the pulse and balances load faster than *NEIX*. On the other hand, *NEIX* is three times slower for a pulse of width 40% than for a pulse of width 10%: a larger overloaded area requires more successive key redistributions, thus more time to balance load. This is not required in the case of *MIG*, as nodes that lie in the middle of the overloaded area are able to initiate multiple parallel node migration requests. Nevertheless, the speed of *MIG* comes at a great cost. In the second graph we present the cost of message exchange in the case of *NEIX* and *MIG*: *MIG* constantly requires a larger number of messages compared to *NEIX*. Finally, in the third graph we present the cost of each algorithm in terms of item exchanges. We notice that *MIG* item exchanges increase slowly when the pulse width increases and are considerably less than in the *NEIX* setting. This is not the case for *NEIX*: a pulse of width 40% requires about 6 times more item exchanges than the ones required by a pulse of 10% width. This happens because nodes that exist in the middle of the overloaded structure need to iteratively transfer a larger number of items until they reach the underloaded area.

These experiments confirm that *MIG* is fast and costly in terms of message exchanges, whereas *NEIX* is slow and expensive in terms of item exchanges. Therefore, it is clear that algorithms that rely only on *MIG* or *NEIX* operations cannot balance efficiently an arbitrary input load.

5.2 Measuring the effectiveness of *NEIXMIG*

In this experiment, we compare *NEIXMIG* against *MIG* and *NEIX* in a number of different input workloads.

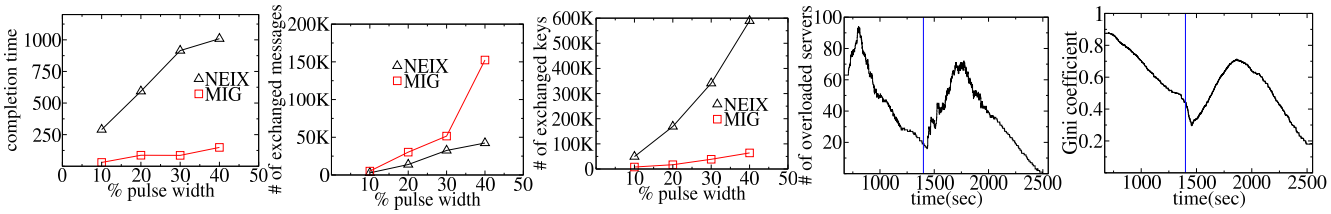

Figure 4. Completion time, exchanged messages and items of *NEIX* and *MIG* for various pulse widths

Figure 5. Overloaded peers and Gini over time for the dyn. setting

Table 1. Percentage gains/losses of *NEIXMIG* vs *NEIX* and *MIG* convergence time and cost

Load type	% Time convergence		% of exchanged messages		% of exchanged items	
	NEIX	*MIG*	*NEIX*	*MIG*	*NEIX*	*MIG*
zipf $\theta = 1$	11.4	-18.7	-26.9	90.7	16.1	-193.7
zipf $\theta = 2$	12.3	-21.3	-30.4	82.8	35.3	-218.8
Pulse 25%	4.2	-280	-10.3	69.5	16.5	-348.5
Pulse 10%	9.0	-401.6	-2.5	59.7	18.3	-375.7

We apply two zipfian loads with $\theta = 1$ and $\theta = 2$ (more biased) and two pulse loads with width 10% and 25%. In Table 1, we present the gains (as positive values) and losses (as negative ones) of the hybrid version in terms of reduction in time, exchanged messages and exchanged items. In the first two rows we present the gain/loss of our approach for the zipfian workloads. We notice that *NEIXMIG* converges faster than *NEIX* and is around 20% slower than *MIG*. We also see that *NEIXMIG* is more expensive than simple *NEIX* in terms of message exchange, as it performs extra probing and routing maintenance messages. Nevertheless, *NEIXMIG* is less expensive than *MIG* around 90%. In the last column, we compare *NEIXMIG* to *NEIX* and *MIG* in terms of item exchange cost, where we observe that selective migrations help *NEIXMIG* to minimize item exchange cost over 15% compared to *NEIX*. *MIG* proves more efficient than *NEIXMIG* in item exchanges cost, but exchanges more messages. In the final two rows we present the results for the pulse workloads where we notice *NEIXMIG*'s similar behavior as in the zipfian setting.

5.3 *NEIXMIG* under dynamic workload

We now present results showing the performance of our *NEIXMIG* method when the workload suddenly changes its skew. We assume an initial pulse load of width 12.5% and height 275. This pulse suddenly moves at time t=1400sec from items [12500, 18700] to [31300, 37600]. Note that this is the worst possible scenario for *NEIXMIG*, since the skew changes completely and abruptly at this time.

Figure 5 shows how the number of overloaded peers and the balancing efficiency changes over time. Naturally, both metrics are affected immediately after the change in load occurs: The Gini coefficient almost doubles in value and so does the number of overloaded peers. Nevertheless, *NEIXMIG* adapts and manages to reduce both quantities. The reason that the convergence time is documented to be larger than that of handling a single pulse is obvious: The very sudden change in skew forces the invalidation of many already performed balance operations and nodes with no load problem suddenly become overloaded.

6 Conclusions

In this paper, we evaluated the performance in terms of bandwidth cost and convergence speed of balancing range queriable data structures using successive item exchanges between neighbors or node migrations. Our experimental results show that none of these methods by itself is capable of efficiently balancing arbitrary workloads: Neighbor item exchanges are expensive in terms of item transfers and slow in terms of convergence speed, whereas node migrations are fast but costly in terms of message exchange. Based on these findings, we proposed and evaluated a hybrid approach that adaptively decides the appropriate balancing action. Load moves in a "wave-like" fashion until it is absorbed by underloaded nodes, and node migration is triggered only when it is necessary. Our simulation results show a gain of 10% and 70% compared to simple *NEIX* and *MIG* in the algorithm convergence time and cost respectively under a variety of skewed and dynamic workloads.

References

[1] S. Sen and J. Wong, "Analyzing peer-to-peer traffic across large networks," in *SIGCOMM IM Workshop*, 2002.

[2] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites," in *WWW*, 2002.

[3] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, "Peer-to-peer support for massively multiplayer games," *Infocom*, 2004.

[4] Q. Luo and J. F. Naughton, "Form-based proxy caching for database-backed web sites," in *VLDB*, 2001, pp. 191–200.

[5] D. R. Karger and M. Ruhl, "Simple efficient load-balancing algorithms for peer-to-peer systems," *Theory of Computing Systems*, vol. 39, pp. 787–804, Nov. 2006.

[6] P. Ganesan, M. Bawa, and H. Garcia-Molina, "Online balancing of range-partitioned data with applications to peer-to-peer systems," in *VLDB*, 2004, pp. 444–455.

[7] S. Surana, B. Godfrey, K. Lakshminarayanan, R. Karp, and I. Stoica, "Load balancing in dynamic structured peer-to-peer systems," *Performance Evaluation*, vol. 63, no. 3, pp. 217–240, 2006.

[8] J. Aspnes and G. Shah, "Skip graphs," *ACM Trans. Algorithms*, vol. 3, p. 37, 2007.

[9] T. Pitoura, N. Ntarmos, and P. Triantafillou, "Replication, load balancing and efficient range query processing in dhts," in *EDBT*, 2006, pp. 131–148.