Passive/Active Load Balancing with Informed Node Placement in DHTs

Mikael Högqvist and Nico Kruber

Zuse Institute Berlin
Takustr. 7, 14195, Berlin, Germany
hoegqvist@zib.de kruber@zib.de

Abstract. Distributed key/value stores are a basic building block for large-scale Internet services. Support for range queries introduces new challenges to load balancing since both the key and workload distribution can be non-uniform.

We build on previous work based on the power of choice to present algorithms suitable for active and passive load balancing that adapt to both the key and workload distribution. The algorithms are evaluated in a simulated environment, focusing on the impact of load balancing on scalability under normal conditions and in an overloaded system.

1 Introduction

Distributed key/value stores [1–3] are used in applications which require high throughput, low latency and have a simple data model. Examples of such applications are caching layers and indirection services. Federated key/value-stores, where the nodes are user contributed, require minimal management overhead for the participants. Furthermore, the system must be able to deal with large numbers of nodes which are often unreliable and have varying network bandwidth and storage capacities. We also aim to support both exact-match and range queries to increase flexibility for applications and match the functionality of local key/value-stores such as Berkeley DB and Tokyo Cabinet.

Ring-based Structured Overlay Networks (SONs) provide algorithms for node membership (join/leave/fail) and to find the node responsible for a key within $O(\log N)$ steps, where N is the number of nodes. One of the main advantages of SONs for large-scale services is that each node only has to maintain state of a small number of other nodes, typically $O(\log N)$. Most SONs also define a static partitioning strategy over the data items where each node is responsible for the range of keys from itself to its predecessor.

At first glance SONs may therefore seem to be a good fit for distributed key/value stores. However, the static assignment of data items to nodes in combination with the dynamic nature of user-donated resources make the design of the data storage layer especially challenging in terms of reliability [4] and load balancing.

The goal of load balancing is to improve the fairness regarding storage as well as network and CPU-time usage between the nodes. Imbalance mainly occurs due to: 1) non-uniform key distribution, 2) skewed access frequency of keys and 3) node heterogeneity. First, by supporting range-queries, an order-preserving hash function is used to map keys to the overlay's identifier space. With a non-uniform key distribution a node can become responsible for an unfair amount of items. Second, keys are typically accessed with different popularity which creates uneven workload on the nodes. The third issue, node capacity differences, also impacts the imbalance. For example, a low capacity node gets overloaded faster than a high capacity node. We assume that nodes are homogeneous or have unit size, where a single physical node can run several overlay nodes.

Our main contribution is a self-adaptive balancing algorithm which is aware of both the key distribution and the item load, i.e. used storage and access-frequency. The algorithm has two modes: *active*, which triggers a node already part of the overlay to balance with other nodes and *passive*, which places a joining node at a position that reduces the overall system imbalance. In both the passive and active mode, a set of nodes are sampled and the algorithm balance using the node with the highest load.

Our target application is a federated URL redirection service. This service allow users to translate a long URL, from for example Google Maps, to a short URL. The redirection service supports look-ups of single URLs as well as statistics gathering and retrieval over time which motivates the need for range queries to execute aggregates. Popular URL redirection providers such as `tinyurl.com` have over 60 million requests per day and close to 300 million indirections.

Section 2 contains the model, assumptions and definitions that are used for the load balancing algorithm presented in Section 3. In Section 4, we evaluate the system using a simulated environment. Results from the simulation show that the algorithm improves the load imbalance within a factor 2-3 in a system with 1000 nodes. In addition, we also show that load balancing reduces the storage capacity overhead necessary in an overloaded system from a factor 10 to 8.

2 System Model

A ring-based DHT consists of N nodes and an identifier space in the range $[0, 1)$. This range wraps around at 1.0 and can be seen as a ring. A node, n_i, at position i has an identifier n_i^{ID} in the ID space. Each node n_i has a *successor*-pointer to the next node in clockwise direction, n_{i+1}, and a *predecessor*-pointer to the first counter-clockwise node, n_{i-1}. The last node, n_{N-1}, has the first node, n_0 as successor. Thus, the nodes and their pointers create a double linked list where the first and last node are linked. We define the distance between two identifiers as $d(x, y) = |y - x| \bmod 1.0$.

Nodes can fail and join the system at any time. When a node joins, it takes over the range from its own ID to the predecessor of its successor. Similarly, when a node n_i fails, its predecessor becomes predecessor of n_i's successor. We model churn by giving each node a mean time to failure (MTTF). To maintain the system size, a failed node is replaced after a recovery time-out.

Storage When a key/value-pair or item is inserted in the system it is assigned an ID using an order-preserving hash-function in the same range as the node IDs, i.e. $[0, 1)$. Each node in the system stores the subset of items that falls within its responsibility range. That is, a node n_i is *responsible* for a key iff it falls within the node's key range $(n_{i-1}^{ID}, n_i^{ID}]$.

Each item is replicated with a replication factor f. The replicas are assigned replica keys according to symmetric replication where the identifier of an item replica is derived from the key and the replica factor using the formula $r(k, i) = k + (i - 1) * \frac{1}{f} \bmod N$, k is the item ID and i is the ith replica [5]. An advantage of symmetric replication is that the replica keys are based on the item key. This makes it possible to look-up any replica by knowing the original key. In other approaches such as successor-list replication [6] the node responsible for the key must first be located in order to find the replicas.

A replica maintenance protocol ensures that a node stores the items and the respective replicas it is responsible for. The protocol consist of two phases; the synchronization phase and the data transfer phase. In the synchronization phase, a node determines which items should be stored at the node using the symmetric replication scheme. And if they are not stored or not up-to-date, which replicas need to be retrieved. The retrieval is performed during the data transfer phase by issuing a read for each item.

Load and Capacity Each node has a workload and a storage capacity. The workload can be defined arbitrarily, but for a key/value-store this is typically the request rate. Each stored item has a workload and a storage cost. A node cannot store more items than its storage capacity allows. The workload, on the other hand, is limited by for example bandwidth, and a node can decide if a request should be ignored or not. We model the probability of a request failure as $P(fail) = 1 - \frac{1}{\mu}$, where μ is the current node utilization, i.e. the measured workload divided by the workload capacity.

Imbalance We define the system imbalance of a load attribute (storage or workload) as the ratio between the highest loaded node and the system average. For example, for the storage, the imbalance is calculated as $\frac{L_{max}}{L_{avg}}$. L_{max} is the maximum number of items stored by a node and L_{avg} is the average number of items per node.

3 Load Balancing algorithm

The only way to change the imbalance in our model is to change the responsibility of the nodes. A node's responsibility changes either when another node joins between itself and its predecessor, or when the predecessor fails. Thus, we can balance the system either *actively* by triggering a node to fail and re-join or *passively* by placing a new node at an overloaded node when joining. Passive balancing uses the system churn, while active induces churn and extra data

```
def placement():
    balanced_ID = $\perp$
    current_distance = $\infty$
    for item in $(n_{i-1}^{ID}, n_i^{ID}]$:
        distance = $f(item^{ID})$ # the placement function
        if distance < current_distance:
            balanced_ID = $item^{ID} + d(item^{ID}, next(item^{ID}))/2$
            current_distance = distance

    return balanced_ID

def sample():
    samples = [(n.load(), n)
               for n in random_nodes(k)]
    return max(samples)

def passive():
    (n_load, n) = sample()
    join(n)

def active():
    (n_load, n) = sample()
    if n_load > local_load * $\epsilon$:
        leave()
        join(n.placement())
```

Fig. 1. Passive and Active load balancing

transfers. We first present the passive/active balancing algorithm followed by the placement function.

The passive/active balancing algorithm presented in Figure 1 uses only local knowledge and can be divided into three parts. 1) sample a set of k random nodes to balance with using e.g. [7], 2) decide the placement of a potential new predecessor and 3) select one of the k-nodes that reduce the imbalance the most. We assume that there is a join function which is used to join the overlay given an ID. `passive` is called before a node is joining and `active` is called periodically. `active` is inspired by Karger's [8] balancing algorithm, but we only consider the case where the node has a factor ϵ less load than the remote node. The ϵ is used to avoid oscillations by creating a relative load range where nodes do not trigger a re-join. `sample` calls a function `random_nodes` that uses a random walk or generates random IDs to find a set of k nodes. The node with the highest load is returned.

Placement Function

The goal of the placement function is to find the ID in a node's responsibility range that splits the range in two equal halves considering both workload and key distribution. When defining the cost for a single load attribute, it is optimal to always divide the attribute in half [9]. We use this principle for each attribute by calculating the ratio between the range to the left of the identifier x and the remaining range up to the node's ID. The optimal position is where this ratio approaches 1. A ratio therefore increases slowly from 0 towards 1 until the optimal value of x is reached, and after 1 the value approaches the total cost for the attribute.

First, let $l_r(a,b) = \sum_{i=0}^{items \in (a,b]} l(item_i)$ be a function returning the load of the items in the range $(a,b]$. $l(item_i)$ is the load of a single item and is defined arbitrarily depending on the load attribute. Second, let n_i be the node at which we want to find the best ID, then the ratio function is defined as follows

$$r(x) = \frac{l_r(n_{i-1}^{ID}, x)}{l_r(x, n_i^{ID})}$$

The workload ratio, $r_w(x)$, could for example be defined using $l(item_i) = weight(item_i) + (rate_{access}(item_i) \times weight(item_i))$. The weight is the total bytes of the item and the access rate is estimated with an exponentially weighted moving mean. For the key distribution ratio, $r_{ks}(x)$, the load function is $l(item_i) = 1$. This means that $r_{ks}(x) = 1$ for the median element in n_i's responsibility range. An interesting aspect of the ratio definitions is that they can be weighted in order to ignore load attributes that changes fast or taking on extreme values.

In order to construct a placement function acknowledging different load attributes, we calculate the product of their respective ratio function. The point x where this product is closest to 1 is where all attributes are being balanced equally. Note that when it equals 1, it means that the load attributes have their optimal point at the same ID.

The placement function we use here considers both the key-space and workload distribution and is more formally described as

$$f(x) = |1 - r_w(x) \times r_{ks}(x)|$$

where x is the ID and n_j is the joining node. The ratio product value is subtracted from 1 and the absolute value of this is used since we are interested in the ratio product value "closest" to 1. Finally, when the smallest value of $f(x)$ is found, a node is placed at the ID between the item, $item_i$ preceding x and the subsequent item, $item_{i+1}$. That is, the resulting ID is $item_i^{ID} + d(item_i^{ID}, item_{i+1}^{ID})/2$.

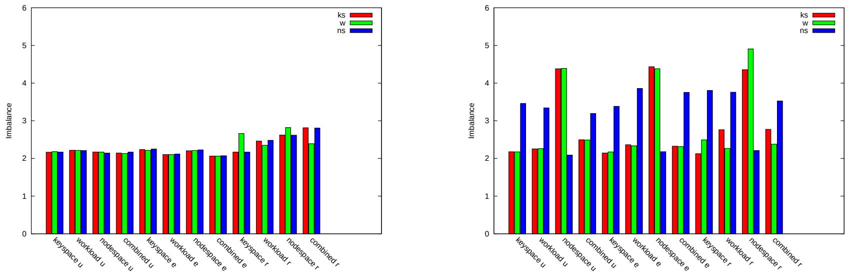
4 Evaluation

This section present simulation results of the passive and active algorithms. The goal of this section is to 1) show the effects of different access-load and key distributions, 2) show the scalability of the balancing strategies when increasing the system size and 3) determine the impact of imbalance in a system close to its capacity limits. Table 1 summarizes the parameters used for the different experiments.

Effect of Workloads In this experiment, we quantify the effect that different access-loads and key distributions have on the system imbalance. The results

	Nodes	Items	Replicas	k	MTTF	Storage	Item Size
Effect of Workloads	256	32768	7	7	∞	∞	1
Network costs	256	8192	7	7	1h	∞	1-1MB
Size of k	256	8192	7	0-20	1h	∞	1
System size	64-1024	2^{15}-2^{18}	3	7	1h	∞	1
Churn	256	8192	7	7	1h-1d	∞	1
Overload	256	8192	7	7	1h	$128 * 7$-$1024 * 7$	1

Table 1. Parameters of the different experiments



(a) Uniform key distribution

(b) Dictionary key distribution

Fig. 2. The effect of different access workloads and key distributions.

from this experiment motivate the use of a multi-attribute placement function. Specifically, we measure the imbalance of the nodespace (ns), keyspace (ks) and the access workload (w).

Four different placement functions are used (x-axis in Fig. 2)

nodespace places a new node in the middle between the node and its predecessor, i.e. $n_i + \frac{d(n_{i-1},n_i)}{2}$.
keyspace places the node according to the median item, $f(x) = |1 - r_{ks}(x)|$.
workload halves the load of the node, i.e $f(x) = |1 - r_w(x)|$
combined uses the placement function defined in section 3.

The simulation is running an active balancing algorithm with $\epsilon = 0.15$.

Workload is generated using three scenarios; uniform (u), exponential (e) and range (r). In the uniform and exponential cases, the items receive a load from either a uniform or exponential distribution at simulation start-up. The range workload is generated by assigning successive ranges of items with random loads taken from an exponential distribution. We expect this type of workload from the URL redirection service when, for example, summarizing data of a URL for the last week.

From the results shown in Figure 2, we can see that the imbalance when using the different placement strategies are dependent on the load type. Figure 2(a) clearly shows that a uniform hash-function is efficient to balance all three metrics under both uniform and exponential workload. In the latter case, this is because the items are assigned the load independently. However, for the range workload, the imbalances are showing much higher variation depending on the placement function. We conclude that in a system supporting range queries, the placement function should consider several balancing attributes for fair resource usage.

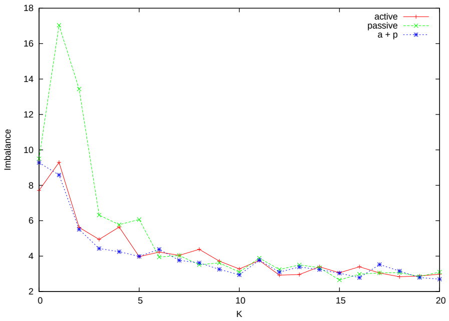
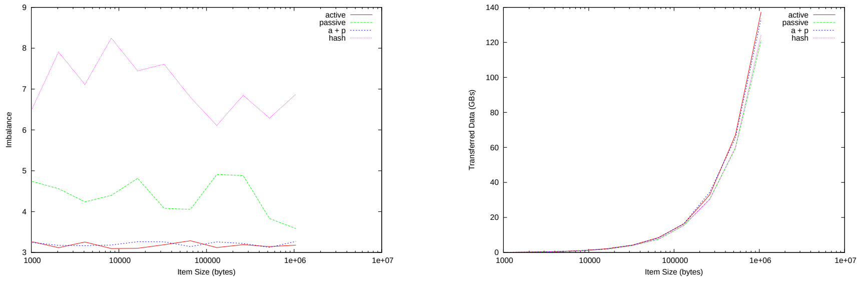


Fig. 3. Imbalance when increasing the number of sampled nodes.

Size of *k* In this experiment, we try to find a reasonable value of the number of nodes to sample, k. A larger k implies more messages used for sampling, but also reduces the imbalance more. The results in figure 3 imply that the value of k is important for smaller values of between 2-10. However, the balance improvement becomes smaller and smaller for each increase of k, similar to the law of diminishing returns. In the remaining experiments we use $k = 7$.

Network costs We define cost as the total amount of data transferred in the system up to a given iteration. This cost is increased by the item size each time an item is transferred. Since there is no application traffic in the simulation environment, the cost is only coming from replica maintenance. That is, item transfers are used to ensure that replicas are stored according to the current node responsibilities. Active load balancing creates traffic when a node decides to leave and re-join the system.

We measure the keyspace imbalance and the transfer cost at the end of the simulation, which is run for 86400s (1 day). Each simulation has 8192 items with 7 replicas and the size of the items is increased from 2^{10} to 2^{20}. The item size has minor impact on the imbalance (Fig. 4(a)). Interestingly, the overhead when using the hash-based balancing strategy as a reference, of active and passive (a+p in the figure) and active only is 5-15% (Fig. 4(b)). The passive strategy



(a) Imbalance vs. Item size

(b) Transferred bytes

Fig. 4. Imbalance and cost of balancing for increasing item size.

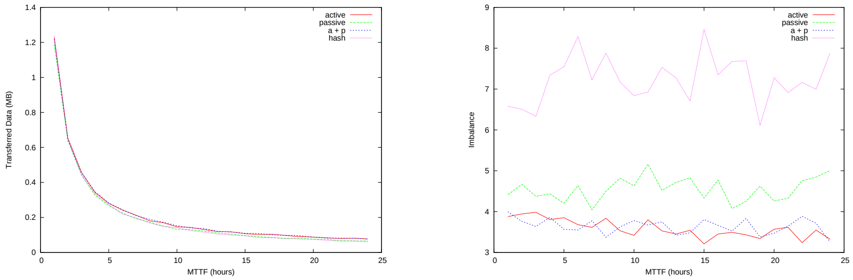
does not show a significant difference. Noteworthy is also that in a system storing around 56 GB of total data (including replicas), over 1 TB aggregated data is transferred. This can be explained with the rather short node lifetime of 3600s.

Churn A node joining and leaving (churn) changes the range of responsibility for a node in the system. Increasing the rate of churn influences the cost of replica maintenance since item repairs are triggered more frequently. In this experiment, we quantify the impact of churn on transferred item cost and the storage imbalance.

In figure 5(a) the node MTTF is varied from 1 to 24 hours. As expected the amount of data transferred is decreasing when the MTTF is increasing. Also as noted in the network costs experiment, the different schemes for load balancing have a minor impact on the total amount of transferred data. Figure 5(b) shows that churn has in principle no impact on the imbalance for the different strategies. This is also the case for the passive approach which only relies on churn to balance the system.

System size The imbalance in a system with hash-based balancing was shown theoretically to be bounded by $O(\log N)$, where N is the number of nodes in the system [10]. However, this assumes that both the nodes and the keys are assigned IDs from a uniform hash-function. In this experiment, we try to determine the efficiency of the placement function with an increasing number of nodes and items.

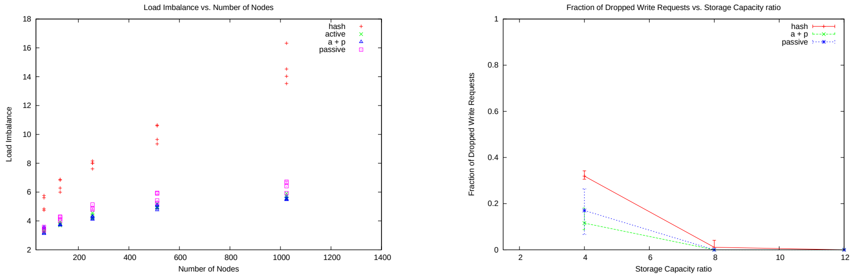
We measure the keyspace imbalance for an increasing number of nodes between 2^5 and 2^{10}. In addition, for each system size we vary the number of items from 2^{15} to 2^{18}. Keys are generated from a dictionary and nodes are balanced using the combined placement function. Four different balancing strategies are compared; 1) IDs generated by a uniform hash-function 2) active without any passive placement, 3) passive without any active and 4) active and passive to-



(a) Bytes transferred with increasing MTTF

(b) Imbalance with varying MTTF

Fig. 5. Imbalance and network cost for varying levels of churn (MTTF).



(a) Increasing nodes and items

(b) Capacity

Fig. 6. Imbalance of the system using different balancing strategies while increasing the system size. The right figure shows the influence of load balancing in an overloaded system.

gether (a+p). For the last three, 7 nodes are sampled when selecting which node to join at or whether to balance at all.

Figure 6(a) shows that the hash-based approach performs significantly worse with an imbalance up to 2-3 times higher compared to the other balancing strategies. Interestingly, the difference in load imbalance when varying the number of items is also growing slightly with larger system sizes. All three variants of the passive/active algorithm show similar performance. The imbalance grows slowly with increasing system size and the difference for different number of items is small. Thus, we draw the conclusion that these strategies are only minimally influenced by system size and number of items. However, note that we need to perform further experiments varying other parameters such as k to validate these results.

Overload In a perfectly balanced system where at most one consecutive node can fail, nodes can use at most up to 50% of their capacity to avoid becoming overloaded when a predecessor fails. This type of overload leads to dropped write requests when there is insufficient storage capacity and dropped read request with insufficent bandwidth and processing capacity. Since a replica cannot be recreated when a write is dropped, this influences the data reliability. The goal of this experiment is to better understand the storage capacity overhead to avoid dropped writes.

We start the experiment such that the sum of the item weights equals the aggregated storage capacity of all nodes. Then by increasing the node's storage capacity we decrease their fill-ratio and thereby the probability of a dropped write. The system is under churn and lost replicas are re-created using a replica maintenance algorithm executed periodically at each node. The y-axis in Figure 6(b) shows the fraction of dropped write requests and the x-axis shows the storage capacity ratio. We do not add any data to the system which means that a write request is dropped when a replica cannot be created at the responsible node because of insufficient storage capacity. We measured the difference with hash-based balancing vs. the active and active + passive with 7 sampled nodes and the combined placement function.

Figure 6(b) shows that a system must have at least 10x the storage capacity over the total storage load to avoid dropped write requests when using hash-based balancing. Active and active-passive delays the effect of overload and a system with at least 8x storage capacity exhibits a low fraction of dropped requests.

5 Related Work

Karger et al. [8] and Ganesan et al. [11] both present active algorithms aiming at reducing the imbalance of item load. Karger uses a randomized sampling-based algorithm which balances when the relative load value between two nodes differs by more than a factor ϵ. Ganesan's algorithm triggers a balancing operation when a node's utilization exceeds (falls below) a certain threshold. In that case, balancing is either done with one of its neighbors or the least (most) loaded node found. Aspnes at al. [12] describe an active algorithm that categorizes nodes as closed or open depending on a threshold and groups them in a way so that each closed node has at least one open neighbor. They balance load when an item is to be inserted into a closed node that cannot shed some of its load to an open neighbor without making it closed as well. A rather different approach has been proposed by Charpentier et al. [13] who use mobile agents to gather an estimate of the system's average load and to balance load among the nodes. Those algorithms however do not explicitly define a placement function or use a simple "split loads in half" approach which does not take several load attributes into account.

Byers et. al. [14] proposed to store an item at the k least loaded nodes out of d possible. Similarly, Pitoura et al. [15] replicate an item to k of d possible identifiers when a node storing an item becomes overloaded (in terms of requests). This technique, called the "power of two choices" was picked up by Ledlie et. al [16] who apply it to node IDs and use it to address workload skew, churn and heterogeneous nodes. With their algorithm, k-Choices, they introduce the concept of passive and active balancing. However, their focus is on virtual server-based systems without range-queries. Giakkoupis and Hadzilacos [17] employ this technique to create a passive load balancing algorithm including a weighted version for heterogeneous nodes. There, joining nodes contact a logarithmic (in system size) number of nodes and choose the best position to join at. Their focus on the other hand is on balancing the address-space partition rather than arbitrary loads. Manku [18] proposes a similar algorithm issuing one random probe and contacting a logarithmic number of its neighbors. An analysis of such algorithms using r random probes each followed by a local probe of size v is given by Kenthapadi and Manku [19]. However, only the nodespace partitioning is examined.

In Mercury [20] each node maintains an approximation of a function describing the load distribution through sampling. This works well for simple distributions, but as was shown in [21] it does not work for more complex cases such as file-names. Instead, [21] introduces OSCAR where the long-range pointers are placed by recursively halving the traversed peer population in each step. Both OSCAR and Mercury balance the in/out-degree of nodes. While this implies that the routing load in the overlay is balanced, it does not account for the placement of nodes according to item characteristics.

6 Conclusions

With the goal of investigating load balancing algorithms for distributed key/value-stores, we presented an active and a passive algorithm. The active algorithm is triggered periodically, while the passive algorithm uses joining nodes to improve system imbalance. We complement these algorithms with a placement function that splits a node's responsibility range according to the current key and workload distribution. Initial simulation results are promising showing that the system works well under churn and scales with increasing system sizes. Ongoing work include quantifying the cost of the algorithms within a prototype implementation of a key/value-store.

Acknowledgments This work is partially funded by the European Commission through the SELFMAN project with contract number 034084.

References

1. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP, ACM (2007) 205–220

2. Rhea, S.C., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., Yu, H.: Opendht: a public dht service and its uses. In: SIGCOMM, ACM (2005) 73–84
3. Reinefeld, A., Schintke, F., Schütt, T., Haridi, S.: Transactional data store for future internet services. Towards the Future Internet - A European Research Perspective (2009)
4. Blake, C., Rodrigues, R.: High availability, scalable storage, dynamic peer networks: Pick two. In Jones, M.B., ed.: HotOS, USENIX (2003) 1–6
5. Ghodsi, A., Alima, L.O., Haridi, S.: Symmetric replication for structured peer-to-peer systems. In: DBISP2P. (2005) 74–85
6. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM. (2001) 149–160
7. Vishnumurthy, V., Francis, P.: A comparison of structured and unstructured p2p approaches to heterogeneous random peer selection. In: USENIX. (2007) 309–322
8. Karger, D.R., Ruhl, M.: Simple efficient load balancing algorithms for peer-to-peer systems. In: IPTPS. Volume 3279 of LNCS., Springer (2004) 131–140
9. Wang, X., Loguinov, D.: Load-balancing performance of consistent hashing: asymptotic analysis of random node join. IEEE/ACM Trans. Netw. **15**(4) (2007) 892–905
10. Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: ACM Symposium on Theory of Computing. (May 1997) 654–663
11. Ganesan, P., Bawa, M., Garcia-Molina, H.: Online balancing of range-partitioned data with applications to peer-to-peer systems. In: VLDB, Morgan Kaufmann (2004) 444–455
12. Aspnes, J., Kirsch, J., Krishnamurthy, A.: Load balancing and locality in range-queriable data structures. In: PODC. (2004) 115–124
13. Charpentier, M., Padiou, G., Quéinnec, P.: Cooperative mobile agents to gather global information. In: NCA, IEEE Computer Society (2005) 271–274
14. Byers, J.W., Considine, J., Mitzenmacher, M.: Simple load balancing for distributed hash tables. In: IPTPS. Volume 2735 of LNCS., Springer (2003) 80–87
15. Pitoura, T., Ntarmos, N., Triantafillou, P.: Replication, load balancing and efficient range query processing in dhts. In: EDBT. Volume 3896 of LNCS., Springer (2006) 131–148
16. Ledlie, J., Seltzer, M.I.: Distributed, secure load balancing with skew, heterogeneity and churn. In: INFOCOM, IEEE (2005) 1419–1430
17. Giakkoupis, G., Hadzilacos, V.: A scheme for load balancing in heterogenous distributed hash tables. In: PODC, ACM (2005) 302–311
18. Manku, G.S.: Balanced binary trees for id management and load balance in distributed hash tables. In: PODC. (2004) 197–205
19. Kenthapadi, K., Manku, G.S.: Decentralized algorithms using both local and random probes for p2p load balancing. In: SPAA, ACM (2005) 135–144
20. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: supporting scalable multi-attribute range queries. In: SIGCOMM, ACM (2004) 353–366
21. Girdzijauskas, S., Datta, A., Aberer, K.: Oscar: Small-world overlay for realistic key distributions. In: DBISP2P. (2006) 247–258