# DHT Load Balancing with Estimated Global Information

## Diplomarbeit

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

Submitted by: Nico Kruber

Supervisor: Prof. Dr. Alexander Reinefeld
Second reader: Prof. Dr. Miroslaw Malek

Berlin, 25th September 2009

# Declaration of own work

I, Nico Kruber, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g. ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed:

Place, Date:

**Abstract**

One of the biggest impacts on the performance of a Distributed Hash Table (DHT), once established, is its ability to balance *load* among its nodes. DHTs supporting range queries for example suffer from a potentially huge skew in the distribution of their items since techniques such as *consistent hashing* [29] can not be applied. Thus explicit load balancing schemes need to be deployed. Several such schemes have been developed and are part of recent research, most of them using only information locally available in order to scale to arbitrary systems.

Gossiping techniques however allow the retrieval of fairly good estimates of global information with low overhead. Such information can then be added to existing load balancing algorithms that can use the additional knowledge to improve their performance. Within this thesis several schemes are developed that use global information like the average load and the standard deviation of the load among the nodes to primarily reduce the number of items an algorithm moves to achieve a certain balance. Two novel load balancing algorithms have then been equipped with implementations of those schemes and have been simulated on several scenarios. Most of these variants show better balance results and move far less items than the algorithms they are based on.

The best of the developed algorithms achieves a $15 - 30\%$ better balance and moves only about $50 - 70\%$ of the number of items its underlying algorithm moves. This variation is also very robust to erroneous estimates and scales linearly with the system size and system load. Further experiments with self-tuning algorithms that set an algorithm's parameter according to the system's state show that even more improvements can be gained if additionally applied. Such a variant based on the algorithm described by Karger and Ruhl [30] shows the same balance improvements of $15 - 30\%$ as the variant above but reduces the number of item movements further to $40 - 65\%$.

# Contents

# 1. Introduction

## 1.1. Context

Distributed Hash Tables (DHTs) store key/value-pairs on several nodes of a network and provide means for inserting, retrieving and deleting a value associated with a key. Each node is assigned a unique node ID in a given ID space uniformly at random and is then responsible for all values with keys near its ID (keys are also mapped to this ID space). By using a technique called *consistent hashing* [29], the DHT then spreads the stored items uniformly over the node ID space which achieves a fair balance without any further effort. More precisely, nodes will have loads varying by $O(\log n)$ times the average load in terms of stored items in a system of $n$ nodes [36, 25]. However, DHTs with range queries like Scalaris [39] cannot use hash functions to spread their items because they need to stay in the order given by their keys. Therefore more effort is needed to balance items among the nodes in such storages.

Consider the following example: Articles are to be stored in a range-query-based DHT with 100 nodes and the key under which an article is stored is its heading. In case of keys in (American) English and nodes responsible for equidistant key ranges, items would then be distributed as shown in Figure 1.1.

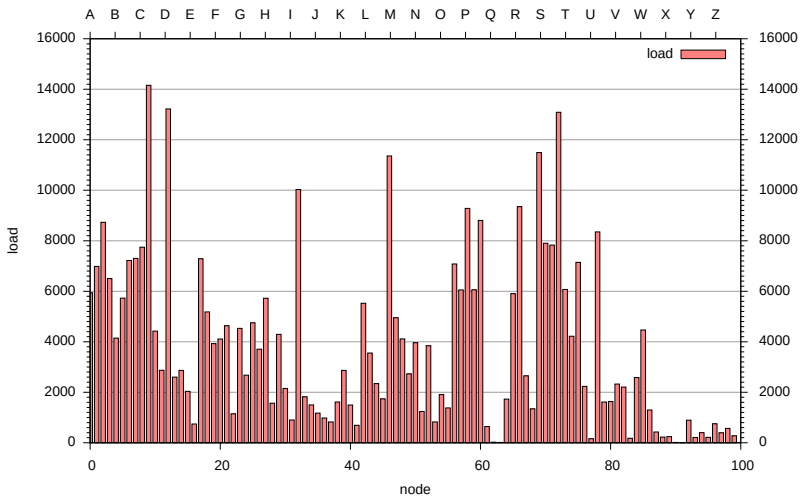



Figure 1.1.: *Item distribution of US-English words on 100 nodes with equidistant key ranges. (list of words aggregated from [11])*

To even out such skewed load distributions load balancing algorithms are required which change the nodes' responsibilities that in turn reduces their load. Such algorithms try to balance an arbitrarily defined *load* at each node and should only use information locally available in order to scale to large systems. Several of such algorithms will be introduced in the following chapters, some of them using different definitions of *load*, e.g. the number of keys a node is responsible for, the number of items a node actually stores or the access-popularity of a node's items. Some also weight load depending on a node's capacities and therefore adapt to heterogeneous environments.

## 1.2. Aims & Objectives

This thesis aims at improving such load balancing algorithms in terms of moved items and reached balance by adding estimates of global information. These values can be retrieved with high confidence and low overhead using gossiping techniques [23, 28] and include approximations of values like the minimum, maximum and average load as well as the standard deviation and system size.

A first approach will try to use the average load of all nodes in its decision on whether to balance two nodes and how much to transfer from one node to another. Most algorithms simply balance two nodes that have been matched by trying to equalise their loads. This does however not involve a node's ideal load - the average load. Therefore several items are transferred multiple times during the algorithm's task to balance the load at each node of the system, especially if a node's load after a balance operation is sufficiently higher than the average load. By knowing the target load and integrating it appropriately, a much better performance can be expected. Preliminary results of an algorithm using the average load already show some improvements compared with its underlying algorithm without that change [27].

Further variations will be introduced into ordinary load balancing algorithms also including some of the other information mentioned above. The resulting algorithms' performance will then be evaluated on a set of given scenarios like the alphabetical distribution of Figure 1.1. It is assumed that with the right use of such information, any algorithm can be significantly improved.

## 1.3. Methods

In order to evaluate any of the introduced algorithms, a simulation will be implemented that emulates a simple DHT with range queries and starts with an initial system load distributed among the nodes according to a given scenario. This emulation will disregard node joins and deletions as well as any other side-effects, e.g. network maintenance,

node failures, network delay and bandwidth etc., to eliminate any other influences when assessing an algorithm. It will however allow analysing the algorithm under different aspects such as different scenarios with different numbers of items and nodes, different choices for an algorithm's parameters and a different accurateness of the estimated global information. It will also provide the ability to run multiple simulations with the same set of parameters in order to allow the evaluation of randomised algorithms that show (slightly) different behaviour in each simulation.

The program will follow the strategy of being easily extensible and will in particular allow additional algorithms and scenarios to be deployed separately and added dynamically via a plugin-based infrastructure. It will also provide means of comparing different algorithms with varying parameters on multiple scenarios. A graphical user interface and a command line client will be created that allow fast evaluations of the algorithms as well as batch-jobs for more time-consuming simulations.

## 1.4. Achievements

At first a survey of the field of Distributed Hash Tables has been given by presenting their concepts and examining their mode of operation including DHTs that support range queries. Additionally a very thorough overview of load balancing schemes that can be applied to (arbitrary) DHTs has been given and several novel load balancing algorithms have been presented. Gossiping algorithms have also been introduced to present a way estimates of global information can be retrieved in Distributed Hash Tables.

Secondly several algorithm variations have been introduced that make use of estimated global information in order to minimise the item movements an algorithm performs as well as the imbalance it reaches. Those variations have then been applied to the load balancing schemes by Karger and Ruhl [30] and Bharambe et al. [12] and have been evaluated by performing simulations with different load distribution scenarios. These variations use estimated values of the system's average and maximum load, the standard deviation of the load among the nodes and the system size.

The best algorithm among those variants limits the original algorithm's item movements in a way that nodes that have a load smaller than the (estimated) average load will not reach a load above this bound. Additionally it only performs such balance operations that increase the standard deviation by at least a factor $s/n$ with $n$ being the system size and $s$ a configurable parameter that has been set to 2.0 and 3.0 for the algorithms by Karger and Ruhl and Bharambe et al. respectively. This variation achieves an up to 30% lower imbalance than the algorithm applied to would achieve alone and only moves about $50 - 70\%$ of its items.

Further experiments that try to tune the algorithms' parameters according to the

system's state show that even more improvements are possible. Applying such *self-tuning* to the algorithm by Karger and Ruhl for example has shown an up to 30% lower imbalance with only about $40 - 65\%$ of the item movements of the original algorithm. Such good performance has however not been achieved by a similar variant that has been applied to the algorithm by Bharambe et al. Further investigations into the field of *self-tuning* algorithms are thus needed.

## 1.5. Outline

At first a deeper insight into the techniques behind Distributed Hash Tables (DHTs) will by given in Chapter 2. It will also present several representatives of DHTs and their characteristics and will introduce methods for achieving range-queriable systems. It will then present gossip algorithms followed by several novel load balancing schemes that are available for such DHTs. Chapter 3 will define the system model that is used in this thesis and the algorithms that have been chosen to be equipped with estimated global information. It will conclude with the introduction of the algorithm variants that have been developed. These variants will be evaluated in the following Chapter 4 starting with a detailed description of the evaluation process itself and the scenarios used. It then presents the results that the implemented simulator created for the different algorithms under different aspects of the simulation. Chapter 5 will finally sum up the achievements of the thesis and will provide ideas about possible extensions of the given algorithms and future work.

# 2. Background / Related Work

This chapter presents Distributed Hash Tables (DHTs), a prominent representative of the class of structured overlay networks, which has been of great interest in research over the past years. The structure of a generic DHT and some of its representatives will be introduced including DHTs that support range queries. This class of DHTs imposes some restrictions on the organisation of the stored resources which need to be considered when designing load balancing algorithms, e.g. stored resources cannot move arbitrarily to different nodes of the network. Before load balancing algorithms are described in the last section of this chapter, gossiping algorithms will be introduced. Those algorithms can be used in peer-to-peer networks to gather estimates of certain global information that is usually not available in such a setting. This information will later be used to improve some of the load balancing algorithms described here.

## 2.1. Distributed Hash Tables (DHTs)

Distributed Hash Tables provide functionality similar to ordinary hash tables. They store *key/value*-pairs on several nodes of a network and provide look-up facilities for retrieving the value associated with a given key. Several such systems exist, but despite their diversity a reference model can be given which models their approaches in a generic manner [8] and is outlined below.

In this model, a DHT maps peers $P$ and resources $R$ to a common identifier space $I$ using mapping functions $f_P : P \to I$ and $f_R : R \to I$. Furthermore, a closeness metric $d : I \times I \to \mathbb{R}$ is defined on $I$ which can be used by a mapping function $M : I \to 2^P$ that associates identifiers with the peers storing them. The peers themselves are organised in a logical network to allow access to every other peer's resources, i.e. by embedding a graph into the identifier space. Following this notation, differences between several DHTs only exist because of the different choices made for the following aspects:

- *Selection of an identifier space with a closeness metric d:* This serves as an address-space for resources and peers and should be large enough to support large systems.
- *Mappings $f_P$ and $f_R$:* These functions may satisfy certain distributional properties which can be exploited for load balancing. They can preserve resource semantics

such as closeness/neighbourhood relations or the order under a given key or completely drop them, e.g. when following a uniform distribution in $I$.

- *Management of the identifier space:* The function $M : I \to 2^P$ assigns each identifier of a resource a set of peers responsible for it. Locating resource $r$ therefore involves finding a peer in $M(f_R(r))$. Note that systems with replication have several peers responsible for each resource.

- *Structure of the logical network:* The logical network can be modelled as a (time-dependent) directed graph $G = (P, E)$ with vertices $P$ (peers) and edges $E$ (direct connections). Also let $N(p)$ be the set of peers a given peer $p$ maintains a connection to, e.g. its neighbours. The overall structure of that graph is then determined by $N(p)$ for every $p \in P$.

- *Routing strategy:* Requests for identifiers need to be routed to their responsible peers. A strategy for that can be described as selecting at a given peer $p$ for an identifier $i$ a set of next peers $R(p, i) \in N(p)$ to which to forward a request. Routing is typically greedy, i.e. $\forall q \in R(p, i) : d(i, f_P(q)) \leq d(i, f_P(p))$, and built on top of the decisions made for the identifier space and its management, e.g. the distance function.

- *Maintenance strategy:* Changes in peer connectivity (referred to as *churn*) may occur quite frequently and create the need for mechanisms to repair the state of the logical network. Since node joins are typically active operations, this task mainly focuses on repairing connections due to node (connection) failures. Maintenance strategy can either follow a proactive approach (heartbeats, periodic probing) or a reactive approach (correction on use, failure or change) or a combination of the two. Functionality of the DHT heavily relies on a consistent network structure making this strategy essential for its operation.

Additionally DHTs provide (supposedly different) implementations for a common set of functionality they expose to their clients.This includes joining and leaving a network, several routing functions, looking up identifiers and getting some administrative information about the local peer and its neighbours. Data management functionality exposes insert, delete and update methods as well as searching for resources using queries of some kind.

Implementations of such structured overlay networks include CAN [37], Pastry [38], Chord [41], Freenet [19], Tapestry [45], Gnutella [5, 1] and more. The following sections will concentrate on the first three which all implement a variant of consistent hashing [29, 33] outlined below. The main focus however is not on a complete description of the different DHTs but to give an overview of their structure and message routing / resource

retrieval algorithms which both is important for load balancing. Also although the descriptions will make use of the introduced terminology and definitions they will not be structured explicitly that way in order to focus on the main aspects of design decisions and establish a better understanding of the techniques. It will however become apparent that the DHTs follow the given model.

### 2.1.1. Consistent Hashing

While traditional hash tables map objects to a static set of buckets, the number of peers to which resources are mapped constantly changes in DHTs. Karger et al. [29] and Lewin [33] describe a *consistent hash function* that operates on a changing set of buckets and provides some *consistency properties*, e.g. adding a bucket only changes the mappings of a minimum fraction of objects needed to maintain a balanced state.

Using the aforementioned syntax, let $P$ be a set of $n = \|P\|$ peers, $I$ the circular interval $[0, 1) \subset \mathbb{R}$ and $f_R$ a random function that maps resources of $R$ $\log(n)$-way independently[1] and uniformly to $I$. Now let each real peer $p$ run $m$ "virtual" peers that operate independently from each other. Virtual peers can be modelled by each peer being mapped to $m$ different identifiers instead of just one: $f_P : P \to I_m \subset I, \|I_m\| = m$ (otherwise the same constraints as $f_R$ apply). Also define a function $M$ that maps each resource $r \in R$ to the peer $p \in P$ that has the closest identifier to $f_R(r)$. Each such hash function has the following properties which also hold for large enough arbitrary $I$:

- *Monotonicity:* If new peers are added to $P$, resources only move from old peers to new peers, but never between old peers.
- Adding a peer $p$ to $P$ changes the mappings of $O(\|R\|/n)$ resources.
- *Balance:* The probability of a resource $r \in R$ being assigned to peer $p \in P$ is $O\left(\frac{1}{n} \cdot \left(1 + \frac{\log(n)}{m}\right)\right)$.

Thus using $m = \Omega(\log(n))$ virtual servers results in a well-balanced state with each node being responsible for $O\left(\|R\|/n\right)$ resources. Having no virtual nodes however ($m = 1$) yields to some nodes having $O(\log(n))$ times more resources associated with them than others because each node is responsible for $O\left((\log(n) + 1) \cdot \|R\|/n\right)$ resources.

### 2.1.2. CAN

A basic CAN network [37] uses a virtual d-dimensional Cartesian coordinate space $C$ for its identifiers and places it on a d-torus for routing. Peers are responsible for their

[1] A random mapping function is $k$-way independent if any $k$ elements are mapped independently. This allows representing real identifiers in $I \subset \mathbb{R}$ with limited precision instead of using an infinite number of bits and also allows for arbitrary large enough discrete $I$.

individual and distinct zones of this coordinate space which is entirely covered at any point in time. A *key*/*value*-pair is stored in CAN by mapping its *key* to a point $q \in C$ using a uniform hash function and storing it at the peer $p$ responsible for the zone containing $q$. Similarly querying for a *key* corresponds to routing to the node responsible for the zone containing $q$. For this every peer maintains a list of immediate neighbours (nodes with zones adjoining their own zone) and routing a message at peer $p$ directed to $q$ is done by forwarding it to the neighbour of $p$ which is responsible for coordinates closest to $q$ (greedy forwarding). Also note that several paths exist and can be used in case of node and connection failures or to deploy a simple request load balancing (see Figure 2.1 for an example). This way using $d$ dimensions and $n = \|P\|$ commensurate zones, each individual node maintains $2d$ neighbours and average routing paths cross $(d/4)(n^{1/d})$ zones (peers).

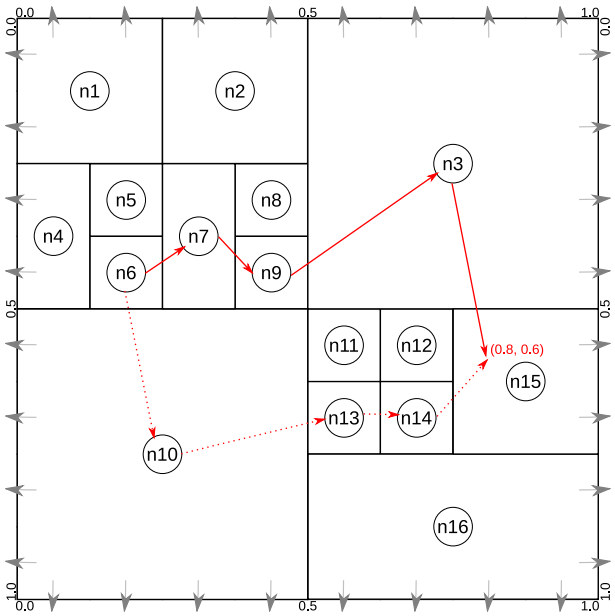



Figure 2.1.: *Planar 2-d CAN with coordinates in range* $[0,1) \times [0,1) \subset \mathbb{R}^2$ *with 16 nodes routing a message from node* $n6$ *to* $q = (0.8, 0.6)$ *(dashed arrows present an alternative route).*

If greedy forwarding fails, an expanding ring search using stateless, controlled flooding may be used to locate peers closer to the destination. From there greedy forwarding will be continued.

In order for a node to join an existing CAN, it needs to take the following steps:

1. Find a CAN node by using some external mechanism, e.g. DNS.
2. (Randomly) choose a point $q$ to join at, contact the peer currently responsible for that point (using normal routing) and split the zone in half assigning one half to the new node.

3. Learn neighbours from the previous occupant and notify them of its arrival.

4. Transfer resources according to the new responsibilities.

These steps involve $O(d)$ existing nodes which need to change their list of neighbours.

A node gracefully leaving the system hands over its zone to a neighbour which is able to join the two zones. If this is not possible the neighbour with the smallest zone will (temporarily) handle both zones. To identify node failures peers normally send periodic update messages to their neighbours. If such message is not received for some time its neighbours each initiate a takeover mechanism and agree on one of them taking over the failed zone, possibly becoming responsible for two zones. To prevent further fragmentation of the coordinate space background zone-reassignment algorithms try to merge zones again.

Further improvements were suggested in order to reduce routing path lengths, i.e. increasing the number of dimensions or using multiple coordinate spaces (*realities*), introducing round-trip-times into routing decisions, caching frequently requested resources or replicating them to their peers' neighbours. See [37] for more details and an evaluation of the various improvements.

### 2.1.3. Pastry

A different approach to realising a distributed hash table is provided by Pastry [38] which uses identifiers represented by 128-bit numbers. The circular identifier space therefore consists of integers in the range $[0, 2^{128} - 1]$ and the two mapping functions $f_P$ and $f_R$ are expected to distribute their results uniformly and independently among the identifier space. Resource identifiers are for example created by applying a secure hash function, e.g. SHA-1 [7], to the resource's name, content and the resource owner's identifier. Also resource $r \in R$ is stored at the $k$ peers with identifiers numerically closest to $f_R(r)$. $k$ can be set individually for each resource at its insertion, influences its availability in case of failures and provides some means of balancing resource requests.

For routing purposes node identifiers in pastry are sub-divided into separate *levels* of $b$ bits with a *domain* at level $l$ being defined as the bits from position $(b \cdot l)$ to $(b \cdot (l+1) - 1)$. Messages are now forwarded using prefix routing, i.e. messages at peer $p$ with destination $f_R(r)$ matching $f_P(p)$ up to level $l$ will be forwarded to a node whose identifier matches the destination's identifier up to at least level $(l+1)$. Also each pastry node $p$ stores information about other peers in 3 different node sets:

- a *routing table* $T$ which contains information about representatives of different domains at different levels: for each level $l$ it contains IP addresses of $(2^b - 1)$ peers with the same prefix as $f_P(p)$ up to level $(l-1)$ but a different domain at level $l$ (to improve route locality, a representative geographically close to $p$ can be chosen),

- a *namespace set* $L$ that contains identifiers and IP addresses of $\|L\|$ peers that are numerically close and centred around $f_P(p)$ which is used for routing too and
- a *neighbourhood set* $M$ storing identifiers and IP addresses of $\|M\|$ peers that are geographically close to $p$ and which is useful for network maintenance. Note that this set has been dropped in later versions of Pastry [15]. The following descriptions however are based on the original version.

While the choice of $b$ influences the size of the routing table (ca. $\lceil \log_{2^b}(n) \rceil \cdot (2^b - 1)$ entries, $n = \|P\|$) and the average routing path length, the sizes of $\|L\|$ and $\|M\|$ can be chosen arbitrarily and are typically $2^b$ and $2^{b+1}$ respectively. Using those tables, a message to resource r arriving at peer $p$ is routed as follows:

*if $f_R(r)$ is in the range of the two farthest nodes in p's namespace set $N(p)$:*
Forward message to $p_i \in N(p)$ so that $|f_R(r) - f_P(p_i)|$ is minimal (possibly $p$).
*else if p's routing table contains a node that shares a longer prefix than p:*
Forward the message to that node.
*else:*
Forward to a known node (from the routing table, namespace set or neighbourhood set) that shares a prefix at least as long as $p$ but is numerically closer to $r$.

Although the third case creates a worst-case with linear performance (in the number of nodes), Rowstron and Druschel [38] argue that this is very unlikely due to the uniform distribution of node identifiers and give an average routing path length of $\lceil \log_{2^b}(n) \rceil$ hops.

Nodes joining a Pastry network need to perform the following 6 steps which involve $O(\log_{2^b}(n))$ remote procedure calls:

1. Find a Pastry node $p_1$ by using some external mechanism.
2. Choose a node identifier (at random), contact the peer $p_2$ currently responsible for resources with that identifier using normal routing.
3. Update its routing tables using the neighbourhood set of $p_1$ and the namespace set of $p_2$ as approximations of its own neighbourhood and namespace sets. Fill the routing table with information from the nodes the *join* message came along.
4. Improve those approximations by requesting the state of the nodes in its routing table and neighbourhood set.
5. Notify peers that need to be aware of the new node and send them its own state.
6. Transfer resources according to the new responsibilities.

Node failures are detected when a node tries to contact another node in its routing table or namespace set. The latter is repaired by using an appropriate node of the namespace set of the live node with the largest identifier in the direction of the failed node. Repairing a representative in the routing table involves contacting another representative at the same level and asking it for the required connection or continuing with requests to nodes at higher levels. The neighbourhood set can be repaired by requesting the neighbourhood sets of the other live nodes in it.

### 2.1.4. Chord

Chord [41] places identifiers of $m$ bits on a circle modulo $2^m$ and performs every calculation modulo $2^m$. A secure hash function, typically SHA-1 [7], is used for mapping resources and peers to this identifier space ($m = 160$ in case of SHA-1). $f_P$ maps peers to identifiers by hashing their IP address and $f_R$ hashes the key of a resource. Using consistent hashing, $M$ maps a resource $r \in R$ to the peer $p \in P$ whose identifier is equal to or follows $r$'s identifier in the identifier space. That is if $predecessor(p)$ denotes the predecessor of a peer $p$ on the identifier circle, $p$ is responsible for all resources with identifiers within $(f_P(predecessor(p)), f_P(p)]$. Note that Chord does not use an explicit load balancing scheme but instead relies on consistent hashing with the use of virtual servers.

Chord nodes store routing information about $m$ nodes in a so-called *finger table*. The $i$'th *finger* of peer $p$'s table, $1 \leq i \leq m$, points to the node $p'$ whose identifier succeeds $f_P(p)$ by at least $2^{i-1}$, i.e. $p.finger[i] = M(f_P(p) + 2^{i-1})$, $p.finger[1] = successor(p)$. Note that consecutive fingers can point to the same node if there is no peer between their designated identifiers. Additionally to the finger table each node maintains a pointer to its predecessor to simplify node join and leave operations. Figure 2.2 shows such a node's complete routing state (including its finger table) in an exemplified Chord ring.

Routing uses those fingers as shortcuts to reach the destination with fewer hops than using successor links alone (which would suffice for routing correctness and result in $O(n), n = \|P\|$ hops)). If peer $p$ needs to find the node $p'$ which is responsible for key $k$, it searches its finger table for the node $j$ whose identifier immediately precedes $k$ and asks $j$ for the node it thinks is closest to $k$. This procedure is repeated until the immediate predecessor of $k$ is found, whose successor is then the node responsible for $k$. Note that those messages could also be forwarded to the nodes recursively instead of implementing an iterative approach as described here. Because the fingers provide shortcuts half-way, quarter-way,... around the circle and the distance to the destination halves in each step this results in $O(\log(n))$ nodes to contact (with high probability[2]).

[2] *with high probability* means probabilities of at least $(1 - O(n^{-1}))$, $n$ being the system size

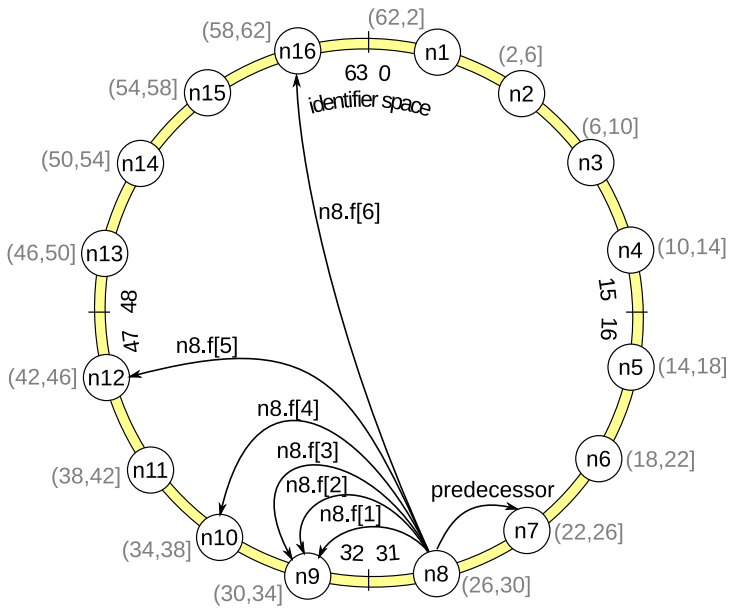



Figure 2.2.: *Chord ring with $m = 6$ and 16 nodes showing the routing pointers of node $n8$ and all nodes' responsibilities in grey (associations with identifiers).*

Nodes joining an existing Chord ring need to take the following steps:

1. Find a Chord node $p_1$ by using some external mechanism.
2. Initialise its predecessor and finger pointers by asking $p_1$ to look them up or copy from a neighbour's finger table and find the correct values on its own (the neighbour of the to-be-inserted node $p$ can be retrieved by asking $p_1$ to look up $M(f_P(p))$ ).
3. Contact (existing) peers that need to be aware of the new node and update their predecessor and finger pointers.
4. Transfer resources according to the new responsibilities.

Alternatively, step 3 could be omitted if the Chord nodes periodically run a stabilisation protocol that fixes their finger tables. Note that this would also allow Chord to handle concurrent joins.

To deal with node failures first recall that Chord only needs to maintain correct successor pointers in order to work properly. To overcome failures of successor pointers, each peer stores an additional list of $r$ successors and uses the first live node in that list in such case. The stabilisation protocol mentioned above also ensures that the finger tables are corrected in case of node failures. Meanwhile, alternative nodes to forward routing messages to are found in the finger table (using the preceding finger to the failed one) or in the successor-list.

Both, a node joining and leaving a Chord ring will require $O\left(\log^2(n)\right)$ messages to be exchanged in order to re-establish the routing state of affected nodes.

### 2.1.5. Conclusion

The previously presented DHTs show that realising a distributed hash table can be done in many different ways while still maintaining the goal of efficient resource look-ups (in terms of visited nodes) with only a small fraction of the system known to a node. CAN puts its resources and peers in a d-dimensional coordinate space and requires each node to maintain $2d$ neighbour links. By using simple forwarding based on the geometric distance of a node to a target resource, it achieves average routing path lengths of $(d/4)(n^{1/d})$ hops and allows simple request load balancing by routing requests through different nodes in the direction of a target. Pastry and Chord both map their nodes and resources to a one-dimensional circular name space with addresses between 0 and $2^m - 1$. Pastry further sub-divides those identifiers into levels of $b$ bits and requires a node to maintain a routing table of size $\lceil \log_{2^b}(n) \rceil \cdot (2^b - 1)$ as well as a namespace and neighbourhood set of fixed sizes. Using prefix-routing the average number of hops during routing is $\lceil \log_{2^b}(n) \rceil$ with a worst-case of $O(n)$. Additionally Pastry allows replication on a per-resource level which can be set at a resource's insertion and also provides request load balancing. Chord on the other hand uses finger tables of size $m$ to point to nodes responsible for an exponentially increasing key distance from the nodes' own keys and achieves routing path lengths of $O(\log(n))$ with high probability.

Except for the request load balancing provided by CAN and Pastry, those three DHTs do not implement any explicit load balancing algorithms to balance the load among the nodes but instead rely on the (passive) load balancing provided by consistent hashing with the help of virtual servers (Chord). Without virtual servers this results in each node being responsible for an $O((\log(n) + 1) \cdot 1/n)$ fraction of the available resources which is brought down to $O(1/n)$ using virtual servers. Further improvements (even without virtual servers) are possible using explicit load balancing algorithms (ref. Section 2.4).

Despite their differences, those DHTs all provide a common set of functionality which allows them to be deployed to the needs of the user and be replaced by one another. However, they only allow simple requests like retrieving resources for a set of single keys and lack support for further extensions such as range-queries covered below.

## 2.2. DHTs with Range Queries

One way of implementing range queries is to build them on top of ordinary DHTs such as the ones described above. Multiple dimensions, i.e. possible attributes in range queries, would be reduced to one dimension using space-filling curves and then split into several

ranges which each serve as a single key that is then stored in the DHT [9, 17, 22] (also see Figure 2.3). Such partition needs to be implemented with care because too few fractions lead to poor load balance and too many will increase look-up costs as several of them may need to be retrieved in order to answer a single range query. The same happens for large multi-dimensional range queries. Another disadvantage is the increased maintenance cost this additional layer imposes on the network.

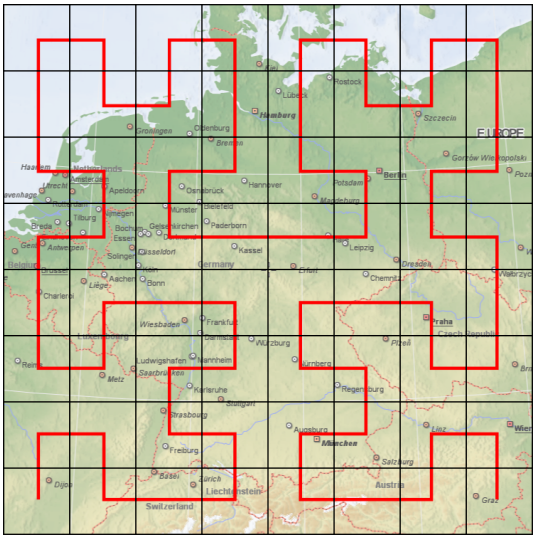



Figure 2.3.: *Example of using a space-filling curve: Patches group coordinates and are then mapped to one dimension according to the progression of the Hilbert curve (approximation level 4) drawn in red. (map: Marble [2], curve: Wikipedia [6])*

Because of these disadvantages, specific DHTs were created that support range queries out of the box. Mercury [12] and Chord$^{\#}$ [39], for example, use key-order-preserving hash functions which allow significantly lower overhead in design complexity and better query performance (in terms of visited nodes per range query) compared to the method depicted above. The following sections will give a short overview of those two implementations which work a bit differently than ordinary DHTs and need to deal with a new set of problems, e.g. significant load imbalance based on the key distribution of their resources.

### 2.2.1. Mercury

Resources in Mercury [12] consist of a list of $(attribute, value)$ pairs with attributes supporting `int`, `char`, `float` and `string` data types. Queries can be created using multiple filters on given attributes which together form a conjunction (disjunctions of filters need to be emulated by issuing a single query for each of them).

Mercury partitions its peers into several *attribute hubs* $H_a$ each denoting a group

that is responsible for a single attribute $a$. The number of hubs should be reasonably small but nodes can be part of several such hubs. Within a single hub $H_a$ each node is responsible for a contiguous range of an attribute $a$ and together the nodes form a circular overlay based on that attribute. This range is assigned to a node when it joins the network. Furthermore, resource $r$ is stored at each node that is responsible for any of its attributes in any hub and is thus sent to every hub $H_a$ with $a \in r$ when it is inserted into the network.

Processing a query first involves a selection of a (single) hub through which the message is routed. Within that hub the query is processed by all nodes which have potential matches. Selecting a hub is therefore crucial for getting a good routing performance and should be done by evaluating the selectivity of each filter of a query. In-hub-routing works by sending the query to the node that is responsible for the first value of the hub's attribute and forwarding it to subsequent nodes still within the range of the query. For that nodes store links to their predecessor and successor nodes within each hub and links to (at least) one node in every other hub. For better robustness to node failures, nodes could alternatively store a (small) number of those links instead of just one.

Similarly to Chord this system would result in $O(n)$, $n = \|P\|$ hops required for processing a query. To provide more efficient routing, $k$ long-distance links are added to the nodes' state (also see the example given in Figure 2.4). Note that $k$ could be different for each node but let's assume that each node contains no more than $2k$ of such links whose construction is given as follows. For each link $l_i$ a node $p$ responsible for the range $[a_l, a_r)$ of attribute $a$ draws a number $x \in [1/n, 1] = J$ using the harmonic probability distribution function $p(x) = (n \cdot \log(x))^{-1}$ for $x \in J$ and stores the node that is responsible for the value $(a_r + (a_{max} - a_{min}) \cdot x)$ within $H_a$. Queries are then forwarded to the node among the long-distance links that minimises the (clockwise) distance to the requested attribute value. Assuming node ranges are uniform, a node responsible for the first value in a given range can be reached with $O\left(\log^2(n) \cdot 1/k\right)$ hops (including the first hop which decides the hub to route in).

Constructing $O(\log(n))$ long-distance links in that manner also enables Mercury to allow uniform random sampling of nodes which is used to gather histograms of system statistics, e.g. load distribution, node-counts and so on. This provides information needed to create the links at all (the number of nodes) and may also be used for implementing a load balancing scheme. Mercury uses a load balancing algorithm similar to the one presented by Karger and Ruhl [30]. Both are described in Section 2.4.2.

Nodes joining Mercury need to complete the following steps:

1. Find a Mercury node $p_1$ by using some external mechanism.

2. Obtain a list of representatives of each hub by querying $p_1$.

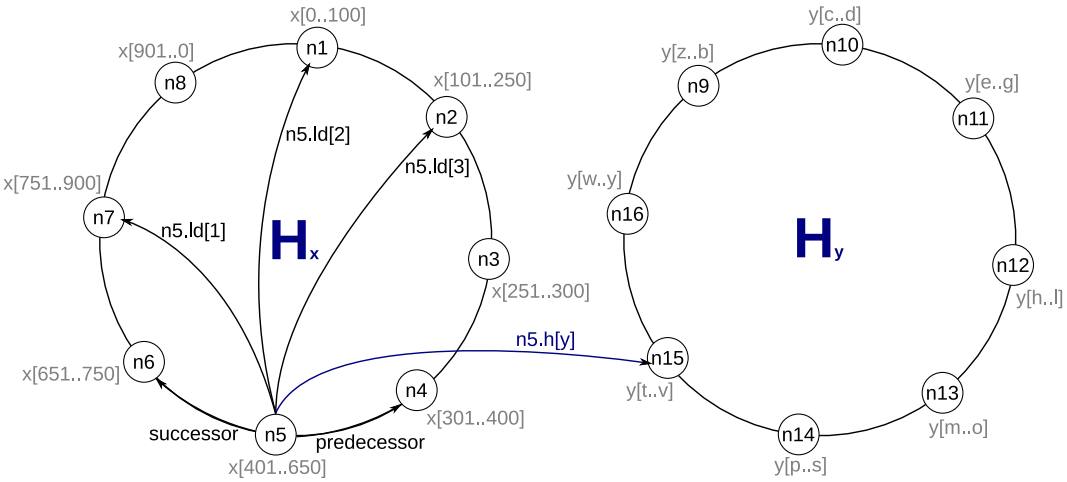



Figure 2.4.: *Mercury network with Hubs $H_x$ and $H_y$ showing node n5's predecessor, successor, cross-hub (h) and $k = 3$ long-distance (ld) links and each node's responsibilities inside its hub.*

3. (Randomly) choose a hub to join at, contact one of its members ($p_2$) and become its predecessor taking half of its values.

4. Copy routing state of $p_2$, create its own long-distance links and get hub representatives different to the ones from $p_2$.

When nodes fail or leave the network, repairing successor and predecessor links is done by using the successor and predecessor link lists mentioned before. Long-distance link failures can be repaired by simply creating new links. Alternatively (and to deal with many link failures) nodes can periodically re-create all links when the number of nodes in the system changes substantially. Finally repairing cross-hub links can be achieved by using a backup link, asking a neighbouring node for its links or (if both fails) using the external mechanism used for node joins.

### 2.2.2. Chord# / Scalaris

Chord# [39] is a variation of the Chord protocol described in Section 2.1.4 and has been implemented in Scalaris [3]. In its basic form it supports one-dimensional range queries but can also be extended for multiple dimensions as described in [39]. It derives from Chord by removing consistent hashing and instead using a key-order preserving hash function to map resources to the identifier space, e.g. by storing the keys in lexicographical order. Nodes are placed at such points of the identifier space that achieve well-enough load distribution. This placement is managed by an explicit load balancing mechanism which constantly changes the nodes' responsibilities according to the current system load. Schütt et al. suggest to use the algorithm presented by Karger and Ruhl [30]

but any of the algorithms described in Section 2.4 is suitable.

In order to keep the routing performance (number of hops required to reach a node responsible for a random resource) at $O(\log(n)), n = \|P\|$, the finger table is constructed differently and operates in the node space rather than the key space. The first finger is the node's successor as in Chord and the $i$'th finger is created by asking the node at finger $(i-1)$ for its $(i-1)$'th finger. This step is repeated as long as fingers point to nodes succeeding the previous finger and not exceeding the current node. The resulting finger table then contains at most $\lceil \log(n) \rceil$ fingers with the longest finger pointing half-way around the node circle, the second longest quarter-way, and so on. It is also guaranteed that no two fingers point to the same node (refer to Figure 2.5 for an example).

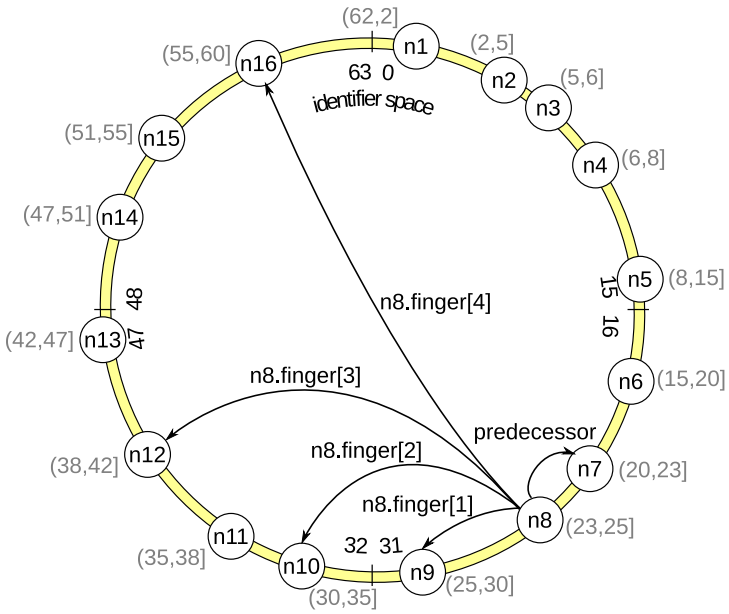



Figure 2.5.: *Chord$^{\#}$ ring with $2^6$ possible IDs and 16 nodes distributed to balance a distribution of resources with hot spots around 6, 24 and 36 (node responsibilities in grey).*

Although the routing algorithm stays the same as in Chord, the number of hops required to reach a desired node is now *guaranteed* to be $O(\log(n))$. This is achieved because fingers in Chord$^{\#}$ definitely decrease the distance to a target node by factor 2 each routing step and not just with high probability. Also (re-)building the finger table requires only $O(\log(n))$ messages compared to $O(\log^2(n))$ in Chord.

### 2.2.3. Conclusion

The aforementioned DHTs show that support for range queries can be achieved with little less or no overhead to ordinary DHTs. In fact, Chord$^{\#}$ even improves Chord's performance by guaranteeing routing performance of $O(\log(n))$ hops and changing only

little compared to Chord. Mercury supports multi-dimensional range queries and uses a so-called *Hub* for each of a resource's attributes. Using $k$ long-distance links it reaches a designated node within $O\left(\log^2(n) \cdot 1/k\right)$ hops. Furthermore by the way those links are generated, Mercury supports random sampling of nodes which it uses to gather system statistics such as estimates of the average load and the number of nodes.

Both DHTs may use arbitrary load balancing algorithms in order to even out the load imbalance that is inherent in the use of order-preserving hash functions. Chord$^{\#}$ suggests an algorithm proposed by Karger and Ruhl [30] while Mercury implements its own variant of this algorithm. Refer to Section 2.4 for a description of those algorithms.

## 2.3. Gossiping

Gossip algorithms can be advantageous for Distributed Hash Tables in several ways. They can for example provide another way of learning random nodes and can be used to adapt the topology of the overlay network to changes. Both is provided by the Cyclon framework [43]. They can also be used to aggregate (global) information with high confidence and low overhead which is of more interest here. Such information includes approximations of values like the minimum, maximum and average load, network size, variance and standard deviation [28].

A generic proactive algorithm calculating those values could for example work by letting each node periodically select another node to exchange information about its local estimate of the desired attribute. Both nodes update their state according to an aggregation-specific *update* function that improves a node's estimate with the help of the other node's estimate. In case of average load computation the nodes could start with local estimates such as their own load. The update function would receive the two estimates $avg_p$ and $avg_q$ of the nodes $p$ and $q$ and both nodes will update their local estimates to $(avg_p + avg_q)/2$ thus achieving a better estimate. Note that the sum of all estimates remains the same as the sum of all nodes' loads and can thus be used to further aggregate the average load the same way. Similarly the minimum and maximum can be calculated by returning $\min(avg_p, avg_q)$ and $\max(avg_p, avg_q)$ respectively and can also be used to collect information about the $k$ minimum/maximum loads (and the nodes holding those values). The variance can be computed by calculating the averages of the nodes' loads and their squares since $\mathrm{Var}(l) = \mathrm{avg}(l^2) - \mathrm{avg}(l)^2$, same for the standard deviation $\sigma_l = \sqrt{\mathrm{Var}(l)}$.

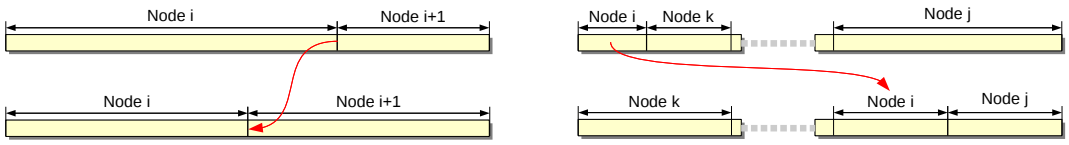
This method provides exponential convergence to the desired value at each node, but best performance can only be guaranteed if the node selection is truly random, e.g. uniform. Nevertheless, this protocol also works by (randomly) selecting nodes from a list of neighbours that is based on the topology of its network or by making random

walks. Experiments conducted in [28] show that the more uniform the random sampling is the faster this algorithm converges.

The following chapters will make use of gossiping algorithms only to retrieve the aforementioned estimates of global information in order to improve load balancing. For an overview of further uses of such algorithms on structured overlay networks refer to [23] and the papers referred there.

## 2.4. Load Balancing in DHTs

As depicted above, some distributed hash tables include (simple) load balancing techniques with some even being immanent in their design, e.g. by using consistent hashing. Their ability to balance load among the system however varies greatly and can generally be improved by deploying a different load balancing algorithm that suits a specific need. That might include a better partition of the address space among the nodes or, more generally, a better balance of an arbitrary *load* like the number of stored resources, a machine's workload including computing power or bandwidth or any other. Also, although not explicitly considered here, one might include node heterogeneity in any balance decision. Other DHTs, in particular those supporting range queries, heavily rely on explicit load balancing mechanisms because the distribution of the stored resources is retained and may be highly skewed.




(a) Slide (b) Jump

Figure 2.6.: *Supported load balancing operations in arbitrary distributed hash tables.*

Note that generic load balancing algorithms can only make use of techniques supported by every DHT and cannot use features specific to a single one. It is for example possible to adjust the responsibility of two neighbouring nodes so that one node takes some resources or responsibilities off of the other. This process is called *sliding* and may be supported directly by the DHT or by removing one of the two nodes and inserting it at an identifier that will result in the desired behaviour. The second generic load balancing primitive is *jumping*, that is a node leaves its current position dropping off all its load and responsibilities to its successor and joining somewhere else in order to take off some of other node's load. Examples for both are presented in Figure 2.6 for a ring-based DHT also showing the changes of every affected node.

The following sections will present several such load balancing algorithms which are

structured as follows. Section 2.4.1 will describe algorithms that try to balance the amount of identifier space each node is responsible for, followed by algorithms trying to balance the actual number of resources in Section 2.4.2. Load balancing algorithms relying on virtual servers or using replication are covered in sections 2.4.3 and 2.4.4. Note that some algorithms' classifications can be ambiguous in which case their main aspects determine the section they are presented in. Further categorisations could be made in order to differentiate between passive and active algorithms, i.e. those that only act on node or item inserts or deletes and those that actively probe the network every once in a while to search for nodes to balance. This additional classification is included in the overview of all presented algorithms given in Section 2.4.5

Note that (until otherwise stated) algorithm descriptions in the following sections will be restricted to ring-based DHTs like Chord which can be done without loss of generality (special care only needs to be taken with the multiple dimensions of a CAN network).

### 2.4.1. Address-Based Load Balancing

Address-based load balancing algorithms aim at partitioning the identifier space uniformly among the participating nodes so that each node is responsible for an equal range of identifiers. This is mostly useful for DHTs using consistent hashing (see Section 2.1.1) where resource identifiers are spread uniformly among the identifier space as well and do not follow a particular distribution. Recall that using uniform and independent hash functions for both nodes and resources still results in an $O(\log(n))$ imbalance. Using virtual servers reduces this imbalance but introduces higher maintenance costs due to the increased number of connections each real host manages. The following sections will describe several address-based load balancing algorithms that will try to reduce the imbalance without using virtual servers.

#### Karger and Ruhl

Karger and Ruhl's address-space balancing scheme [30] first adds an ordering to addresses of the form $\frac{x}{y} = \frac{2b+1}{2^a}$ in the circular identifier interval $I = [0, 1]$ such that $\frac{x}{y} < \frac{x'}{y'} \Leftrightarrow (y < y')$ *or* $(y = y'$ *and* $x < x')$. Equation 2.1 shows the order of some addresses with this specification.

$$0 = 1 < \frac{1}{2} < \frac{1}{4} < \frac{3}{4} < \frac{1}{8} < \frac{3}{8} < \frac{5}{8} < \frac{7}{8} < \frac{1}{16} < \frac{3}{16} < \frac{5}{16} < \frac{7}{16} \ldots \tag{2.1}$$

Secondly, each node maintains a set of $O(\log(n))$ potential positions it can place itself at (solely dependent on the node itself, e.g. on its IP address). It now occasionally checks the address range between each such position and the succeeding active node on

the ring and places itself at the position with the range that covers the smallest address under the given ordering. It can be observed that nodes place themselves at positions close to all small addresses (under the given ordering) which distributes them nearly uniformly among the address space (each node is responsible for an $O(1/n)$ fraction) with high probability thus achieving a ratio between the largest and smallest interval of $O(1)$.

#### Bienkowski et al.

Bienkowski et al. [13] give a load balancing algorithm for ring-based DHTs which estimates the total number of nodes by having each node maintain an additional connection to a random position in the ring (a *marker*) and count the number of markers that fall into the interval of the node itself and some of its successors. Let $i$ and $m$ be the length of the encountered interval and the number of encountered markers respectively, then a node continues to add the succeeding node's data (interval length and number of markers) as long as $m < \log(1/i)$. At the end, $i$ is decreased so that $m = \log(1/i)$ using the information of the last visited node. Let $n_i$ be the solution of $\log(x) - \log(\log(x)) = \log(1/i)$. It follows that with high probability $n_i$ is within constant factors of the real number of nodes $n$ and there are global constants $v, u$ so that $v \cdot n_i \leq n \leq u \cdot n_i$.

Bienkowski et al. now use these values to define three categories of intervals: *short* intervals of length at most $\frac{4}{v \cdot n_i}$, *long* intervals of length at least $\frac{12 \cdot u}{v^2 \cdot n_i}$ and *middle* intervals of lengths in between. Note that the given interval definitions have been chosen so that middle and long intervals have lengths of at least $4/n$ and halving long intervals never creates short intervals.

In the algorithm, nodes with short intervals whose predecessors also cover short intervals try to contact nodes with long intervals with probability $1/2$ and move to a position which splits those nodes' intervals into halves. The search for suitable partner nodes starts at a random position on the ring and continues to look at up to $6 \cdot \log(u \cdot n_i)$ of the succeeding nodes. If routing messages to random destinations is of complexity $R$ then this algorithm achieves a constant ratio between the largest and smallest interval in $O(1)$ rounds with each node incurring a communication cost of $O(R + \log(n))$ per round.

#### Manku et al.

Manku [34] describes an algorithm for choosing appropriate node identifiers upon insertion by contacting the node responsible for a random identifier as well as $c \cdot \log(n)$ of its neighbours (using a small constant $c$) and selecting an identifier so that the largest covered interval among those nodes is split into halves. Node departures are handled

similarly by moving at most one node of the $c \cdot \log(n)$ neighbours of the departing node taking into account the intervals they cover. This algorithm achieves a ratio between the largest and smallest node interval of at most 4 using $\Theta(R + \log(n))$ messages, $R$ being the number of messages needed to contact a random node of the used DHT, and can be tuned to achieve a ratio of $(1 + \epsilon), \epsilon > 0$ at the cost of re-assigning $O(1/\epsilon)$ nodes instead of one node and an increased message cost.

Later Kenthapadi and Manku [31] generalise the scheme of using random and local probes describing algorithms that conduct $r$ random probes each followed by a local probe discovering $v$ of its neighbours and selecting an identifier to split the largest of those intervals. They state that with $r \cdot v \geq c \cdot \log(n)$ the ratio between the largest and smallest interval is at most 8 with high probability where $c$ is a small constant. $n$ can be estimated from the first random probe to ensure that condition. Such schemes use $\Theta(r \cdot R + v)$ messages which allows fine-tuning of the number of local and random probes with respect to the message cost.

#### Giakkoupis and Hadzilacos

Giakkoupis and Hadzilacos [24] employ the *power of multiple random choices* paradigm to create a load balancing algorithm they extend to support heterogeneous nodes. Their algorithm ensures that each key interval a node is responsible for has a length of $1/2^d$ for some constant $d \in \mathbb{N}$ and its endpoints are integer multiples of its length. It adjusts node responsibilities only at *join* and *leave* operations and works as follows: Nodes joining the system first contact the nodes responsible for a logarithmic (in system size) number of points selected uniformly and independently at random. If $1/2^d$ is the length of the interval the node contacts to join the DHT, then $\lceil a_{join} \cdot d + b_{join} \rceil$ identifiers are looked up for some positive system-wide parameters $a_{join}$ and $b_{join}$. The node then splits the largest interval in halves. Similar to this nodes leaving the system will again issue a logarithmic number of requests for nodes ($\lceil a_{leave} \cdot (d + 1) + b_{leave} \rceil$ identifiers if $1/2^d$ is the length of the node's interval), merge the smallest interval with its adjacent interval and assign the leaving node's interval to the node removed due to this merge. As in the algorithm by Manku, a ratio between the largest and smallest node interval of at most 4 is reached but $O(R \cdot \log(n))$ messages need to be exchanged.

In the weighted version of the protocol nodes have an associated *weight* (an integer power of 2 with a system-wide upper bound $W$) proportional to their power, e.g. computing power, bandwidth or storage capacity, and are organised in groups containing adjacent nodes. The same technique as in the unweighted version is now used to balance the intervals of those groups while an additional *group management protocol* handles the balance inside a group and splits or merges groups in order to keep the sum of all weights of its nodes between $W$ and $(2W - 1)$. Therefore the ratio between the largest and smal-

lest group interval is 4 and the overall balance depends on this protocol. It could either achieve a perfect balance inside each group, requiring that up to all its nodes change their responsibilities, or settle for only a few changes to the nodes' associations and achieve an adequate ratio of its nodes' largest and smallest intervals.

### 2.4.2. Item-Based Load Balancing

Item-based load balancing algorithms try to balance the actual distribution of the resources among the nodes and do not rely on a uniform resource distribution in the identifier space. This makes them particularly suitable for range-queriable DHTs that use order-preserving hash functions. Exemplary distributions of resources that result from an alphabetical storage can be seen in Figure 1.1 on page 9 and Figure 4.1 on page 55.

Although some of the depicted address-based load balancing algorithms may be extended to support item-based load balancing as well, e.g the random and local probes used by Manku et al. and the power of multiple random choices paradigm incorporated by Giakkoupis and Hadzilacos could use the node's actual load instead of the covered address space, there are also some specific algorithms handling this category of load balancing which are introduced in the following sections. Also note that item-based load balancing generally allows arbitrary definitions of *load* that could for example take into account a node's capacity, a resource's size and popularity, network latency and more or combine any of those.

#### Karger and Ruhl

Karger and Ruhl's item balancing algorithm [30] is a randomised load balancing algorithm that lets each node $n_i$ occasionally contact another node $n_j$ at random and tries to balance those nodes if their load differs by at least a factor of $0 < \epsilon < 1/4$ which is a system-wide constant. It uses the two generic load balancing primitives *slide* and *jump* introduced above to adjust how an interval is split between two neighbouring nodes and to move a node in order to capture some other node's resources respectively. Note that when node $n_i$ is removed due to such a move, $n_i$'s successor $n_{i+1}$ gets all of $n_i$'s resources which can be a severe burden for $n_{i+1}$.

After $n_i$ and $n_j$ establish contact, Karger and Ruhl's algorithm first checks whether $n_j$ is $n_i$'s successor in which case the two can be directly balanced. Otherwise it tries to balance the most loaded node of the three involved nodes $n_i$, $n_j$ and $n_{j+1}$ (lines 16-21 in listing 2.1) by either balancing $n_j$ with its successor or moving $n_j$ to a position that would result in $n_j$ receiving half of $n_i$'s resources.

Karger and Ruhl prove that if each node contacts $\Omega(\log(n))$ random nodes ($n$ being

```
1  karger_item(DHT d, double ε) {
2   foreach (Node n_i ∈ d) {
3    Node n_j = d.getRandomNodeExcept(n_i); // get another random node
4    if (load(n_i) ≤ ε · load(n_j)) {
5     balance(n_j, n_i);
6    } else if (load(n_j) ≤ ε · load(n_i)) {
7     balance(n_i, n_j);
8    }
9   }
10 }
11
12 balance(Node n_i, Node n_j) { // load(n_i) > load(n_j)
13  if (n_i == n_{j+1}) {
14   slide(n_i, n_j); // equalise load of n_i, n_j
15  } else {
16   if (load(n_{j+1}) > load(n_i)) {
17    slide(n_j, n_{j+1}); // equalise load of n_j, n_{j+1}
18   } else { // load(n_{j+1}) ≤ load(n_i) -> move n_j, balance with n_i
19    jump(n_j, n_i); // move n_j to take half of n_i's resources
20                    // n_j's resources are moved to n_{j+1}
21   }
22  }
23 }
```

Listing 2.1: *Item-based load balancing by Karger and Ruhl.*

the number of nodes in the system, $l_{avg}$ the nodes' average load), this will result in every node having a load of at most $(16/\epsilon) \cdot l_{avg}$ with high probability. Contacting another $\Omega(\log(n))$ nodes will bring all nodes' loads to at least $(\epsilon/16) \cdot l_{avg}$. Increasing the number of nodes to contact when searching for a node to balance with will increase the probability of those bounds.

### Bharambe et al. (Mercury)

In Mercury [12] (ref. Section 2.2.1) a variant of Karger and Ruhl's algorithm is used based on the histograms the DHT provides. Firstly, the *local load* $l_{local}(n_i)$ of a node $n_i$ is defined to be the average load of itself, its successor and its predecessor. Secondly Mercury's histograms are used to retrieve an estimate of the system's average load $l_{avg}$. A node $n_i$ is then said to be *light* if $l_{local}(n_i)/l_{avg} < 1/\alpha$ and *heavy* if this ratio is greater than $\alpha$. This ensures that light nodes have only light neighbours with high probability. If a light node's neighbour is heavy, the two nodes need to balance their load. Additionally heavy nodes probe the system for light nodes which (if found) leave their current position and move to the heavy node in order to take some of its resources. Listing 2.2 shows an

implementation of this algorithm.

```
1  mercury(DHT d, double α) {
2   foreach (Node n_i ∈ d) {
3    if (isLight(n_i)) {
4     if (isHeavy(n_{i+1})) {
5      slide(n_i, n_{i+1}); // equalise load of n_i, n_{i+1}
6     } else if (isHeavy(n_{i-1})) {
7      slide(n_{i-1}, n_i); // equalise load of n_{i-1}, n_i
8     }
9    } else if (isHeavy(n_i)) {
10    Node n_j = d.getRandomNodeExcept(n_i); //get another random node
11    if (isLight(n_j)) {
12     // n_i may be lightly loaded ⇒ use most loaded node of n_i, n_{i-1}, n_{i+1}
13     Node n'_i = getMostLoaded(n_i, n_{i-1}, n_{i+1});
14     if (n_j.isNeighbourOf(n'_i)) {
15      slide(n'_i, n_j); // equalise load of n'_i, n_j
16     } else if (n'_i ≠ n_j) {
17      jump(n_j, n'_i); // move n_j to take half of n'_i's resources
18                       // n_j's resources are moved to n_{j+1}
19     }
20    }
21   }
22  }
23 }
```

Listing 2.2: *Example implementation of the load balancing algorithm by Bharambe et al.*

Provided that $\alpha \geq \sqrt{2}$, the ratio between the highest and average load (as well as the ratio between the average and lowest load) is bound by a factor of $\alpha$. Also note that by tolerating a small skew, i.e. by setting $\alpha$ appropriately, unnecessary item movements due to balance operations during load oscillations can be prevented or at least reduced.

#### Ganesan et al.

Ganesan et al. [21] describe a load balancing algorithm for range-partitioned data that can also be applied to DHTs. It tries to balance load among nodes whenever the load at a node increases or decreases by a certain factor $\delta$ and is called the *Threshold Algorithm*. More precisely they define a sequence of thresholds $T_i = \lfloor c \cdot \delta^i \rfloor, i \geq 1$ for some constant $c > 0$ and whenever a node's load increases to a $(T_m + 1)$ the algorithm tries to adjust its load as follows. If one of its neighbours has a load of at most $T_{m-1}$ the node balances its load with the neighbour with the smallest load. Otherwise the node (let it be $n$) searches for the least-loaded node $n_k$ and, if $n_k$'s load is at most $T_{m-2}$, tells it to leave its current position (moving all its items to its successor) in order to take over half of

$n$'s load. This might result in further recursive invocations of this adjustment at the affected nodes. See listing 2.3 for a complete description of the load balancing procedure performed if such interval is exceeded due to a resource's insertion.

```
1  adjustLoad(Node n_i) { // let load(n_i) ∈ (T_m, T_{m+1}]
2   Node n_j = minLoadedNeighbour(n_i); // least loaded neighbour
3   if (load(n_j) ≤ T_{m-1}) { // adjust neighbours
4    slide(n_i, n_j); // equalise load of n_i, n_j
5    adjustLoad(n_j);
6    adjustLoad(n_i);
7   } else {
8    Node n_k = findLeastLoadedNode();
9    if (load(n_k) ≤ T_{m-2}) {
10    jump(n_k, n_i); // move n_k to take half of n_i's resources
11                    // n_k's resources are moved to n_{k+1}
12    adjustLoad(n_{k+1}); // adjust load of n_k's old successor
13   }
14  }
15 }
```

Listing 2.3: *Method used to adjust load in the* Threshold Algorithm *by Ganesan et al. when a resource insertion results in node $n_i$'s load exceeding a threshold.*

Load adjustments due to resource deletions are handled accordingly: if the node's load drops below a threshold $T_m$ it tries to balance with the highest-loaded neighbour with a load of at least $T_{m+1}$ or tries to move to the highest-loaded node of the system to take half of its elements if that node has a load of at least $T_{m+2}$. This definition reduces too hasty load balancing operations - and thus resource movements - in case of oscillating loads around the threshold $T_m$.

Ganesan et al. show that each $\delta \geq \Phi := (\sqrt{5}+1)/2 \approx 1.62$ can be chosen achieving a ratio of $\delta^3$ between the highest and lowest load. They also state that finding the least and most loaded nodes (line 8 of the *adjustLoad* method in listing 2.3) is not necessary for their results to hold true. Instead the node could move to any node that violates a *GlobalBalance* condition, i.e. for node $n_i$ with a load in the interval $(T_{r-1}, T_r]$ find a node $n_k$ with a load not in the interval $(T_{r-3}, T_{r+2}]$.

#### Aspnes et al.

Aspnes et al. [10] describe a load balancing algorithm for range-queriable data structures that uses arbitrary definitions of *load* and groups keys into buckets with each peer storing some of them (similar to virtual servers). A "free-list" of buckets is maintained, e.g. by using a separate overlay network or storing all such buckets near a fixed key, and is used to take load off of heavily loaded nodes. Buckets are further divided into *closed* and *open*

buckets depending on a certain threshold based on their load. They are furthermore partitioned into groups of two (closed, open) or three buckets (closed, open, closed) and retain this structure by transferring resources as needed, e.g. when a resource is to be inserted into a closed bucket, it moves one of its resources to the neighbouring open bucket and accepts the new resource. If an insertion makes an open bucket closed, one of the buckets from the free list is taken and inserted accordingly (this may transform a group of two into a group of three or a group of three into two groups of two buckets). Deleting resources works similarly and may lead to empty buckets which are returned to the free list during re-structuring.

A bucket's size can be changed by adjusting the threshold that classifies a bucket as closed and requires re-structuring the bucket groups. Such changes will be enforced when the overall system load increases or decreases sufficiently. A centralised version of this algorithm can for example double this threshold when the free list becomes empty and halve it when half the number of nodes is in that list. This results in a worst-case maximum load of 4 times the average load but requires a global controller to adjust the bucket size. It also creates heavy load movements during such migrations.

Aspnes et al. also describe a distributed version which resizes the buckets based on an estimate of the system's average load (gained by gossiping techniques) and prevents simultaneous resource migrations. In order to achieve the latter, buckets are again organised into groups of two pairs (closed, open) or one triple (closed, open, closed). Let $M$ be the load of a closed bucket and $1/4 < e_g < 1/2$, $1/8 < c_g < 1/4$ be random expansion and congestion thresholds for a group $g$ of buckets. $g$ performs localised expansion (doubling its threshold to classify nodes as closed) if its estimated average load $l$ satisfies $l > e_g \cdot M$ and performs localised contraction (halving the threshold) if $l < c_g \cdot M$ allowing each bucket group to migrate separately.

#### Charpentier et al.

An alternative to using gossiping to gather approximations of global knowledge is to use cooperative mobile agents. Those agents, while moving from one node to another, could also be used to initiate load balancing operations. Charpentier et al. [16] use that technique which is solely sketched here to present a rather different approach using techniques from the research field of mobile agents. In their algorithm agents gather approximations of the system's average load. They first start in an initialisation phase which tries to estimate the average load to a certain degree of accuracy. Several agents can be supplied and cooperate with each other to improve their estimates and speed-up their initialisation phase by exchanging their data. In their second phase agents order nodes with loads higher than their calculated average to migrate some of their resources to either or both of their neighbours thus achieving some load balancing.

### 2.4.3. Virtual-Server-Based Load Balancing

Several research papers focus on DHTs that use multiple virtual servers on each real peer and balance load by moving those virtual servers. As depicted above, using consistent hashing and deploying $\Omega(\log(n))$ virtual servers (in a system with $n$ real peers) randomly along the identifier space will lead to each peer being responsible for an $O(1/n)$ fraction of the stored resources (ref. Section 2.1.1). Another advantage is that each peer can easily adjust its load by moving some of its virtual servers to any other peer instead of just being able to shed load to its neighbours or move itself. This however comes at the cost of maintaining $\Omega(\log(n))$ additional network connections, an increased number of routing hops while looking up random resources as well as increased churn on node failures which are generally the reasons why the use of virtual servers is not preferred. Nevertheless the following sections show some algorithms that make use of this technique as it has been of interest in past research and still is.

#### Rao et al.

Rao et al. [36] present three different load balancing schemes that try to move virtual servers from heavily to lightly loaded nodes. *Load* in their case can be any single resource, e.g. storage, bandwidth or CPU capacity, so this algorithm could also be classified as an item-based algorithm. Nodes are considered heavy if their current load exceeds their target load and are otherwise light. Balancing a heavy peer $p_h$ with a light peer $p_l$ will move the virtual server $v$ to $p_l$ that does not make $p_l$ heavy and is the lightest virtual server making $p_h$ light or the heaviest virtual server in case $p_h$ cannot be made light that way. The three schemes now differ in how heavy and light nodes are matched in order to start balance operations.

The *One-to-One Scheme* lets light nodes occasionally probe other nodes at random and virtual servers are transferred if the probed node is heavy. Letting only light nodes try to contact heavy nodes (instead of heavy nodes trying to contact light nodes) will not introduce additional workload on heavy nodes and will therefore not increase the risk of highly loaded systems getting overloaded or trashed due to unnecessary and unsuccessful probes. The second scheme implements a *One-to-Many* matching technique by letting light nodes register with one of $d$ system-wide *directories* which are also maintained by the DHT, e.g. by storing a directory at the node responsible for its key. Heavy nodes may now look at such directories and pick the least loaded node to shed some of their virtual servers to. This scheme is now extended by registering heavy nodes with those directories as well and letting the node that stores a directory occasionally match heavy and light nodes in a *Many-to-Many* fashion to optimise the load balance even more.

Rao et al. [36] now analyse their algorithms in a static setting, i.e. an initial load is

distributed among a fixed number of nodes. Neither new resources are added or deleted, nor are nodes. Using the total number of moved load and the number of probes the algorithm needs to achieve a state with only light nodes, they conclude that the amount of load moved is not dependent on the used scheme although the one-to-one scheme needs more probes in order to succeed. Also in the one-to-many scheme with 16 nodes per directory most heavy nodes succeed in getting light by contacting only one directory.

#### Godfrey et al.

Godfrey et al. [25] further extend the many-to-many scheme introduced by Rao et al. and analyse it in dynamic networks, that is nodes and resources are dynamically added and removed. In their version, each node initially contacts a random directory and sends its load and capacity information and repeats to send this data to another random directory whenever it transfers any load. Additionally if a node becomes heavy, i.e. its current load is above a given emergency threshold, it contacts its chosen directory and tries to shed load immediately. Directories on the other hand create schedules for transferring load among all their known nodes and execute them periodically. They also perform the immediate balance requests issued by their nodes. Virtual servers are transferred based on a greedy algorithm that moves each heavy node's lightest server to a common pool and matches each of those servers (starting from the heaviest) to the node that suffers the least impact of such transfer relative to its capacity.

Their evaluation shows that using periodic load balancing with an emergency threshold allows selecting significantly larger execution periods and thus achieves better node utilisation with less load movement. They also show that the number of directories deployed does not severely affect the achieved node utilisation and that by using 16 directories node utilisation is only 3% higher than in a centralised approach (1 directory).

#### Chen and Tsai

In a recent paper, Chen and Tsai [18] try to improve the many-to-many scheme depicted above by introducing ant system heuristics (ASH) to re-assign the virtual servers. Their algorithm, called *Dual-Space Local Search* (DSLS), describes an iterative procedure consisting of three stages:

1. Construct an initial solution for the current iteration using the ASH algorithm.

2. Improve this solution by evaluating further local solutions in its neighbourhood both in terms of load balance and movement cost.

3. Update pheromone trails if a better solution was found.

In step 1, pheromone variables $\tau_{ij}$ are used to denote node $j$'s desire of taking virtual server $i$ (with a given maximum desire $\tau_{max}$ used if $i$ is already assigned to $j$). Now with probability $p_0$ server $i$ is assigned to another node $k$ with enough capacity and a maximal $\tau_{ik}$ and with probability $(1 - p_0)$ it is assigned to another node with enough capacity under the probability function $p_{ij} = \frac{\tau_{ij}}{\sum_{k \in P} \tau_{ik}}, \forall j \in P$. $p_0$ thus allows fine-tuning the algorithm between exploiting already found solutions (high values) or explore new variations (low values). If all nodes are fully occupied, a random assignment is used.

Step 2 first tries to find a good solution (if the first step hasn't found any yet) by shifting load from overloaded nodes to their neighbours and eventually further. If a feasible solution is found, i.e. no node exceeds its capacity, a cost-reducing function tries to minimise movement costs by moving servers back to their original location if possible.

Chen and Tsai's evaluation shows that their variation achieves better load balance than the previous implementation of the many-to-many scheme and moves only little more resources than is necessary in order to balance their scenarios. In contrast to the previous work though the number of deployed directories has a severe affect on their algorithm which performs better with less directories.

### Ledlie and Seltzer

Ledlie and Seltzer [32] use the multiple random choices paradigm to deploy an algorithm that generates $k$ different verifiable identifiers for each node at which the node can create virtual servers. This algorithm, called *k-Choices*, primarily works at node joins where a node chooses a target load and a maximum number of $k/2$ virtual servers to create. It then creates new virtual servers as long as its target load and the total number of servers are not exceeded. Each such join happens at the one of the still available identifiers which results in the lowest cost in terms of difference between the target and real loads of the two affected nodes. Additionally to this *passive* implementation, k-Choices can also work *actively* and re-select identifiers at any time (not just at node joins) if the change induces a big enough benefit, e.g. a node is over- or underloaded. Also nodes could create more virtual servers or delete some.

During their experiments, Ledlie and Seltzer show that their active algorithm achieves a good load balance for $k = 8$ identifiers. It was still able to do so under highly dynamic networks and with large amounts of skewed load.

### Godfrey and Stoica

A technique described by Godfrey and Stoica [26] can reduce the additional cost induced by virtual servers by placing each peer's $k$ virtual servers in a $\Theta(k/n)$ fraction of the identifier space instead of spreading them randomly. This way they can share a single set

of network links. Godfrey and Stoica show that the ratio between highest and average load can be reduced to $(1 + \epsilon)$ for any $\epsilon > 0$ although increasing route lengths and number of links to maintain by only a constant factor (if applied to an arbitrary DHT). They further evaluate an implementation based on Chord using $2 \cdot \log(n)$ virtual servers and achieve a ratio of less than 4.

### 2.4.4. Load Balancing using Replication

Sometimes replication is not only used to ensure resource availability in such highly dynamic networks as DHTs, but is also suggested to carry out simple load balancing by placing replicas at lightly loaded nodes thus evening out the overall imbalance. However replication is not in the main focus of this work, so the following sections only briefly describe some of the available techniques.

#### Byers et al.

Byers et al. [14] combine load balancing and replication techniques by making use of the *power of two choices* paradigm. Each resource is assigned $d$ different identifiers using $d$ different hash functions. It is then associated with the $k$ most lightly loaded peers responsible for any of the identifiers. The rest of the peers may store redirection pointers to those storage locations to simplify searches (resulting in increased maintenance costs). Otherwise searches will be carried out for all possible identifiers in parallel.

#### Xu and Bhuyan

Xu and Bhuyan [44] collect information about the stored resources' access history and use this as their definition of *load* to balance the impact of requests to popular resources among the (possibly heterogeneous) nodes. They first describe a static load distribution algorithm which splits a node's zone into two halves depending on its load (unlike its key range) when a new node arrives. Secondly a dynamic load distribution algorithm steps in when nodes become overloaded and balances their load among neighbouring nodes (possibly including their neighbours as well, and so on). In a final step they specify a replication scheme which enhances their access history with network topology information and replicates resources to peers near a group of peers with the highest request rates. Requests from those peers can now be redirected to the replicas to reduce the access latency and the original node's (access) load.

#### Pitoura et al.

Pitoura et al. [35] design a DHT which uses replication for efficient range query processing and load balancing of resource accesses. For this they use a so-called *multi-rotation hash*

*function* that assigns a resource multiple identifiers of an identifier ring. Whenever a node becomes overloaded (in terms of request access load), it will add additional replicas at some of the available identifiers and issue replication requests to its neighbours' resources as well. Replicating whole arcs of the identifier space will improve performance of range-queries that start on a replicated resource location and continue at its original location's successor because its data is replicated to the current location's successor as well. It should also be mentioned that the underlying algorithm of this system can be applied to different DHTs as well.

### 2.4.5. Conclusion

As can be seen from the previously described algorithms, the goal of balancing *load* in a distributed hash table can be tackled from several angles. There are at first algorithms which try to balance the address-space, i.e. give each node responsibility for an equal part of the identifier space. Those algorithms rely on the uniform distribution of the resources' identifiers in order to give each node an equal amount of resources to store. They are therefore not suitable for range-queriable DHTs that are based on order-preserving hash functions. Another kind of algorithms tries to balance an arbitrarily defined *load* by moving items (resources) and nodes accordingly and are thus called *item-based*. Those algorithms mostly concentrate on *load* being the size of all resources stored by a node or the stored resources' popularity (number of requests). Heterogeneous algorithms set those into relation to a node's capacity. An overview of all presented algorithms and their classifications is given in Table 2.1.

| **Algorithm** | **item/addr.** | **active/passive** | **Notes** |
|---|---|---|---|
| *Karger & Ruhl (1) [30]* | address-based | active | |
| *Bienkowski et al. [13]* | address-based | active | estimates the network's size |
| *Manku [34]* | address-based | passive (node) | |
| *Kenthapadi & Manku [31]* | address-based | passive (node) | |
| *Giakkoupis & Hadzilacos [24]* | address-based | passive (node) | a weighted version exists |
| *Karger & Ruhl (2) [30]* | item-based | active | |
| *Bharambe et al. [12]* | item-based | active | uses estimate of average load |
| *Ganesan et al. [21]* | item-based | passive (item) | uses least/most loaded nodes |
| *Aspnes et al. [10]* | item-based | passive (item) | uses estimate of average load |
| *Charpentier et al. [16]* | item-based | active | uses mobile agents, average load |
| *Rao et al. [36]* | item-based | active | uses virtual servers (VS-based) |
| *Godfrey et al. [25]* | item-based | act+pass(item) | VS-based |
| *Chen & Tsai [18]* | item-based | act+pass(item) | VS-based, ant system heuristics |
| *Ledlie & Seltzer [32]* | item-based | act+pass(node) | VS-based |
| *Godfrey & Stoica [26]* | address-based | passive (node) | VS-based |
| *Byers et al. [14]* | item-based | passive (item) | uses replication |
| *Xu & Bhuyan [44]* | item-based | active | replication, file access history |
| *Pitoura et al. [35]* | item-based | active | uses replication |

Table 2.1.: *Overview of the presented load balancing algorithms.*

Comparing the algorithms' performance with each other is however not that simple. At first, several metrics for *performance* exist. Some use the ratio between the highest and the average load of the system, some the ratio between the highest and lowest load. Others measure the variance of the fraction of address-space the nodes are responsible for or the deviation from the average load. Secondly values are sometimes given in Landau notation thus hiding constant factors that matter when comparing otherwise equally well performing algorithms. Additionally the costs of achieving a certain performance, e.g. item movements or the number of interchanged messages, are often not mentioned either or are not comparable.

Further impairing the lack of comparisons is the fact that no common test scenarios have been agreed upon which each algorithm can be tested with and that resemble the different use cases of DHTs. For example some papers evaluate their algorithm(s) by simulating them in a static setting, i.e. an initial load is distributed among a fixed number of nodes and neither new resources nor nodes are added or deleted. Those simulations mostly use load distributions that follow a certain probability distribution, e.g. normal or exponential distributions. Other algorithms, especially passive ones, need dynamic simulations as they only act when nodes or items are inserted or deleted. This provides even more flexibility in setting up a test scenario. Furthermore most algorithms also allow some fine-tuning by setting their parameters according to the scenario and the needs of the user.

# 3. Improving load balancing algorithms with global information

This chapter will present the general concepts of adding estimates of global information to existing load balance algorithms in order to improve their performance. It will start introducing the underlying system model that describes the DHT the algorithms can work on, the operations it supports and any further assumptions. It will then present the (original) algorithms that have been chosen to exemplify how such information is integrated and which affect it has. The final section will cover the changes that were made to those algorithms and the ideas behind them. Detailed descriptions of all mentioned algorithms in pseudo-code can be found in appendix A.

## 3.1. System Model

Let $d$ be an arbitrary DHT that operates in the identifier space $I = [0, m) \subset \mathbb{N}$ that wraps around at the end and forms a ring. On this ring a *clockwise* direction is defined as going towards increasing keys possibly wrapping around at $m \rightarrow 0$. An *interval* $(a, b]$ of this ring includes all keys that are encountered when traversing the ring clockwise starting at (but excluding) $a$ and stopping at $b$ (inclusive). Note that it is possible that $a > b$ in which case $(a, b]$ covers all keys greater than $a$ and less than or equal to $b$.

Let $d$ consist of $n$ homogeneous nodes (peers) $p \in P$, each being responsible for an interval $(a_p, b_p] \subset I$ so that exactly one peer is responsible for any identifier $id \in I$:

$$\begin{aligned} \forall p_i, p_j \in P,\ p_i \neq p_j : \quad (a_{p_i}, b_{p_i}] \cap (a_{p_j}, b_{p_j}] &= \emptyset \\ \bigcup_{p \in P} (a_p, b_p] &= I \end{aligned}$$

The successor of peer $p_i$ is the peer $p_j$ which is responsible for the following interval, i.e. $a_{p_j} = b_{p_i}$. A predecessor is defined analogously. From the previous definitions follows that there is exactly one successor and predecessor for each peer. Connections between those are maintained so that predecessor and successor pointers form a double-linked list. Additional connections to further peers exist in order to allow efficient routing from any peer of the network to any other.

The DHT stores (arbitrary) resources, i.e. items, that are mapped to $I$ using an order-preserving hash function and stored at the peer which is responsible for their identifier. Each peer has a load $l(p)$ equal to the number of items it stores. The imbalance of the DHT is defined as the standard deviation of its nodes' loads, i.e.

$$\begin{aligned} \mathit{imbalance}(d) &:= \sigma_l \\ &= \sqrt{\frac{\sum_{p\in P}\left(l(p)-\mu\right)^2}{n}}, \quad \mu = \frac{\sum_{p\in P} l(p)}{n} \\ &= \sqrt{\frac{\sum_{p\in P} l(p)^2}{n} - \left(\frac{\sum_{p\in P} l(p)}{n}\right)^2} \\ &=: \sqrt{\mathrm{avg}(l^2) - \mathrm{avg}(l)^2} \end{aligned}$$

The first goal of a load balancing scheme is to reduce this imbalance. For this, two types of operations are supported: *slide* and *jump*, previously described in Section 2.4. They effectively adjust some nodes' responsibilities which implies item movements. The second goal of a balance algorithm is thus to reduce the number of moved items in order to reach a certain imbalance. There is a trade-off between these two goals because a smaller imbalance can generally only be reached by moving more items.

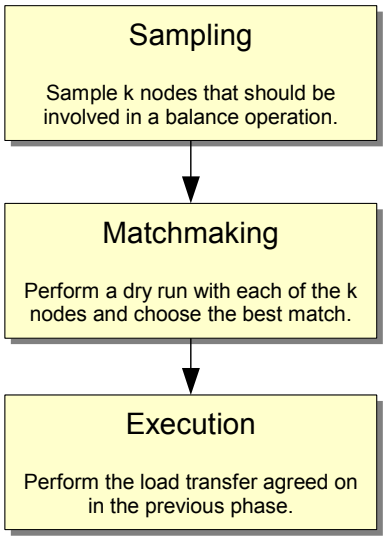
Further operations include the retrieval of a node's successor and predecessor and the ability to get a random node of the whole system. The latter might be natively supported by the DHT or can be implemented by using random walks (suggested in [12]) or gossip algorithms [23] or by generating a random ID and returning the node responsible for it (assuming uniform node responsibilities). Balance algorithms heavily rely on those methods. Some also need to retrieve the node that is responsible for a given key but this is not the case for the algorithms in this section.

Every node also runs a gossiping algorithm that continuously calculates estimates of certain global information such as the system's size, average load and standard deviation (ref. Section 2.3). It is assumed that those estimates are within a certain *error rate* (in percent) of the exact values and that they are re-calculated every once in a while in order to stay within this bound.

During the following sections it will also be assumed that the system is static, i.e. the total number of items and nodes in the system stays constant. Algorithms will thus start their operation on a DHT whose nodes' responsibilities are uniformly distributed among the identifier space. Stored resources follow a certain distribution the algorithms don't know about. The static nature of the system requires that the algorithms actively probe for other nodes to balance with and cannot operate passively.

## 3.2. Algorithms

In order to analyse the effect of adding estimated global information to existing load balancing schemes, two novel active item-based balance algorithms were chosen and equipped with this knowledge. Those algorithms include the item-balancing scheme introduced by Karger and Ruhl [30] (from here on referred to as "`karger`") and the algorithm used in Mercury [12] ("`mercury`"), both described in Section 2.4.2.




At each execution and for each node, both sample one random node and then decide whether to balance with it and how many items to transfer between the nodes. At most three nodes are involved in such a decision and only they can change their load: the two neighbouring nodes in case of a *slide* operation and in case of a *jump* the moved node, its (original) successor and the node jumped to. Karger and Ruhl also suggest to perform multiple random samples and balance with the best node among them but do not describe how to decide for the best. Here, algorithms with multiple randomly sampled nodes will operate in three phases as presented in the figure to the right. The first step involves sampling a given number of (unique) nodes uniformly at random. With each such candidate node, the balance algorithm is simulated and the best among them (or none) is chosen. It follows the execution of the algorithm with this node (if there is one). Listing 3.1 shows the modified `karger` algorithm and the changes compared to the original scheme described by Karger and Ruhl. Detailed algorithm descriptions in pseudo-code are given in appendix A.

The way the *best match* among the candidates is chosen is crucial for the algorithm. Here it is defined as the node that improves the standard deviation of the system's load the most. If no such node exists, i.e. no balance operation would decrease the standard deviation, no operation is performed. Note that this method will also be used if only one random sample is requested. Also note that only local knowledge is required to take that decision. This can be easily deduced from the following observations.

First recall that $\sigma_l = \sqrt{\frac{\sum_{p\in P} l(p)^2}{n} - \left(\frac{\sum_{p\in P} l(p)}{n}\right)^2}$ and that the sum of the loads as well as the number of nodes, $n$, is not changed by performing either *slide* or *jump*. Thus only the first sum needs to be examined. The candidate node that reduces this sum the most will reduce the standard deviation the most. If, for example, the involved nodes' loads $l(p_i), l(p_j)$ change to $l'(p_i), l'(p_j)$, then the change of the sum can be determined as $l'(p_i)^2 + l'(p_j)^2 - l(p_i)^2 - l(p_j)^2$ (similarly with three nodes).

```
1  karger_item(DHT d, double ε, int samples) {
2   foreach (Node n_i ∈ d) {
3    // get k unique random nodes that are not equal to n_i:
4    Nodes candidates = d.getUniqueRandomNodes(n_i, samples);
5    // get best candidate by simulating the algorithm:
6    Node n_j = getBest(d, ε, n_i, candidates);
7    if (d.exists(n_j)) {
8     if (load(n_i) ≤ ε · load(n_j)) {
9      karger_balance(n_j, n_i);
10    } else if (load(n_j) ≤ ε · load(n_i)) {
11     karger_balance(n_i, n_j);
12    }
13   }
14  }
15 }
16
17 karger_balance(Node n_i, Node n_j) { // load(n_i) > load(n_j)
18  if (n_i == n_{j+1}) {
19   slide(n_i, n_j); // equalise load of n_i, n_j
20  } else {
21   if (load(n_{j+1}) > load(n_i)) {
22    slide(n_j, n_{j+1}); // equalise load of n_j, n_{j+1}
23   } else { // load(n_{j+1}) ≤ load(n_i) -> move n_j, balance with n_i
24    jump(n_j, n_i); // move n_j to take half of n_i's resources
25   }
26  }
27 }
```

Listing 3.1: *karger with multiple samples, changes to the original algorithm in red*

## 3.3. Adding global information

The two previously introduced algorithms are now equipped with additional knowledge about estimated global information. The accurateness of it is unknown to them though. The following sections will introduce each single estimate that is added, how it is used and the ideas behind. A final section will describe how several of those estimates can be used together and will present the idea of self-tuning algorithms.

### 3.3.1. Average load

The key to reducing the number of moved items is to understand which of them do at least have to be moved in order to reach a state with minimal imbalance. In such a state every node has the same load as the average load among all nodes (only then is $\sigma_l = 0$). This ideal state however does sometimes not exist for a given total amount

of load and a number of nodes, e.g. if the total load is not divisible by the number of nodes. If it exists and arbitrary item movements are possible, an optimal number of load transfers can be reached by moving items only from overloaded nodes (those with a load higher than the average) to underloaded nodes (less load than the average) and never make them overloaded. Also overloaded nodes would only move so many items in order to become balanced. Figure 3.1 shows the load that would get moved in such case. However, it is not possible to apply this algorithm to DHTs following the given model since, for example, items can not be moved arbitrarily.

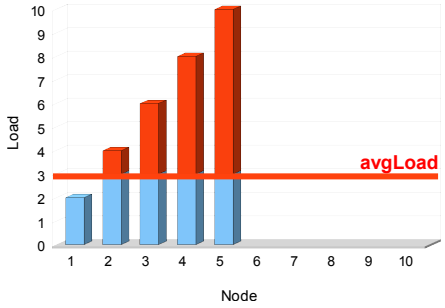



Figure 3.1.: *Load that needs to be moved in order to reach a minimal imbalance (in red).*

The following sections will present some ways, information about the average load can be added to existing load balance algorithms. They try to incorporate some of the ideas of the optimal item movement pictured above.

### Variant `avg1`

As one of the first things, one might observe that existing load balancing schemes often try to even out the balance of two neighbouring nodes. This moves load off of overloaded nodes quickly but often results in unnecessary item movements, e.g. transferred items would need to be moved again if the receiving node gets overloaded after the balance operation. Those movements can be reduced if the amount of items that can be moved is lower than or equal to the (estimated) average load.

This variant will thus hook into the algorithms to replace the decision about how much load - and thus items - is to be moved from one node to another. It will never move more than the (estimated) average load from the heavier loaded node to the lighter one. Algorithms with this implementation will have `_avg1` appended to their name. It is the hope that this will reduce the number of item movements while not changing the algorithm's operations too much and thus keeping the balance results the original algorithms expose.

#### Variant `avg2`

A natural extension to `avg1` would further restrict item movements when balancing neighbouring nodes and never make the light node heavy. Additionally this variant will only move items from heavy nodes (those with a load higher than the average) to light nodes (less load than the average) and limit the number of transferred items to the minimum required in order to make the heavy node balanced (load equal to the average). Algorithms with this implementation will have `_avg2` appended to their name.

While less item movements can be expected from this variation, evaluations will show whether those changes are too invasive in order to reach the same imbalance.

#### Variant `avg3j`

Both of the previous variations will not not have any influence on the decision whether a node is jumping to another position or not. In `avg2`, for example, this results in some light nodes becoming heavy nonetheless because a jumping node's items are transferred to them. This is not the case in an ideal item movement though.

A third implementation will thus restrict jumping so that this does not happen. If a node makes its successor heavy by jumping or if the successor is already heavy, no balance operations is performed. This variant can be combined with either of the previous two variants (which are not already included!) and appends `_avg3j` to the algorithm's base name. It will reduce unnecessary item movements by restricting jumps. However without jumping, reducing the system's imbalance would take significantly longer and is sometimes not possible when further restrictions apply. It is unclear whether the same imbalance can be reached as with the original algorithm, especially since a jump is normally doing more good than bad.

### 3.3.2. Standard deviation and system size

As the standard deviation measures the system's imbalance, it can also be used to quantify the quality of a single load balance operation. Assuming the average load does not change and the current standard deviation $\sigma_l$, the system's size $n$ and the load changes of the current balance operation are known, the new standard deviation can be calculated as follows. Let $p_i$ and $p_j$ be the peers that changed their load from $l(p_i), l(p_j)$ to $l'(p_i), l'(p_j)$, then:

$$\begin{aligned}\sigma'_l &= \sqrt{\frac{\sum_{p\in P\setminus\{p_i,p_j\}} l(p)^2}{n} + \frac{l'(p_i)^2}{n} + \frac{l'(p_j)^2}{n} - \left(\frac{\sum_{p\in P} l(p)}{n}\right)^2} \\ &= \sqrt{\sigma_l^2 - \frac{l(p_i)^2}{n} - \frac{l(p_j)^2}{n} + \frac{l'(p_i)^2}{n} + \frac{l'(p_j)^2}{n}}\end{aligned}$$

#### Variant `stddev2`

This variation will hook into the algorithms to replace the decision about which node from a list of candidates is the best and thus also decides whether to balance or not. Aside from the original algorithm implementations mentioned above, an estimate of the standard deviation of the nodes' loads as well as the system's size is retrieved and used to calculate an estimate of the new standard deviation. Additionally, the algorithms are equipped with one more parameter, `s`, and the best node among some candidates is then determined as the node that, if used in a balance operation, would improve the standard deviation the most and by at least a factor $s/n$. If no such candidate is available, no balance operation is performed. Algorithms with this implementation will have `_stddev2` appended to their name.

This rationale behind this idea is to omit balance operations that do not improve the overall balance enough compared to the big picture. It will thus tolerate a small skew as balance algorithms mostly already do. This way the algorithm can concentrate on bigger improvements and will (maybe) handle the smaller ones at a later time when they are responsible for the imbalance. With this restriction it can be expected that almost the same balance can be reached while moving fewer items. However special care has to be taken on the decision of the value of $s$ which needs to be adapted to the algorithm this variant is applied to (and possibly also to the load distribution in the DHT, in particular the overall total load). If the algorithm generally only moves a few items, the affect on the imbalance can't be as high as an algorithm moving more items and thus $s$ needs to be lower in order not to block too many operations. This will be evaluated as well.

### 3.3.3. Combined variants

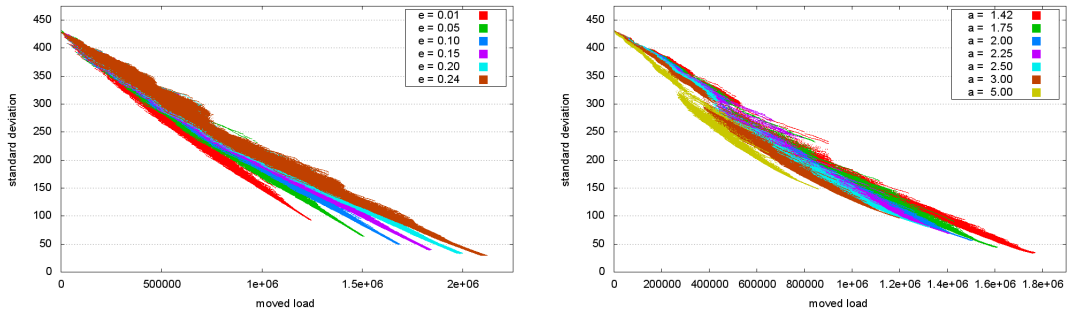
Several of the ideas pictured above can be combined to form new variants of an algorithm. This new variant is then expected to show even better results since two different measures of improving the imbalance and/or reducing the number of moved items are applied together. One of those combinations has already been suggested above: applying `avg1` and `avg2` to `avg3j`. Both resemble the ideal item movement more than a single method alone while `avg3j_avg2` resembles it the most. Less item movements can thus be expected and the simulation will show the influence on the imbalance reached at the end. Additionally to this combination, `stddev2` can be applied. It is expected to improve every aforementioned variant by setting an appropriate $s$ and thus only executing the most useful balance operations.

The following combinations are possible and will be evaluated:

| | | | |
|---|---|---|---|
| `avg1_stddev2` | `avg3j_avg1` | `avg3j_avg1_stddev2` | `avg3j_stddev2` |
| `avg2_stddev2` | `avg3j_avg2` | `avg3j_avg2_stddev2` | |

### 3.3.4. Self-tuning algorithms

Most algorithms can be fine-tuned by adjusting certain parameters, i.e. $\epsilon$ in `karger` and $\alpha$ in `mercury`. Different values for those parameters will result in different performances in each scenario as can be seen in Figure 3.2 showing plots of moved items vs. imbalance data points of the `karger` and `mercury` algorithms with different parameters. This allows the analysis of the progression of the algorithms during the whole simulation and shows which imbalance can be reached by moving a certain amount of items. They have been taken from Section 4.4 and will be analysed in more detail there.




(a) `karger` (b) `mercury`

Figure 3.2.: *Balance results for* ***karger*** *and* ***mercury*** *with different parameters, scenario: Wikipedia page titles (en), error rate:* 25%.

It can be seen that algorithms with parameters tolerating a bigger skew in the load distribution, i.e. larger epsilon and smaller alpha respectively, start off better than the others. Every imbalance they reach, they reach by moving less items. This might be exploited for self-tuning algorithms in a way that the algorithm starts off tolerating a bigger skew and sets its parameter(s) for better imbalance results during its execution and according to the current distribution of load among the nodes. Estimates of global information will be used to gain knowledge about this distribution. This change will hopefully result in the algorithm starting off with the good results of parameters tolerating a bigger skew and continue the way the others do, finally arriving at the best imbalance the original algorithm can achieve but by moving less items.

This idea has been applied to the `karger` and `mercury` algorithms and results in the following calculations that set the algorithm's parameters for each node during their execution[1]:

```
1  epsilon = bound(0.01, l_avg / max(l_avg + σ_l, l_max - σ_l), 0.24);
2  alpha   = bound(1.42, (l_avg + σ_l) / l_avg, 10.00);
```

[1] **bound** sets the value to the first argument if it is smaller and to the last if it is larger

The idea behind both is that if the standard deviation is high only those nodes should be balanced that are responsible for this high value, i.e. those nodes that have the highest loads in the system. The trick is not to set this bound too restrictively since then too few nodes could be matched with each other. This is why for `karger` a node should only be balanced with an other if their load differs by at least a factor of $f_1 = l_{avg}/(l_{max} - \sigma_l)$, i.e. the average load divided by the maximal load minus the standard deviation. The first idea was to use a factor of $f_2 = l_{avg}/(l_{avg} + \sigma_l)$ but this did not achieve the anticipated results so a more restrictive approach was taken by using $f_1$ which is usually smaller than $f_2$. Using the maximum of the two in the calculation above is purely technical and covers the case if $f_1 > f_2$.

The same idea could unfortunately not be applied to `mercury` since it operates differently. At first, balancing is coupled to the average load and occurs only between heavy and light nodes, i.e. nodes with their local load being greater than $\alpha$ or lower than $1/\alpha$ respectively. Thus setting *alpha* influences both bounds instead of a (more flexible) factor as in `karger`. It therefore needs to be set with more care. Additionally the meaning is different which is why it has been set to $(l_{avg} + \sigma_l)/l_{avg}$.

In contrast to the other variants above, a self-tuning variant needs to be adjusted to the way its algorithm operates and the parameters it uses. It can thus not be formalised independently of the algorithm which is why the different implementations will probably show different behaviours. The achievements of one self-tuning algorithm can therefore not necessarily be transferred to another. However it may be evaluated whether the idea behind is good. Self-tuning algorithms will have `_self` added to their names.

# 4. Evaluation

The following sections will evaluate the performance of the proposed algorithms from the previous chapter. The evaluation has been carried out by implementing a simulator and running these algorithms on different scenarios which will be introduced below. The metrics used for this are explained alongside with a brief overview of the simulator program itself. Finally the collected data will be analysed and further simulations will show how robust the algorithms are in regard to several aspects of the simulations.

## 4.1. Simulation scenarios

In order to evaluate the proposed algorithms, several scenarios were set up for the algorithms to balance. Real-life applications often store data with keys made of words and numbers, e.g. titles of articles or names of files. Those alphanumeric keys therefore usually follow the distribution of the words of a certain language. Scenarios resembling real-life applications have thus been set up by taking the list of page titles of the English, German and French Wikipedia (page title dumps from 16/08/2009 (en), 10/08/2009 (de) and 19/08/2009 (fr) [4]). Note though that the English Wikipedia (as well as the German and French one) does not only have English page titles but instead describes topics of all languages using English. Nonetheless all three exhibit different distributions as the scenarios' plots in Figure 4.1 show. Additional scenarios include key-distributions following a normal distribution with different parameters as well as an exponential distribution.

Each of the mentioned key distributions was included into a scenario with 5, 10, 20 or 40 thousand nodes which was set up with an initial load of 0.5, 1, 2 or 4 million items. Every node was given an identifier in the circular ID space $I = [0, 2^{64} - 1)$ uniformly at random. In case of alphabetical distributions, page titles were then hashed to an identifier using an order-preserving hash function and from that list of unique keys the requested number of items was drawn uniformly at random. Scenarios following a normal or exponential probability distribution create keys by drawing them randomly according to their distribution and create items accordingly. Those items are finally inserted at the nodes responsible for them. It might happen that a key is drawn multiple times, in which case a neighbouring key is tried or a new key is re-drawn until no conflicts occur. Figure 4.1 shows the resulting load distributions among the nodes in key order

for scenarios with 10 000 nodes and a total load of 1 000 000.

## 4.2. Metrics

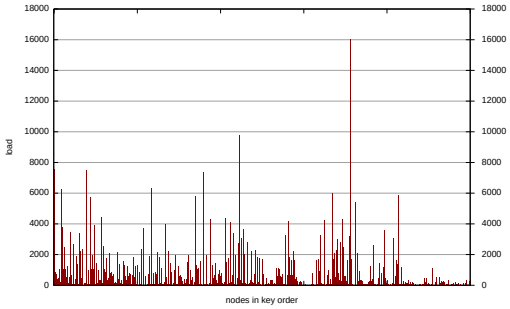
The *performance* of the different algorithms was measured using three different metrics. The first metric is the *standard deviation*, $\sigma_l$, of the nodes' loads which measures the degree of imbalance among them, as defined by the system model. The higher its value, the more imbalanced the nodes are in regard to their loads (ref. Section 3.1). An optimal state ($\sigma_l = 0$) is reached when each node stores the system's average number of items.

The second metric is the amount of *moved load*, i.e. moved items. Assuming the cost of transferring a set of items from one arbitrary node to another is proportional to the number of items, the moved load determines the overall transfer cost of the algorithm. Improving an algorithm's performance could therefore either mean reaching the same balance, i.e. standard deviation, by moving less load or reaching a better (lower) standard deviation while moving the same amount of load.

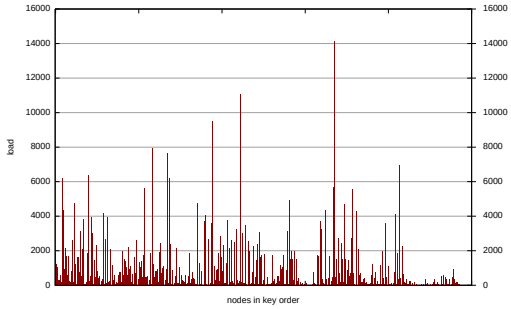
For the sake of comparability the ratio $\delta_{mal}$ between the system's maximal and average load is included as well since this, along with the ratio between the system's maximal and minimal load, has been used by several algorithms introduced in Section 2.4. The latter though has been omitted here because it is not well-defined if the minimal load is 0. Compared to the standard deviation as a metric for *imbalance* though, the ratio $\delta_{mal}$ has certain disadvantages. For example, a small number of highly loaded nodes could create a very high ratio. Continuing to balance the system and further improving its balance might not decrease this ratio though if at least one of those nodes keeps its load. In a system where this single overloaded node would have a severe impact on the overall availability it makes sense to say that its balance has not really improved. However, this should typically not be the case for DHTs storing items and so this metric is disregarded in further discussions.

## 4.3. Simulator program

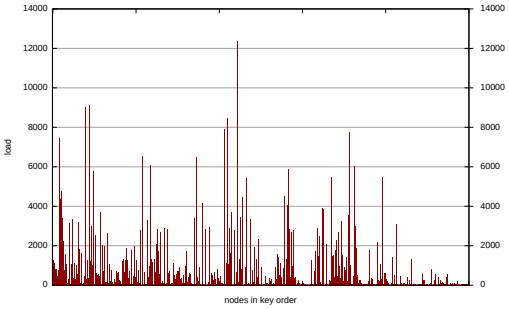
In order to evaluate the algorithms' activities on a DHT with the different scenarios, a simulator program was implemented that emulates such a DHT. It is based on Qt [40] and consists of a common library, a command line client running simulations specified by JavaScript files and a graphical user interface that supports immediate evaluation of a simulation's results with several integrated plots of the metrics described above and their relations. A screenshot of the GUI can be found in Figure 4.2, a sample simulation script in Figure 4.1. Both interfaces offer means of exporting collected simulation results to gnuplot data files and creating appropriate gnuplot scripts that generate such plots.



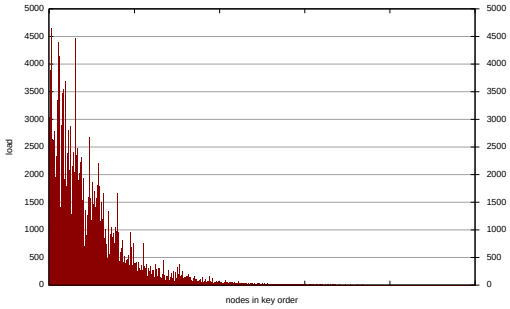

(a) Wikipedia page titles (en)



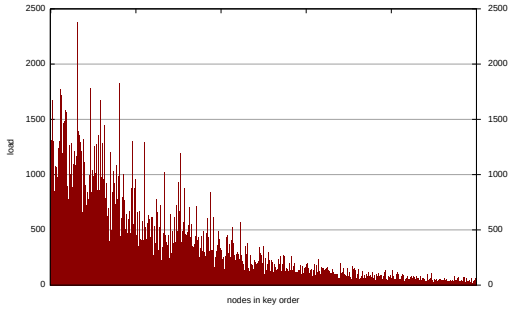

(b) Wikipedia page titles (de)



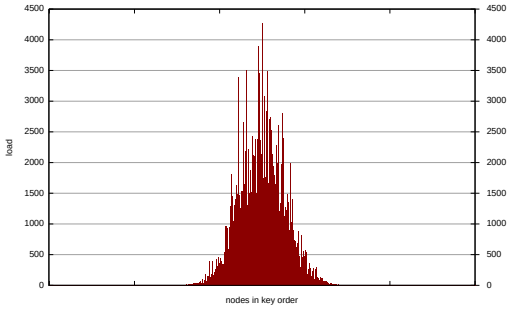

(c) Wikipedia page titles (fr)



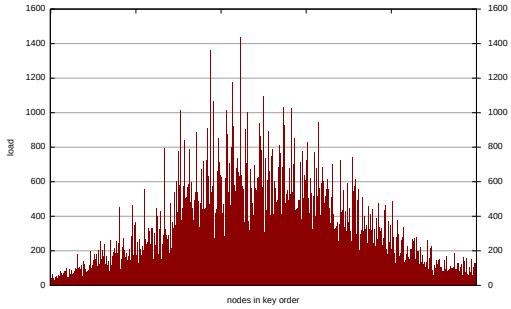

(d) Exponential - $Exp(\lambda = 6 \cdot 10^{-19}))$




(e) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$




(f) Normal - $N(\mu = 2^{61}, \sigma^2 = 1 \cdot 10^{18})$




(g) Normal - $N(\mu = 2^{61}, \sigma^2 = 4 \cdot 10^{18})$

Figure 4.1.: *Simulation scenarios based on alphabetical, exponential and normal item distributions showing the load (number of items) of each of the* 10 000 *nodes in key order (*1 000 000 *items in total).*

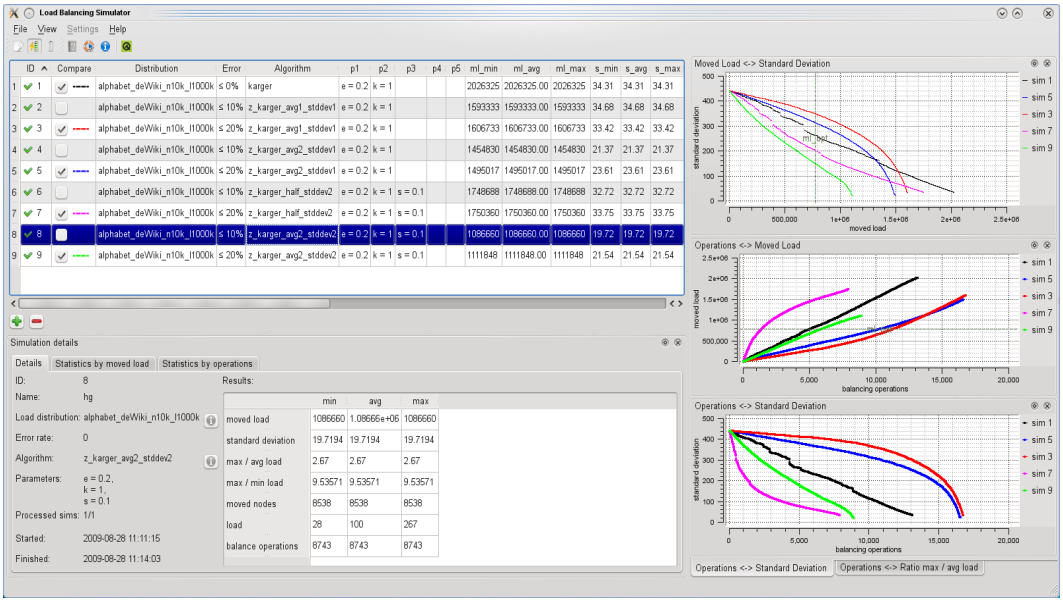



Figure 4.2.: *Snapshot of the Simulator GUI.*

Algorithms and simulation scenarios ("Load Distributions") are implemented as plug-ins and thus new variations can be easily deployed. Load distributions have one parameter: the *error rate* that influences the exactness of the estimates of global information as described above. Algorithms can have several parameters that can all be set in either application interface and are included in the results' data files.

Each simulation has a name and a description and consists of a load distribution and an algorithm. Due to the random nature of most algorithms, a simulation can be specified to run several times (each such test run starts with the same initial parameters). The results of each such simulation will be aggregated to averages over all test runs also storing the minimal and maximal values reached. When an algorithm is invoked by the simulation, it will iterate over all the nodes in the system and will perform its operations for each encountered node. The number of algorithm executions can be specified at the start of a simulation.

Three containers store simulation results: The first container stores the different values of each test run's state at the end of its life-time including the aggregates of the amount of moved load, standard deviation, moved nodes, load among the nodes, number of balance operations and the ratios between the maximal and minimal as well as maximal and average load. Another container stores such values each time the amount of moved load changes and can thus for example be used to plot the moved load against the standard deviation to show which balance state was reached at which costs. Plots showing the standard deviation and the number of moved items for each balance operation, i.e. each operation in which the algorithm signals nodes to perform a jump or slide, can be

```
var ldists = simulation.getLoadDistributions("n10k_l1000k");
var algName = "karger";
var simDesc = "Find best parameters for each scenario.";
var testRuns = 100; // number of test runs of each simulation
var maxTime = 200;// number of algorithm executions in each test run
var collectMI = true; // data by moved items
var collectOp = true; // data by balance operations
var errorRate = 0;
var e_test = new Array(0.01, 0.10, 0.20); // epsilons to test
var k = 1; // number of samples

for (var i = 0; i < ldists.length; ++i) {
 var ldistName = ldists[i];
 var gnuplotPath = "data/" + algName + "--" + ldistName;
 if (simulation.resultsExist(gnuplotPath)) {
  print("skipping " + gnuplotPath);
 } else {
  print("starting " + gnuplotPath);
  simulation.startAutoExportToGnuplot(gnuplotPath, 100);
  for (var i_e = 0; i_e < e_test.length; ++i_e) {
   var algPars = { e: e_test[i_e], k: k };
   var simName = gnuplotPath + "-k" + k + "-e" + algPars.e;
   simulation.addSim(simName, simDesc, ldistName, testRuns,
    errorRate, algName, algPars, collectMI, collectOp);
  }
 simulation.runSims();
 }
}
```

Listing 4.1: *Example simulation script for the Simulator CLI.*

created from the data stored in the third container. It creates such snapshots each time the number of balance operations changes. Since the latter two generate quite much data, they can optionally be turned off.

Much effort has been put into parallelising the core components of the Simulator. As such it uses multiple threads to run the simulations, process the results and export them. At first, each simulation creates two worker threads for the latter two containers. Each test run, which is executed in a separate thread, has a local cache of such snapshots for itself which is piped to a job-queue of the appropriate worker at the end of its life-time. This worker combines the different snapshots one after another. The integration of the snapshot of a simulation's final state into the other container is however done by itself. The number of threads used for concurrent test runs can be limited by providing the command line parameter "`-j <number>`" to the program. By default, the number of

available CPU cores is used. If the simulation scripts enable results to be automatically exported, there will be one more thread which performs those exports similarly to the worker threads mentioned above.

The simulator is available under the GPL version 3 or later [20] and can found on the enclosed DVD as binary packages for Windows and Linux. Source code, license information and documentation is included as well. The latter could be generated at any time by running the Doxygen tool [42] with the supplied `Doxyfile`.

## 4.4. Simulation results

The following sections will evaluate all simulations that have been run in order to analyse the algorithms themselves and under different aspects of the simulations. As only summarised information may be provided here, all simulations' detailed results are available on the enclosed DVD and can also be re-generated using the provided scripts or the simulator GUI. At first, every introduced variant will be analysed on the `karger` algorithm. This evaluation will go into great detail and try to cover most aspects of the simulation in order to assess the robustness of the algorithms as well. Afterwards the same variations will be applied to the `mercury` algorithm that will show whether the same effects can be observed with another algorithm, too. The basic algorithms themselves will always serve as a reference for the algorithm variations. Also note that this should not be a comparison between `karger` and `mercury` and only the effects of the variants compared to their basic algorithm are evaluated.

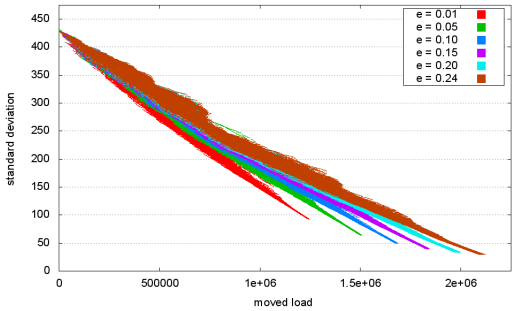
### 4.4.1. Karger item balancing

#### Without added global information

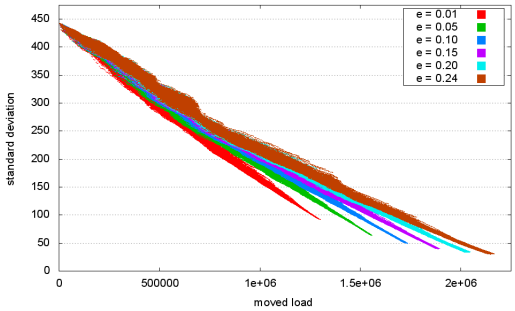
Figure 4.3 shows the standard deviation that the original `karger` algorithm reached after each load movement in each test scenario. It presents the algorithm's performance during its execution and shows which balance can be reached at which cost. Plotted are the collected data points of all simulations with different values of the $\epsilon$ parameter, 100 test runs, 200 algorithm executions and one sampled node. Since Karger and Ruhl state that $\epsilon$ should be greater than 0 and less than $1/4$, the following values have been chosen: 0.01, 0.05, 0.10, 0.15, 0.20 and 0.24. The number of algorithm executions was originally set to 100 which turned out to be too low for some of the karger variants that in turn exhibited results varying too much from the average of all test runs. Those that still show these variations will be discussed in their respective sections below. Also the effect of setting this number even higher will be evaluated for all algorithm variations.

As can be seen from figures 4.3 (a)-(c), for each single value of $\epsilon$, the alphabetical scenarios show similar behaviour. Evaluating an algorithm's variation will therefore at first concentrate on the scenario created from the English Wikipedia's page titles. Similarly the two exponential distributions will be assessed in favour of the normal distributions which, when compared, show only slightly different results. The variant that performs best on these three scenarios ("Wikipedia (en)", "$Exp(\lambda = 6 \cdot 10^{-19})$" and "$Exp(\lambda = 2 \cdot 10^{-19})$") will finally be simulated on all of them in order to make sure it does not show any different behaviour on the other three scenarios.

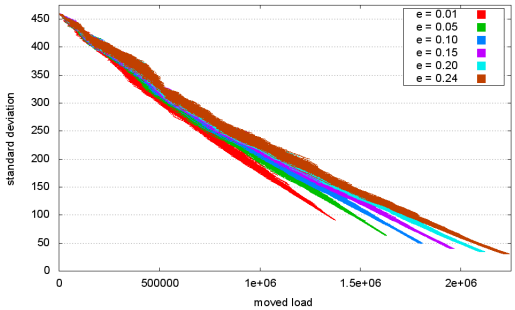
The plots of Figure 4.3 also show that no matter which scenario, smaller $\epsilon$ values



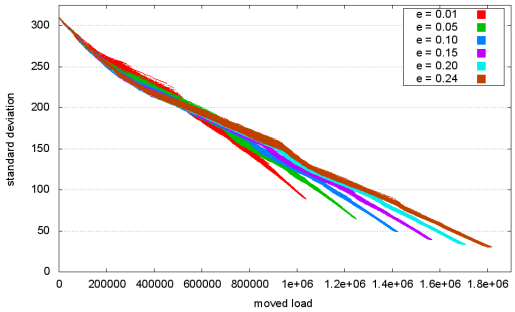

(a) Wikipedia page titles (en)



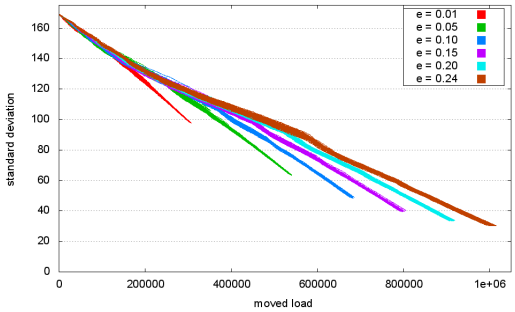

(b) Wikipedia page titles (de)



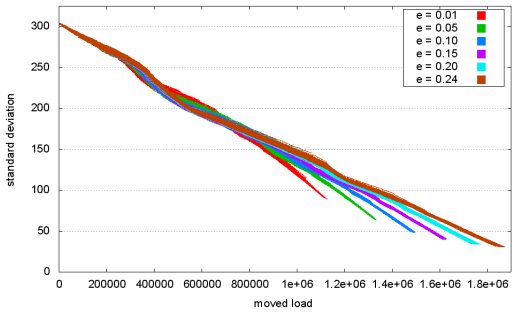

(c) Wikipedia page titles (fr)



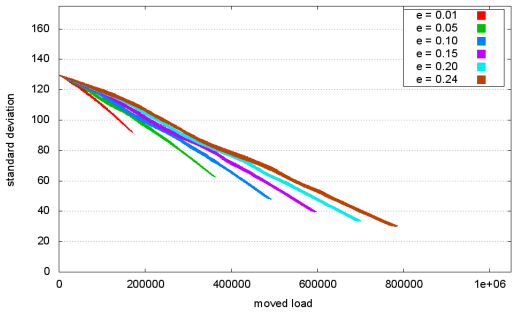

(d) Exponential - $Exp(\lambda = 6 \cdot 10^{-19}))$




(e) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$




(f) Normal - $N(\mu = 2^{61}, \sigma^2 = 1 \cdot 10^{18})$




(g) Normal - $N(\mu = 2^{61}, \sigma^2 = 4 \cdot 10^{18})$

Figure 4.3.: *Balance results for* `karger` *with different* $\epsilon$ *and for each of the scenarios.*

will always reach a given standard deviation by moving fewer load. To understand that effect, recall that `karger` only balances two nodes with each other if their load differs by at least a factor $\epsilon$. Thus small values will make the algorithm only balance nodes with big load differences and since balancing the most loaded nodes affects the imbalance the most, this will lead to a better imbalance with the same number of moved items compared to greater $\epsilon$. However, higher values will reach a better imbalance at the end, which can be seen, too. For the following simulations, a fixed $\epsilon$ of 0.24 is used which resulted in the best imbalance at the end.

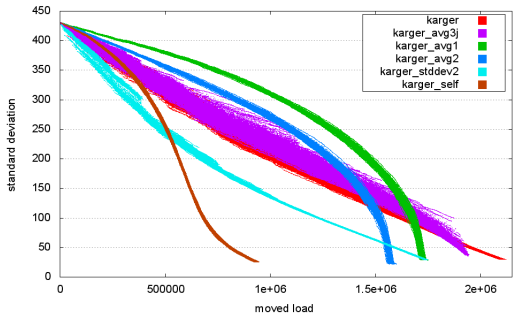
### With added global information

In order to assess the improvements of a `karger` variant, it has been simulated with the "Wikipedia (en)", "$Exp(\lambda = 2 \cdot 10^{-19})$" and "$Exp(\lambda = 6 \cdot 10^{-19})$" scenarios using $\epsilon = 0.24$ as mentioned above. For those simulations a 25% inaccurateness of global information was set in order to resemble the estimate of gossiping. This percentage is supported by experiments conducted by Jelasity et al. [28] evaluating the quality of a gossip algorithm that estimates the system size in a dynamic system. It is thus also a reasonable value for the static simulations presented here. Further discussions about the influence of this *error rate* on the simulation results will however be held at the end of this section.

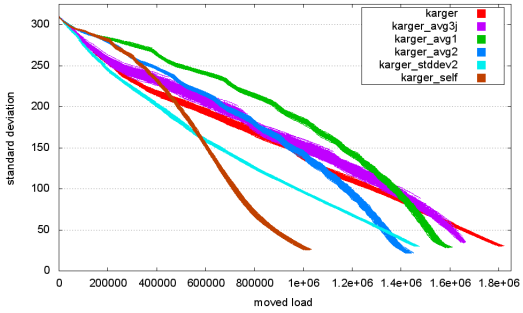
It follows a detailed performance analysis of the different `karger` variants previously introduced. The evaluations will concentrate on showing the imbalance that can be reached by moving a certain amount of load, i.e. items, and the imbalance as well as the number of moved items at the end of each simulation.

The results of the algorithm variants pictured in Figure 4.4 show that **`avg3j`** needs to move more items to reach the same imbalance most of the time during its execution[1]. Only at the end it comes around and falls below the data points of the original `karger` without however achieving the same imbalance. The algorithm is probably refusing some jumps that would have been useful nonetheless, i.e. the potential improvements by the jump would be greater than the prevented light node getting heavy. A possible reason for this might be the error of the estimate being too high but simulations with the exact value of the average load show no better results. In fact, simulations with an error of 25% produce even lower imbalances. Also as Table 4.1 shows, these jumps are traded for slide operations: `karger_avg3j` performs up to 12% fewer jumps than `karger` and increases the number of slides by up to 26%, depending on the scenario. An additional effect that can especially be seen in Figure 4.4a is that the variance of the data points is quite high among different test runs. This also affects the final imbalance at the end

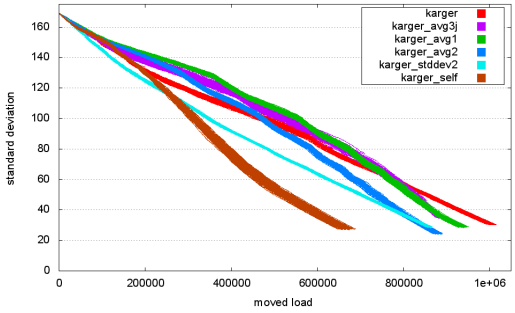
[1] time and number of moved items correlate




(a) Wikipedia page titles (en)




(b) Exponential - $Exp(\lambda = 6 \cdot 10^{-19})$)




(c) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$

Figure 4.4.: *Balance results for the* **karger** *variants with one variation, error rate* 25%.

of each test run that varies by up to 35% (ref. Table 4.2 on page 66). Since the final $\delta_{mal}$ with `karger_avg3j` is also higher than with `karger`, heavier nodes in the system still exist. It seems that those nodes have not been balanced because they have been matched with nodes for which a jump was not possible. In that case `avg3j` suggests no alternative operation so the heavy node's load is left unchanged and more attempts - and thus time - are needed in order to either find a node that is able to jump or to balance with one of its neighbours. It turns out that by further increasing the number of algorithm executions, the variance of the final imbalance of `karger_avg3j` can be greatly reduced (see the respective section below).

The **`avg1`** and **`avg2`** variants perform better, although a significant time of the simulation they expose worse results than the original algorithm, which is understandable because the item movements they carry out do not have such a great impact on the balance as the original ones. However towards the simulation's end, they significantly increase their performance in both imbalance and number of moved items and outperform `karger`. This is because towards the end fewer and fewer nodes are responsible for the bad imbalance (most of the previously heavy nodes have already been balanced) and thus balance operations on the few heavy nodes quickly improve the standard deviation without many item movements. Additionally these two variants achieve a better overall

| **Algorithm** | **Wikipedia (en)** | | $\boldsymbol{Exp(\lambda = 2 \cdot 10^{-19})}$ | | $\boldsymbol{Exp(\lambda = 6 \cdot 10^{-19})}$ | |
|---|---|---|---|---|---|---|
| *name* | *slides* | *jumps* | *slides* | *jumps* | *slides* | *jumps* |
| karger ($err = 0\%$) | 1101.49 | 12709.44 | 598.17 | 6435.19 | 444.10 | 12681.83 |
| ..._avg1 | 1132.43 | 16517.17 | 605.74 | 7600.33 | 529.43 | 16079.09 |
| ..._avg2 | 1625.72 | 15140.70 | 767.75 | 7835.22 | 1178.19 | 14316.18 |
| ..._avg3j | 1158.86 | 12018.87 | 752.07 | 5661.78 | 484.59 | 12199.47 |
| ..._stddev2 | 432.06 | 7451.13 | 334.98 | 4936.37 | 124.93 | 7548.05 |
| ..._self | 577.39 | 9384.95 | 219.64 | 5756.06 | 133.95 | 11304.61 |
| ..._avg1_stddev2 | 403.59 | 8317.22 | 312.57 | 5414.44 | 222.31 | 8494.17 |
| ..._avg2_stddev2 | 229.39 | 8532.15 | 255.44 | 5592.58 | 96.49 | 8861.53 |
| ..._avg3j_avg1 | 1245.92 | 13521.21 | 792.10 | 6040.37 | 681.64 | 13694.82 |
| ..._avg3j_avg2 | 2589.08 | 10609.35 | 1226.50 | 5539.62 | 2106.98 | 10522.54 |
| ..._avg3j_stddev2 | 528.08 | 6958.42 | 472.35 | 4192.76 | 184.06 | 7064.11 |
| ..._avg3j_avg1_stddev2 | 396.56 | 6633.99 | 412.44 | 4273.73 | 297.52 | 6986.91 |
| ..._avg3j_avg2_stddev2 | 396.56 | 6633.99 | 412.44 | 4273.73 | 297.52 | 6986.91 |
| ..._self_avg2_stddev2 | 122.36 | 7635.50 | 128.22 | 5043.02 | 42.91 | 7617.31 |

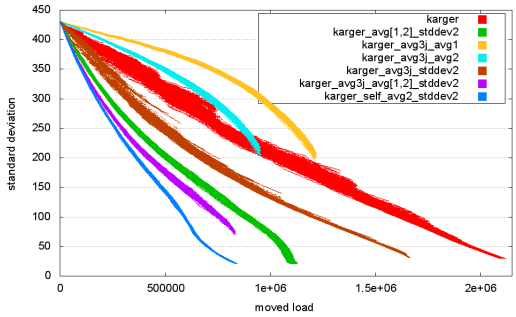
Table 4.1.: *Average number of slide and jump operations of the different* **`karger`** *variants (error rate* $= 25\%$ *unless otherwise stated,* $\epsilon = 0.24$*,* $s = 1.5$ *where appropriate).*

balance at the end of the simulation which is owed to the fact that balancing two nodes in the original `karger` could result in both of them being in the bounds of $\epsilon$ and thus not being considered for balancing anymore. Here on the other side the receiving node's load would at least be nearer to the average load which results in a better imbalance. In `avg2` this node would be even better balanced than in `avg1` so the standard deviation is expected to be lower at the end with probably fewer item movements. This effect has been confirmed by the simulations as the plots and the results at the end of the simulations in Table 4.2 show.

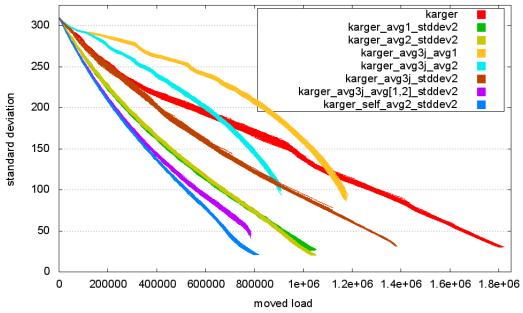
Algorithms using the **`stddev2`** variation need a further parameter, $s$, that influences the decision on whether to perform a balance operation or not. Initial simulations using `karger_stddev2` have shown that $s = 1.5$ is a good value for all scenarios which will thus be used for every variant of Karger and Ruhl's algorithm that incorporates `stddev2`. Figure 4.4 clearly supports that limiting balance operations to those that are really worth it is a good choice. In all three scenarios the resulting imbalance is comparable to the one of the `avg1` variant and better than the result of the original algorithm but in contrast to these two, throughout the whole simulation any given standard deviation is reached by moving far less items.

The **self-tuning** algorithm starts off similar to the original `karger` in the first and third scenario but then quickly gets to the same imbalance of the other algorithms by moving far less items. In the second scenario it first operates similarly to `avg2` but then outperforms it as well. Surprisingly, in the first two scenarios most of the time `karger_self` reaches the same imbalance by even moving far less items than `karger`

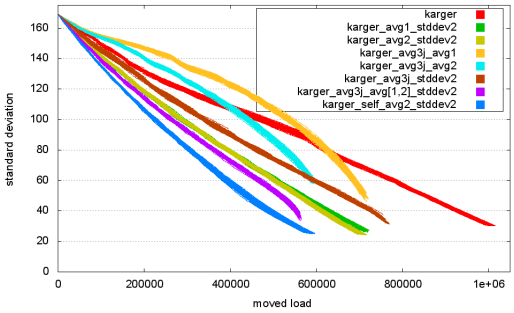
with $\epsilon = 0.01$, e.g. in the alphabetically distributed scenario an imbalance of 100 is reached by moving nearly half the number of items (680 000 versus 1 200 000). The $\epsilon$ set by the self-tuning variant is however never below 0.01. The only possible reason for this would be the (slightly) fluctuating value of $\epsilon$ due to the error rate. This might allow some balance operations which quickly propagate items from heavily loaded nodes to lightly loaded ones and are otherwise not performed. It can not be an effect due to the randomness of the algorithm since all 100 simulations exhibit similar behaviour which can be seen by the thinness of the algorithm's scatter plot, especially in the first two scenarios. The resulting imbalance at the end of the simulations is also around $10 - 15\%$ better than the original algorithm. Overall, it can be said that the self-tuning algorithm did perform even better than anticipated and is the best among the algorithm variants using a single variation.




(a) Wikipedia page titles (en)




(b) Exponential - $Exp(\lambda = 6 \cdot 10^{-19})$)




(c) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$

Figure 4.5.: *Balance results of the combined* `karger` *variants, error rate* 25%. *Indistinguishable curves have been merged, i.e.* `avg[1,2]` *is short hand for "*`avg1` *or* `avg2`*".*

Further simulations, shown in Figure 4.5, have been run in order to analyse the effect of the combination of several of the above evaluated variants incorporated into the `karger` algorithm. The plots show that although the **`avg3j_avg1`** and **`avg3j_avg2`** variants resemble the ideal item movement the most, they don't seem to perform well in any of the three simulations and especially in the alphabetically distributed scenario. This

is probably due to the restricted jump capabilities of the avg3j component as argued above. However, if this variant is combined to **avg3j_stddev2**, it performs similar to stddev2 alone and does not seem to be affected by the limited jumps as much as the original algorithm. Table 4.1 shows an increased number of slide operations of up to 47% and jumps being reduced by less than 15% when adding avg3j to stddev2. The restrictions added by stddev2 however seem to compensate for the restricted jumps and only allow such operations that are useful. Since there is no real improvement to this variant alone though, avg3j_stddev2 could be dropped.

Figure 4.5 also shows that in general variants using the **stddev2** component perform quite well. In particular combinations with **avg1** and **avg2** seem to profit from that. Both exhibit very similar behaviour and can sometimes be indistinguishable in the plots which is why their data points have been merged to a single scatter plot. In those algorithms, restricting balance operations to the ones that significantly reduce the imbalance greatly improves their performance so that any given imbalance is achieved by moving less items than the original algorithm and less items than those variants alone. The omitted operations have thus prevented nodes from taking part in future balance decisions with greater impact due to the restrictions of $\epsilon$ (recall that stddev2 only refuses balance operations and does not influence them in any other way). Additionally, the good imbalance reached at the end of the simulations is maintained (ref. Table 4.2). The latter could have been expected since all participating variants alone exhibit quite similar results. The small impact of choosing either avg1 or avg2 though was surprising but seems to be owed to the greater influence of stddev2. Table 4.1 shows another interesting effect: adding stddev2 to any other algorithm reduces its number of slide operations more than the number of jumps in terms of percentages. Thus the majority of the slides in algorithms without stddev2 does not significantly change the overall imbalance and hence most balance improvements can be achieved by moving nodes.

Further combining **stddev2** with **avg3j** and one of **avg1** or **avg2** looks quite promising at the start, regarding the number of items moved in order to achieve a certain imbalance. It is also surprising that avg3j further improves the algorithms that way although previous combinations with it have not shown this behaviour. At the end though they do not reach the imbalance that can be reached by avg2_stddev2 which is probably owed to the restrictions avg3j imposes on jumps. Additionally a large variance of the resulting imbalance can be observed as before with avg3j variants.

Substitutional for variants combined with the self-tuning facilities implemented for karger, the **karger_avg2_stddev2** variant which showed the best results among the previously evaluated algorithms has been equipped with a **self-tuning** $\epsilon$. The plots in Figure 4.5 show the superiority of this variant over all others since it reaches any imbalance by moving far less items. Additionally Table 4.2 shows that the resulting

| Scenario | Algorithm | Simulation results (avg) | | |
|---|---|---|---|---|
| *name* | *name* | *moved load* | *stddev* | $\delta_{mal}$ |
| Wikipedia (en) | karger ($err = 0\%$) | 2101882.19 $^{\pm 1.00\%}$ | 30.17 $^{\pm 2.45\%}$ | 2.07 |
| | ..._avg1 | 1734141.30 $^{\pm 0.99\%}$ | 29.07 $^{\pm 2.26\%}$ | 2.13 |
| | ..._avg2 | 1582001.44 $^{\pm 1.17\%}$ | 22.40 $^{\pm 2.98\%}$ | 2.15 |
| | ..._avg3j | 1934212.76 $^{\pm 0.65\%}$ | 39.22 $^{\pm 35.27\%}$ | 10.81 |
| | ..._stddev2 | 1738616.51 $^{\pm 0.64\%}$ | 29.28 $^{\pm 2.16\%}$ | 2.12 |
| | ..._self | 935499.20 $^{\pm 1.33\%}$ | 25.82 $^{\pm 2.59\%}$ | 2.20 |
| | ..._avg1_stddev2 | 1113145.43 $^{\pm 1.61\%}$ | 22.02 $^{\pm 3.58\%}$ | 2.15 |
| | ..._avg2_stddev2 | 1109117.52 $^{\pm 1.56\%}$ | 20.92 $^{\pm 1.91\%}$ | 1.88 |
| | ..._avg3j_avg1 | 1209741.98 $^{\pm 0.86\%}$ | 206.97 $^{\pm 6.45\%}$ | 133.82 |
| | ..._avg3j_avg2 | 945844.46 $^{\pm 1.51\%}$ | 214.48 $^{\pm 5.60\%}$ | 136.38 |
| | ..._avg3j_stddev2 | 1661461.88 $^{\pm 0.36\%}$ | 31.27 $^{\pm 3.05\%}$ | 3.62 |
| | ..._avg3j_avg1_stddev2 | 816453.40 $^{\pm 3.17\%}$ | 81.73 $^{\pm 15.97\%}$ | 75.59 |
| | ..._avg3j_avg2_stddev2 | 816453.40 $^{\pm 3.17\%}$ | 81.73 $^{\pm 15.97\%}$ | 75.59 |
| | ..._self_avg2_stddev2 | 835771.74 $^{\pm 1.15\%}$ | 21.43 $^{\pm 2.08\%}$ | 1.99 |
| $Exp(\lambda = 6 \cdot 10^{-19})$ | karger ($err = 0\%$) | 1807032.79 $^{\pm 0.95\%}$ | 30.70 $^{\pm 2.27\%}$ | 2.02 |
| | ..._avg1 | 1591271.05 $^{\pm 1.17\%}$ | 28.98 $^{\pm 2.82\%}$ | 2.10 |
| | ..._avg2 | 1430089.78 $^{\pm 1.53\%}$ | 21.82 $^{\pm 2.31\%}$ | 2.09 |
| | ..._avg3j | 1654011.51 $^{\pm 0.66\%}$ | 35.96 $^{\pm 18.33\%}$ | 7.40 |
| | ..._stddev2 | 1462922.39 $^{\pm 0.86\%}$ | 30.30 $^{\pm 1.64\%}$ | 2.12 |
| | ..._self | 1018445.93 $^{\pm 1.58\%}$ | 25.98 $^{\pm 3.40\%}$ | 2.09 |
| | ..._avg1_stddev2 | 1036543.22 $^{\pm 1.65\%}$ | 26.82 $^{\pm 3.29\%}$ | 2.18 |
| | ..._avg2_stddev2 | 1041755.81 $^{\pm 1.22\%}$ | 20.72 $^{\pm 1.96\%}$ | 1.93 |
| | ..._avg3j_avg1 | 1171385.61 $^{\pm 1.06\%}$ | 93.58 $^{\pm 8.39\%}$ | 30.55 |
| | ..._avg3j_avg2 | 900437.51 $^{\pm 1.14\%}$ | 103.84 $^{\pm 10.00\%}$ | 32.76 |
| | ..._avg3j_stddev2 | 1380104.63 $^{\pm 0.28\%}$ | 31.78 $^{\pm 2.22\%}$ | 3.33 |
| | ..._avg3j_avg1_stddev2 | 781607.82 $^{\pm 0.66\%}$ | 47.33 $^{\pm 13.49\%}$ | 21.07 |
| | ..._avg3j_avg2_stddev2 | 781607.82 $^{\pm 0.66\%}$ | 47.33 $^{\pm 13.49\%}$ | 21.07 |
| | ..._self_avg2_stddev2 | 804922.24 $^{\pm 1.62\%}$ | 21.06 $^{\pm 2.05\%}$ | 1.90 |
| $Exp(\lambda = 2 \cdot 10^{-19})$ | karger ($err = 0\%$) | 1003902.63 $^{\pm 1.14\%}$ | 30.59 $^{\pm 1.74\%}$ | 2.03 |
| | ..._avg1 | 937143.51 $^{\pm 1.43\%}$ | 28.99 $^{\pm 2.36\%}$ | 2.10 |
| | ..._avg2 | 880638.66 $^{\pm 1.41\%}$ | 24.45 $^{\pm 1.92\%}$ | 1.99 |
| | ..._avg3j | 880305.73 $^{\pm 0.89\%}$ | 35.55 $^{\pm 11.68\%}$ | 5.81 |
| | ..._stddev2 | 860193.57 $^{\pm 1.06\%}$ | 28.76 $^{\pm 1.84\%}$ | 2.06 |
| | ..._self | 667771.88 $^{\pm 3.05\%}$ | 27.53 $^{\pm 2.90\%}$ | 2.09 |
| | ..._avg1_stddev2 | 713955.21 $^{\pm 1.45\%}$ | 26.86 $^{\pm 2.44\%}$ | 2.13 |
| | ..._avg2_stddev2 | 706704.75 $^{\pm 1.55\%}$ | 24.65 $^{\pm 2.05\%}$ | 1.96 |
| | ..._avg3j_avg1 | 711236.50 $^{\pm 1.37\%}$ | 50.87 $^{\pm 9.25\%}$ | 15.09 |
| | ..._avg3j_avg2 | 590921.06 $^{\pm 0.98\%}$ | 60.80 $^{\pm 5.26\%}$ | 16.66 |
| | ..._avg3j_stddev2 | 765143.97 $^{\pm 0.56\%}$ | 32.17 $^{\pm 3.49\%}$ | 3.75 |
| | ..._avg3j_avg1_stddev2 | 560023.34 $^{\pm 0.95\%}$ | 37.13 $^{\pm 10.10\%}$ | 12.05 |
| | ..._avg3j_avg2_stddev2 | 560023.34 $^{\pm 0.95\%}$ | 37.13 $^{\pm 10.10\%}$ | 12.05 |
| | ..._self_avg2_stddev2 | 585624.70 $^{\pm 2.15\%}$ | 25.40 $^{\pm 2.02\%}$ | 1.93 |

Table 4.2.: *Results of the different* `karger` *variants, best variants for each scenario marked in yellow (error rate* $= 25\%$ *unless otherwise stated,* $\epsilon = 0.24$, $s = 1.5$ *where appropriate,* 100 *test runs with* 200 *algorithm executions each).*

imbalance at the simulations' end is only slightly worse than `karger_avg2_stddev2` without self-tuning. It looks like the idea of an $\epsilon$ that varies depending on the system's state is also applicable to the introduced variants and can thus achieve even better results by combining the advantages of all of them.

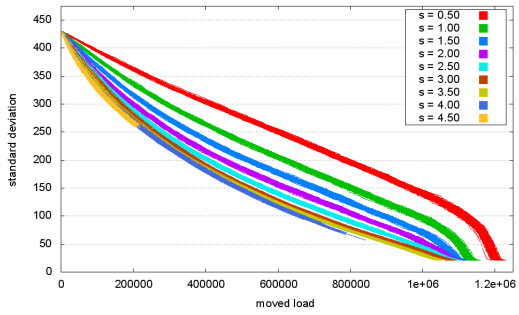
### Best `karger` variants

Thus `karger_avg2_stddev2` with and without self-tuning are considered the best variations of the original `karger` algorithm. Both however can still be influenced by their $s$ parameter that defines the minimal imbalance reduction a balance operation should have in order to be considered for execution. They have thus been simulated with different values for $s$ to find the one with the best performance. The plots in Figure 4.6 show the results of these simulations with the three main scenarios. They show that the `karger_avg2_stddev2` variant without self-tuning relies quite much on the correct value being set. Higher $s$ result in a slightly better imbalance at the simulations' end up to a certain bound. Higher values also reach a given imbalance by moving less items which can be clearly seen. If $s$ is increased too much though, the algorithm will not reach a good imbalance at all which can even happen with small increments as the first two plots show (scenarios "Wikipedia (en)" and $Exp(\lambda = 6 \cdot 10^{-19})$). In both scenarios, $s = 3.5$ still shows a good result but $s = 4.0$ already is quite bad and the result of $s = 4.5$ is even unacceptable. The $Exp(\lambda = 2 \cdot 10^{-19})$ scenario however does not show this behaviour with the simulated values although a slightly worse final imbalance can already be seen with $s = 4.5$. The effect will probably start with higher values.

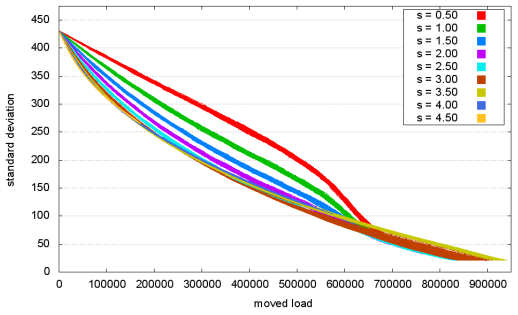
When the `stddev2` variant was developed, its new parameter was integrated in such a way that suggested that its best performance can be reached by a single $s$ for any scenario, i.e. the value of $s$ does not influence (much) the algorithm's performance on any scenario. $s$ thus needs to be set depending on the number of items an algorithm moves in a single balance operation and the resulting change the operation has on the overall balance. If an algorithm only ever moves very few items those moves will probably not have a great impact on the overall imbalance and thus a too high value for $s$ will omit too many balance operations for the algorithm to work. In the case of `karger_avg2_stddev2` however, the number of items the algorithm moves depends on the system's average load, so $s$ needs to be set with care in order not to block too many operations.

The effect of the different performances of the `karger_avg2_stddev2` variant with $s \geq 4.0$ in the three scenarios can however not be explained with a different average load because all scenarios share a common average load of 100. It might instead result from the different imbalances of the scenarios: A single balance operation may not have a big enough affect on the system's imbalance in terms of percentages in scenarios with a greater imbalance. Especially since the number of moved items is limited by the average

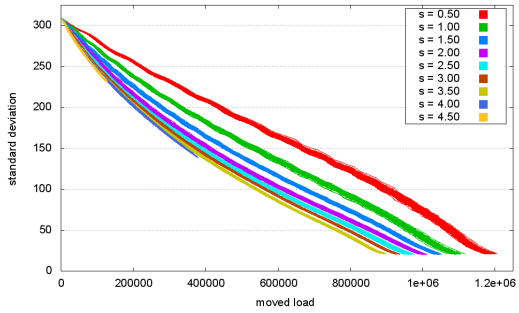
load. This suggests that a weighted bound based on the current imbalance is more useful than statically dropping all operations that do not increase the imbalance by a factor of at least $s/n$ as suggested by `stddev2`. It will probably also be beneficial for this algorithm variant when applied to arbitrary scenarios.



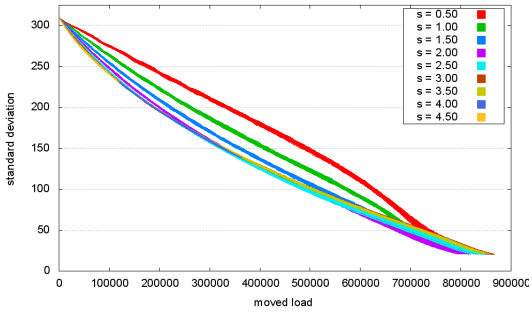

(a) Wikipedia (en), `karger_avg2_stddev2`



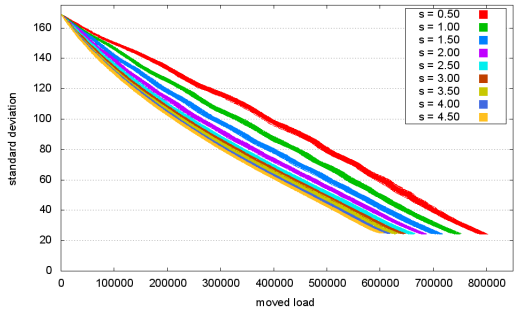

(b) Wikipedia (en), `karger_self_avg2_stddev2`



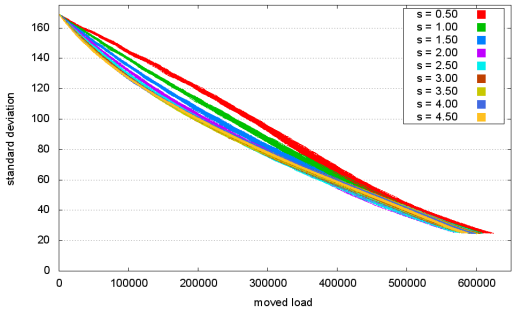

(c) $Exp(\lambda = 6 \cdot 10^{-19})$, `karger_avg2_stddev2`




(d) $Exp(\lambda = 6 \cdot 10^{-19})$, `karger_self_avg2_stddev2`




(e) $Exp(\lambda = 2 \cdot 10^{-19})$, `karger_avg2_stddev2`




(f) $Exp(\lambda = 2 \cdot 10^{-19})$, `karger_self_avg2_stddev2`

Figure 4.6.: *Balance results of the best* **`karger`** *variants with different error rates. (scenario: Wikipedia (en),* $\epsilon = 0.24$ *where appropriate,* 100 *test runs,* 200 *algorithm executions)*

From those simulations alone, the *optimal* $s$ might be chosen as 3.0 but simulations later carried out to evaluate the influence of the error rate revealed that this value was not performing good with low error rates. This is why a safer value of 2.0 was chosen instead (ref. Figure 4.7c on page 73).

Different values of $s$ seem to affect the self-tuning variant `karger_self_avg2_stddev2` in another way than the same algorithm without self-tuning. Results of the self-tuning algorithm shown in figures 4.6 b, d and f show the same behaviour as before with $s \geq 4.0$ in the first two scenarios. Also in the third, it can not be observed. In contrast to the variant without self-tuning though their final imbalances only vary insignificantly and do not necessarily improve with higher $s$.

| **Scenario** | **Algorithm** | **Simulation results (avg)** | | |
|---|---|---|---|---|
| *name* | *name* | *moved load* | *stddev* | $\delta_{mal}$ |
| Wikipedia (en) | karger ($err = 0\%$) | 2101882.19 $^{\pm 1.00\%}$ | 30.17 $^{\pm 2.45\%}$ | 2.07 |
| | karger_avg2_stddev2 | 1090815.91 $^{\pm 2.45\%}$ | 20.97 $^{\pm 1.53\%}$ | 1.87 |
| | karger_self_avg2_stddev2 | 831745.79 $^{\pm 0.92\%}$ | 21.38 $^{\pm 1.93\%}$ | 1.95 |
| Wikipedia (de) | karger ($err = 0\%$) | 2150263.47 $^{\pm 0.94\%}$ | 30.73 $^{\pm 2.72\%}$ | 2.03 |
| | karger_avg2_stddev2 | 1121778.42 $^{\pm 1.65\%}$ | 20.65 $^{\pm 1.99\%}$ | 1.88 |
| | karger_self_avg2_stddev2 | 842240.22 $^{\pm 1.01\%}$ | 20.95 $^{\pm 1.93\%}$ | 1.91 |
| Wikipedia (fr) | karger ($err = 0\%$) | 2223748.31 $^{\pm 0.82\%}$ | 31.17 $^{\pm 1.85\%}$ | 2.06 |
| | karger_avg2_stddev2 | 1166642.63 $^{\pm 1.91\%}$ | 20.44 $^{\pm 2.13\%}$ | 1.91 |
| | karger_self_avg2_stddev2 | 858124.86 $^{\pm 1.36\%}$ | 20.54 $^{\pm 2.06\%}$ | 1.92 |
| $Exp(\lambda = 6 \cdot 10^{-19})$ | karger ($err = 0\%$) | 1807032.79 $^{\pm 0.95\%}$ | 30.70 $^{\pm 2.27\%}$ | 2.02 |
| | karger_avg2_stddev2 | 1000030.18 $^{\pm 1.28\%}$ | 20.70 $^{\pm 1.62\%}$ | 1.89 |
| | karger_self_avg2_stddev2 | 809677.18 $^{\pm 2.46\%}$ | 20.87 $^{\pm 1.93\%}$ | 1.90 |
| $Exp(\lambda = 2 \cdot 10^{-19})$ | karger ($err = 0\%$) | 1003902.63 $^{\pm 1.14\%}$ | 30.59 $^{\pm 1.74\%}$ | 2.03 |
| | karger_avg2_stddev2 | 679464.20 $^{\pm 1.14\%}$ | 24.64 $^{\pm 1.77\%}$ | 1.95 |
| | karger_self_avg2_stddev2 | 583767.38 $^{\pm 2.76\%}$ | 25.36 $^{\pm 1.56\%}$ | 1.93 |
| $N(\mu = 2^{61}, \sigma^2 = 1 \cdot 10^{18})$ | karger ($err = 0\%$) | 1858444.98 $^{\pm 1.09\%}$ | 30.98 $^{\pm 1.79\%}$ | 2.03 |
| | karger_avg2_stddev2 | 998929.11 $^{\pm 1.02\%}$ | 20.03 $^{\pm 1.57\%}$ | 1.86 |
| | karger_self_avg2_stddev2 | 846733.68 $^{\pm 3.10\%}$ | 19.85 $^{\pm 2.14\%}$ | 1.87 |
| $N(\mu = 2^{61}, \sigma^2 = 4 \cdot 10^{18})$ | karger ($err = 0\%$) | 778226.73 $^{\pm 1.25\%}$ | 30.49 $^{\pm 1.89\%}$ | 2.04 |
| | karger_avg2_stddev2 | 553003.90 $^{\pm 1.22\%}$ | 25.86 $^{\pm 2.09\%}$ | 1.94 |
| | karger_self_avg2_stddev2 | 501005.91 $^{\pm 3.41\%}$ | 26.39 $^{\pm 2.02\%}$ | 1.93 |

Table 4.3.: *Results of the best* `karger` *variants for all scenarios (error rate* $= 25\%$ *unless otherwise stated,* $\epsilon = 0.24$*,* $s = 2.0$ *where appropriate,* 100 *test runs with* 200 *algorithm executions each).*

Starting from a certain value of $s$, the number of moved items needed to get the final imbalance is increasing instead of decreasing monotonously as in the variant without self-tuning. This *barrier* at which this change is starting depends on the scenario but all three scenarios show a decreasing number of moved items up to $s = 2.0$ which is thus considered *optimal* for the self-tuning algorithm, too. The insignificant effect of different $s$ is probably due to the algorithm's $\epsilon$ being coupled to (an estimate of) the system's standard deviation and thus probably a higher value for that is being used earlier than with lower $s$. This dampens the success of higher values and thus results in

the performances shown.

Since the *optimal* values for parameters of the best karger variants have been set, they can now be simulated on all scenarios previously introduced. Table 4.3 shows the final results of such simulations using `karger` and the `karger_avg2_stddev2` variant with and without a self-tuning $\epsilon$ parameter. Also given is the maximum deviation of this average value to the minimal and maximal value among all 100 test runs. It shows that in the alphabetical scenarios, the algorithm variants both achieve an imbalance that is around 30% better than the final imbalance using the original algorithm. Additionally, both move far less items: `karger_avg2_stddev2` without self-tuning moves only about 50% of the amount `karger` moves which can be further reduced to 40% if self-tuning is used. Results of the $Exp(\lambda = 6 \cdot 10^{-19})$ and $N(\mu = 2^{61}, \sigma^2 = 1 \cdot 10^{18})$ scenarios show similar results (45% and 55% less item movements respectively with the same improvements of the imbalance). In the other two scenarios, imbalance improvements are only at about 15% with $30 - 35\%$ and $35 - 40\%$ fewer item movements respectively. Variances from those average results are negligible which indicates that the given results can be expected for any simulation despite the algorithms having a random component. The results of the latter two scenarios not being as good as the others can probably be explained by the scenarios already having a better imbalance at the start of the simulations and $\epsilon$ limiting any further improvements.

#### Number of algorithm executions

As mentioned above, some algorithms, especially those with the `avg3j` variant, have been limited by the number of algorithm executions the simulations were set up with. Simulating them with a too low value will result in a large variance of their results at the simulations' final state. This can be observed in the results presented in Table 4.2 on page 66 with most variations of `avg3j`. In order to further analyse the effect of the number of algorithm executions and see whether this is the limiting factor or inherent in the algorithm, further simulations have been set up using 200, 400 and 800 algorithm executions.

Results of those simulations on the "Wikipedia (en)" scenario are shown in Table 4.4 and indicate that the majority of the algorithms are not affected by the increased number of executions. They mostly achieve insignificantly better imbalances by moving slightly more items. This is good for real-world scenarios where the algorithms are not stopped after an arbitrary number of executions but continue to operate.

All `avg3j` variants but `avg3j_stddev2` however show significant improvements when executed more often. `karger_avg3j` without any more changes for example achieves a 15% better imbalance when executed 800 times instead of 200. Additionally the results of the different test runs then vary only by up to 6.60% instead of 35.27 which is a major

improvement. These results are achieved by moving less than 1% more items. The fact that the variance of the number of moved items with 200 executions is very little supports the assumption that this variant is omitting too many of the potential balance operations and is only waiting for a *perfect match* to occur which is left to chance. In order to achieve comparable results, it would thus have to be called at least 4 times as often as `karger`, but since the improvements compared to that are quite small, another variant is a better choice.

| **Algorithm** | Number of algorithm executions | | | | | |
|---|---|---|---|---|---|---|
| | **200** | | **400** | | **800** | |
| *name* | *moved load* | *stddev* | *moved load* | *stddev* | *moved load* | *stddev* |
| karger ($err = 0\%$) | 2101882.19 ±1.00% | 30.17 ±2.45% | 2103140.72 ±0.99% | 30.03 ±2.45% | 2103796.72 ±0.99% | 29.96 ±2.44% |
| ..._avg1 | 1734141.30 ±0.99% | 29.07 ±2.26% | 1735693.54 ±0.95% | 28.87 ±2.21% | 1736382.82 ±0.93% | 28.79 ±2.25% |
| ..._avg2 | 1582001.44 ±1.17% | 22.40 ±2.98% | 1583620.65 ±1.15% | 22.11 ±2.76% | 1584258.24 ±1.16% | 22.00 ±2.92% |
| ..._avg3j | 1934212.76 ±0.65% | 39.22 ±35.27% | 1945354.89 ±0.55% | 35.11 ±23.09% | 1951379.16 ±0.54% | 33.17 ±6.60% |
| ..._stddev2 | 1738616.51 ±0.64% | 29.28 ±2.16% | 1739724.63 ±0.64% | 29.14 ±2.36% | 1740337.26 ±0.65% | 29.06 ±2.35% |
| ..._self | 935499.20 ±1.33% | 25.82 ±2.59% | 938410.78 ±1.28% | 25.37 ±3.44% | 939270.53 ±1.30% | 25.24 ±3.14% |
| ..._avg1_stddev2 | 1113145.43 ±1.61% | 22.02 ±3.58% | 1114870.02 ±1.60% | 21.71 ±3.41% | 1115548.02 ±1.61% | 21.59 ±3.54% |
| ..._avg2_stddev2 | 1109117.52 ±1.56% | 20.92 ±1.91% | 1110666.33 ±1.58% | 20.71 ±1.89% | 1111222.90 ±1.60% | 20.63 ±2.15% |
| ..._avg3j_avg1 | 1209741.98 ±0.86% | 206.97 ±6.45% | 1244770.19 ±0.72% | 186.00 ±6.54% | 1271884.97 ±1.11% | 166.41 ±9.26% |
| ..._avg3j_avg2 | 945844.46 ±1.51% | 214.48 ±5.60% | 980313.28 ±1.33% | 193.18 ±5.60% | 1006870.52 ±1.06% | 173.92 ±8.54% |
| ..._avg3j_stddev2 | 1661461.88 ±0.36% | 31.27 ±3.05% | 1664858.96 ±0.32% | 30.55 ±2.95% | 1666627.45 ±0.26% | 30.21 ±3.13% |
| ..._avg3j_avg1_stddev2 | 816453.40 ±3.17% | 81.73 ±15.97% | 843764.96 ±1.84% | 68.15 ±18.39% | 856083.76 ±1.17% | 57.42 ±19.66% |
| ..._avg3j_avg2_stddev2 | 816453.40 ±3.17% | 81.73 ±15.97% | 843764.96 ±1.84% | 68.15 ±18.39% | 856083.76 ±1.17% | 57.42 ±19.66% |
| ..._self_avg2_stddev2 | 835771.74 ±1.15% | 21.43 ±2.08% | 839216.44 ±0.94% | 20.93 ±1.54% | 839990.51 ±0.90% | 20.83 ±1.61% |

Table 4.4.: *Average results of the* `karger` *variants with different number of algorithm executions, most affected algorithms marked in yellow (Wikipedia (en), error rate* 25% *unless otherwise stated,* $\epsilon = 0.24$*,* $s = 1.5$ *where appropriate,* 100 *test runs).*

The imbalance results of the `avg3j_avg1` and `avg3j_avg2` variants can be improved by more executions, too. But unlike the pristine `avg3j` variant, the variance of these results is even higher in terms of percentages. This leaves room for improvements and it is unclear whether an even higher number of executions would solve this. At least

the number of moved items with 800 executions increases by only $5 - 7\%$ compared to 200. `avg3j_stddev2` behaves similarly to `stddev2` alone as already observed above. This continues with an increased number of executions.

The `avg3j_avg1_stddev2` and `avg3j_avg1_stddev2` variants - that looked quite promising at the start of the simulations (ref. Figure 4.5 on page 64) - continue to improve their final imbalance with an increased number of executions. Simulating it four times as often as originally though only achieves an imbalance of 57.42 at the end. Although this is achieved by moving only about 5% more items, the results vary even more than with 200 executions in terms of percentages. If the improvements can be continued by increasing the number of executions even more, it might get near the `avg2_stddev2` variant but it would then probably need much more tries to find nodes that can be balanced with each other.

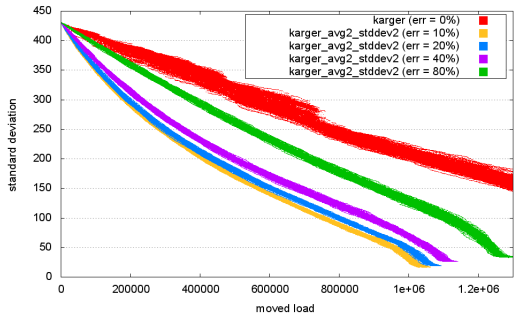
### Error rate

One of the major aspects of all algorithm variations introduced above is that they work with *estimated* global information. Gossiping algorithms are used to retrieve any of such values, e.g. average load, standard deviation, and will approximate them to a certain degree which will be more exact the more often the gossip algorithm is executed. In highly dynamic systems however they need some time to incorporate any changes and thus provide worse approximations for a while. The quality of this information in the given simulations can be influenced by setting an error rate (previously at 25%). Further simulations with the best `karger` variants identified above should clarify how dependent their performance is on the accurateness of the global information.

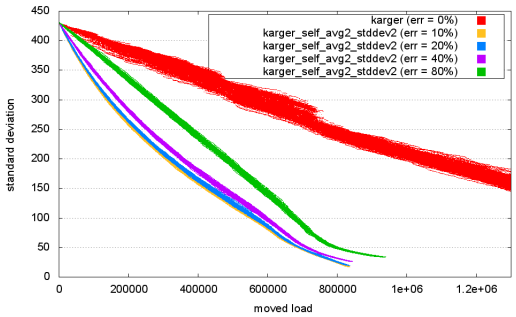
As such, simulations with error rates from $10 - 80\%$ have been run whose results are shown in Figure 4.7. It also includes results of simulations with `karger_avg2_stddev2` using $s = 3$ that has been mentioned above as a candidate for the *best* value of $s$. As the result table shows, this parameter probably restricts too many balance operations (moving up to $l_{avg} \pm 10\%$ items in a single balance operation is more likely to hit this barrier than $l_{avg} \pm 20\%$).

The plots in Figure 4.7a,b show the results of the `karger_avg2_stddev2` variant without and with self-tuning and also plot the ordinary `karger` algorithm for comparison. Recall that the latter is independent from the error rate since it does not use any global information. Furthermore, the variant without self-tuning seems to be influenced by the error rate a bit more than the other. Its performance however does degrade gracefully so that even with an error of up to 80% an imbalance that is comparable to `karger` can be reached by moving about 25% more items than the same algorithm with a 10% error and still about 40% less than `karger`. According to this the algorithm seems pretty robust to erroneous global information. This is probably due to the fact that the

bound that $\epsilon$ imposes has a greater affect than an error rate that only determines how many items are moved.




(a) `karger_avg2_stddev2` ($e = 0.24,\ s = 2$)




(b) `karger_self_avg2_stddev2` ($s = 2$)

| **Algorithm** | | **Error** | **Simulation results (avg)** | | |
|---|---|---|---|---|---|
| *name* | *parameters* | | *moved load* | *stddev* | $\delta_{mal}$ |
| karger | $e = 0.24$ | $\leq$ 0% | 2101882.19 $\pm$ 1.00% | 30.17 $\pm$ 2.45% | 2.07 |
| ..._avg2_stddev2 | $e = 0.24,\ s = 2$ | $\leq$ 10% | 1041627.75 $\pm$ 2.08% | 16.79 $\pm$ 3.02% | 2.05 |
| | | $\leq$ 20% | 1078691.18 $\pm$ 2.28% | 19.19 $\pm$ 2.00% | 1.93 |
| | | $\leq$ 40% | 1123017.37 $\pm$ 2.10% | 26.36 $\pm$ 1.37% | 1.88 |
| | | $\leq$ 80% | 1291849.74 $\pm$ 1.91% | 33.38 $\pm$ 1.46% | 1.97 |
| | $e = 0.24,\ s = 3$ | $\leq$ 10% | 416431.18 $\pm$30.26% | 168.79 $\pm$26.22% | 9.98 |
| | | $\leq$ 20% | 1058711.34 $\pm$ 1.39% | 19.27 $\pm$ 2.09% | 1.97 |
| | | $\leq$ 40% | 1102975.35 $\pm$ 1.76% | 26.57 $\pm$ 1.45% | 1.87 |
| | | $\leq$ 80% | 1268950.19 $\pm$ 2.21% | 33.56 $\pm$ 1.67% | 1.97 |
| ..._self_avg2_stddev2 | $s = 2$ | $\leq$ 10% | 830177.11 $\pm$ 0.80% | 17.79 $\pm$ 3.77% | 2.11 |
| | | $\leq$ 20% | 830203.73 $\pm$ 0.80% | 19.85 $\pm$ 2.47% | 2.01 |
| | | $\leq$ 40% | 836896.21 $\pm$ 0.92% | 26.79 $\pm$ 1.71% | 1.86 |
| | | $\leq$ 80% | 927569.14 $\pm$ 1.37% | 34.20 $\pm$ 1.57% | 1.97 |

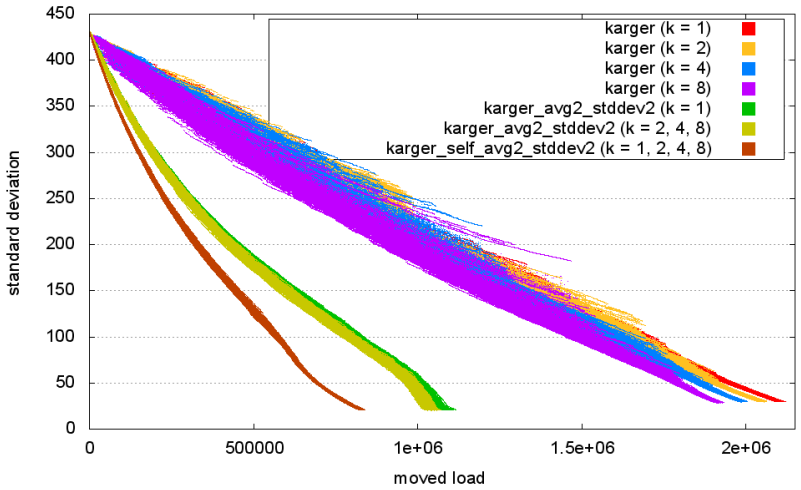
(c) Results at the end of the simulations

Figure 4.7.: *Results of the best* **`karger`** *variants with different error rates. (Wikipedia (en), 100 test runs, 200 algorithm executions each, plots cut off at* $1.3m$ *moved items)*

Even more robust is the `karger_self_avg2_stddev2` variant with self-tuning. Error rates of 10 or 20% nearly differ at all from each other and increasing the error to 40% not even moves 1% more items by still achieving a fair imbalance. A bigger difference can be seen with 80% which offers almost the same imbalance of `karger_avg2_stddev2` at 80% but still moving less than $1m$ items. The better result is probably owed to $\epsilon$ being adapted to the system's state every time at each node independently from any previous state. After all with an average load of 100 and several nodes with loads over 1 000 (ref. Figure 4.1 on page 55), an 80% difference does not matter that much. This might be different for simulations with higher loads though.

### Number of sampled nodes

Another aspect of the algorithms is their ability to sample multiple random nodes instead of just one. In that case a node can decide with which of the sampled nodes it balances. This should at least allow the algorithm to get to lower imbalances earlier than with only one sample but introduces additional traffic on the network. The algorithms of the following simulations were set up to sample $k \in \{1, 2, 4, 8\}$ different nodes uniformly at random, dry-run the balance operations with each of them and choose the best among them as described in Section 3.2. The results of such simulations using `karger` and the two best variants from above are shown in Figure 4.8.




(a) Results during the simulations (indistinguishable curves have been merged)

| **Algorithm** | | **Simulation results (avg)** | | |
|---|---|---|---|---|
| *name* | *parameters* | *moved load* | *stddev* | $\delta_{mal}$ |
| karger | $e = 0.24, k = 1$ | 2101882.19 $^{\pm 1.00\%}$ | 30.17 $^{\pm 2.45\%}$ | 2.07 |
| | $e = 0.24, k = 2$ | 2050068.45 $^{\pm 0.68\%}$ | 29.98 $^{\pm 2.37\%}$ | 2.01 |
| | $e = 0.24, k = 4$ | 1988928.43 $^{\pm 0.81\%}$ | 30.01 $^{\pm 2.12\%}$ | 1.99 |
| | $e = 0.24, k = 8$ | 1917664.25 $^{\pm 0.87\%}$ | 28.83 $^{\pm 2.21\%}$ | 1.95 |
| ..._avg2_stddev2 | $e = 0.24, s = 2, k = 1$ | 1090815.91 $^{\pm 2.45\%}$ | 20.97 $^{\pm 1.53\%}$ | 1.87 |
| | $e = 0.24, s = 2, k = 2$ | 1047354.14 $^{\pm 2.09\%}$ | 20.75 $^{\pm 1.74\%}$ | 1.75 |
| | $e = 0.24, s = 2, k = 4$ | 1040849.12 $^{\pm 2.21\%}$ | 20.67 $^{\pm 2.29\%}$ | 1.69 |
| | $e = 0.24, s = 2, k = 8$ | 1051528.20 $^{\pm 1.91\%}$ | 20.65 $^{\pm 1.72\%}$ | 1.67 |
| ..._self_avg2_stddev2 | $s = 2, k = 1$ | 831745.79 $^{\pm 0.92\%}$ | 21.38 $^{\pm 1.93\%}$ | 1.95 |
| | $s = 2, k = 2$ | 830804.58 $^{\pm 1.12\%}$ | 21.03 $^{\pm 1.90\%}$ | 1.76 |
| | $s = 2, k = 4$ | 829238.02 $^{\pm 1.18\%}$ | 20.96 $^{\pm 1.90\%}$ | 1.69 |
| | $s = 2, k = 8$ | 827633.51 $^{\pm 0.93\%}$ | 20.95 $^{\pm 2.18\%}$ | 1.67 |

(b) Results at the end of the simulations

Figure 4.8.: *Results of the best* `karger` *variants with different numbers of sampled nodes (k). (Wikipedia (en),* 100 *test runs,* 200 *algorithm executions each, error rate* 25%*)*

The plots clearly show the expected effect with the `karger` algorithm: The more nodes are sampled the more the scatter plot of the algorithm turns bottom left, i.e. the same imbalance is reached by moving less items. Also the final results are slightly better with higher $k$ but as seen with $k = 4$ this is not always the case in contrast to the number of moved items that decreases monotonously.

This effect however can not be observed with the `avg2_stddev2` variants with or without self-tuning. In the latter, a difference can only be found between either sampling one node or more (algorithms with $k \in \{2, 4, 8\}$ perform nearly the same). The insignificant differences of the standard deviation and the number of moved items at the end of the simulations may be owed to the $\epsilon$ preventing any further improvements. In contrast to the standard deviation though, the ratio $\delta_{mal}$ of the maximum load to the average load always decreases with higher values of $k$. So higher values do at least show some effect: the maximum load in the system decreases (recall that the average load is constant throughout the whole simulation).

Finally, the self-tuning algorithm does not show any significant differences between the simulations with a different number of sampled nodes, except for a decreasing $\delta_{mal}$. This is probably due to the strong coupling of $\epsilon$ to the current system's state and the fact that this variant without self-tuning was already not influenced much by different numbers of sampled nodes.

### Scalability

Up until now all simulations have always been carried out with 10 000 nodes and a total load of 1 000 000 items and parameters have been set according to such scenarios. In this section the algorithms will show whether they also perform as expected in scenarios with more nodes or greater loads.

As such `karger` and its best two variants have been run on the "Wikipedia (en)" scenario with 10 000 nodes and different total loads as shown in Table 4.5. This will, substitutionally for all scenarios, clarify whether the algorithms work with different average loads as well. At first it can be seen that the ordinary `karger` algorithm nearly doubles both the number of moved items and the standard deviation at the end of the scenarios when the total load is doubled. This effect is inherent in the increased total load and can also be observed with the two karger variants. The variance among the two values for all 100 test runs also stays within reasonable bounds and does not increase with the increased load. These results indicate good scalability in terms of system load for all three algorithms.

Furthermore scenarios with a different system size, i.e. number of nodes, can be set up to evaluate whether the algorithm's performance depends on the number of nodes in the system. The `stddev2` variant for example has already been developed with this in mind:

| Algorithm | Total load | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 500 000 | | 1 000 000 | | 2 000 000 | | 4 000 000 | |
| | *moved l.* | *stddev* | *moved l.* | *stddev* | *moved l.* | *stddev* | *moved l.* | *stddev* |
| karger | 1048218 ±0.82% | 15.14 ±2.15% | 2101882 ±1.00% | 30.17 ±2.45% | 4205981 ±1.01% | 60.49 ±2.84% | 8414797 ±0.83% | 121.09 ±2.58% |
| ..._av2_st2 | 545453 ±1.89% | 10.46 ±1.97% | 1090815 ±2.45% | 20.97 ±1.53% | 2183785 ±1.83% | 41.93 ±1.58% | 4362201 ±1.86% | 83.90 ±1.52% |
| ..._se_av2_st2 | 414751 ±0.75% | 10.66 ±2.37% | 831745 ±0.92% | 21.38 ±1.93% | 1666331 ±0.83% | 42.73 ±2.75% | 3336612 ±1.14% | 85.71 ±1.87% |

Table 4.5.: *Average results of the* `karger` *variants with different total loads (Wikipedia (en), error rate* 0% *with* `karger`*, otherwise* 25%*,* $\epsilon = 0.24$*,* $s = 2.0$ *where appropriate,* 100 *test runs,* 200 *algorithm executions each,* 10 000 *nodes, moved load rounded down to the nearest integral, abbreviated algorithm names).*

it considers the fact that if more nodes share the same amount of load, a single balance operation will probably affect the overall imbalance less than with fewer nodes. This is why the system size has been integrated there as well. The same three algorithms as above also had to complete the "Wikipedia (en)" scenario with a fixed amount of 1 000 000 items but different system sizes.

The results of these simulations, presented in Table 4.6, show that `karger` scales linearly with an increasing number of nodes. Inversely to the simulations above, the standard deviation decreases if more nodes share the same total load. It halves with `karger` compared to scenarios with half as many nodes while the number of moved items constantly increases by about 10%. The latter should ideally not change much but such a small increase when doubling the system size is acceptable.

| Algorithm | Number of nodes | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 5 000 | | 10 000 | | 20 000 | | 40 000 | |
| | *moved l.* | *stddev* | *moved l.* | *stddev* | *moved l.* | *stddev* | *moved l.* | *stddev* |
| karger | 1926994 ±1.14% | 60.64 ±4.15% | 2101882 ±1.00% | 30.17 ±2.45% | 2306485 ± 0.58% | 15.32 ± 1.41% | 2506472 ±0.36% | 7.57 ± 0.83% |
| ..._av2_st2 | 1063600 ±2.48% | 42.53 ±2.66% | 1090815 ±2.45% | 20.97 ±1.53% | 1122572 ± 1.57% | 10.58 ± 1.58% | 902782 ±7.76% | 30.24 ±21.17% |
| ..._se_av2_st2 | 823234 ±1.79% | 43.38 ±3.57% | 831745 ±0.92% | 21.38 ±1.93% | 756367 ±12.36% | 21.73 ±69.05% | 613656 ±6.75% | 35.88 ±18.32% |

Table 4.6.: *Average results of the* `karger` *variants with different system sizes (Wikipedia (en), error rate* 0% *with* `karger`*, otherwise* 25%*,* $\epsilon = 0.24$*,* $s = 2.0$ *where appropriate,* 100 *test runs,* 200 *algorithm executions each, total load* 1 000 000*, moved load rounded down to the nearest integral, abbreviated algorithm names).*

The `avg2_stddev2` variant without self-tuning however only halves the reached standard deviation up to a system size of 20 000 nodes. With 40 000 nodes it gets much

worse and also shows a great variance among the 100 test runs. This indicates that the number of algorithm executions is not high enough for the given scenario which might especially hit `stddev2` variants since they block balance operations and depend on a possibility to find alternatives. The same restriction applies to the self-tuning algorithm which already performs worse than `karger` with 20 000 nodes and continues to do so with more nodes. Further simulations with an increased number of algorithm executions to 400 support the previous assumption of this being the limiting factor. In those, `karger_self_avg2_stddev2` achieves an imbalance of about 5.22 with 40 000 nodes by moving about 882453 items. These values are the ones that could have been expected by this algorithm and since in real-world scenarios the algorithm would operate indefinitely and would not stop after an arbitrary number of executions, this variant is still a good choice that does scale linearly with the system size as well.

#### Summary of Results

Most of the introduced variants that incorporate estimated global information into the `karger` algorithm use it to their advantage. They provide much better final results in terms of both number of moved items and the imbalance of the system. Especially good performances are achieved by the `avg2`, `stddev2` and self-tuning variants which combined with each other provide even better results. An improvement of up to 60% less item movements with a 30% better standard deviation can be achieved using an optimal value for `stddev2`'s $s$ parameter. These effects can be observed with the scenarios that have a greater imbalance at the simulations' start. Scenarios which already start with smaller imbalances only show improvements of up to 40% less item movements and a 15% better imbalance at the simulations' end.

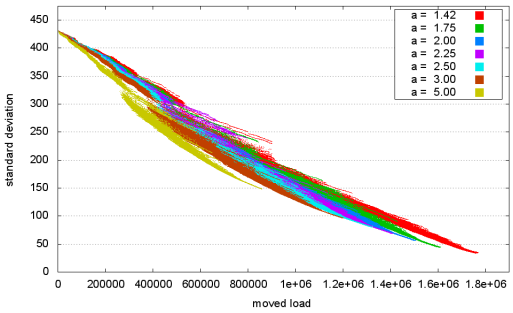
Simulations have also shown that the best two algorithms, i.e. `karger_avg2_stddev2` and `karger_self_avg2_stddev2`, are pretty robust against fluctuations in the quality of the estimated global information they use. They can however not provide the same improvements on the final imbalance with too erroneous data but at least still show major improvements in the number of moved items. With an error of 80% and in an alphabetical scenario (high starting imbalance) still about 40% and 55% less items are moved. Compared to the ordinary `karger` algorithm though, the final imbalance is around 10% worse.

Better results by sampling multiple random nodes can only be observed with the original `karger` algorithm. Sampling more nodes in `karger_avg2_stddev2` produces measurable but insignificant improvements over one sampled node. Differences of sampling either 2, 4 or 8 nodes are negligible though. If self-tuning is also applied to this algorithm variant changes in the number of sampled nodes can nearly be observed and can be buried in the variance of the results among the different test runs.

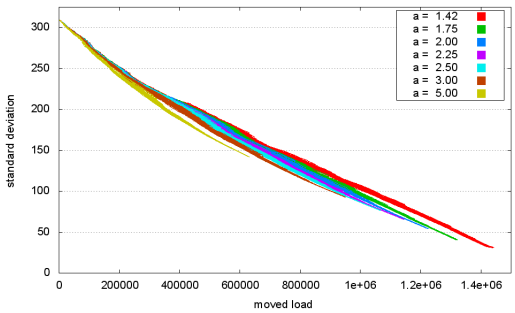
Further simulations of an alphabetical scenario with different total loads or different system sizes have also shown that these two variants scale linearly with those changes. They also offer the same improvements over the original algorithm in terms of both, moved items and imbalance at the end of the simulations. The only thing that can be noticed though is that if the system size, i.e. number of nodes, increases more algorithm executions are needed than with the ordinary `karger` algorithm. This is due to the fact that these variants omit several balance operations and wait for better node matches. It should be considered if these algorithms are applied to real-world scenarios since in order to achieve the same imbalance in the same time they would need to be called more often, e.g. 4 times as much (`karger` already achieves its results with 100 algorithm executions). Further investigations on the aspect of *time*, that was previously ignored, are needed in order to draw any further conclusions to applications on real systems.

### 4.4.2. Mercury

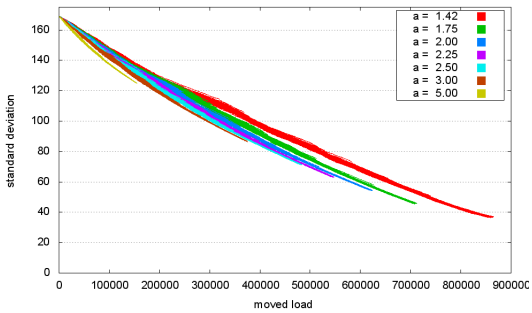
All of the introduced algorithm variants have also been applied to a second algorithm, `mercury`, in order to evaluate whether the effects observed with `karger` also apply to other balance algorithms. The following sections will analyse those variations the same way as the analysis has been done with `karger` above. In contrast to this though it will not be as thorough as before and will concentrate only on proving the effectiveness of the variants and will neither evaluate the influence of the number of algorithm executions nor the effect of multiple sampled nodes. The evaluation will thus also only use the three main scenarios that have been used with the `karger` variants. The other scenarios exhibited very similar results, so the simulated scenarios can be restricted to those three without loss of generality.




(a) Wikipedia page titles (en)




(b) Exponential - $Exp(\lambda = 6 \cdot 10^{-19})$)




(c) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$

Figure 4.9.: *Balance results for* `mercury` *with different* $\alpha$ *for the three main scenarios (error rate* $= 25\%$*).*

#### Without added global information

At first the original `mercury` algorithm has been simulated with different values of the $\alpha$ parameter in order to evaluate its influence. Since $\alpha \geq \sqrt{2}$, values starting from 1.42 have been chosen up to a value that still reaches a fair imbalance. Those values include 1.42, 1.75, 2.00, 2.25, 2.50, 3.00 and 5.00. The simulation results of `mercury` with those

values are presented in Figure 4.9. The plot for each scenario shows the imbalance reached after moving the given number of items and thus indicates the relation between the two invariants *quality*, i.e. imbalance, and *cost*. As with `karger` it can be seen that parameters that tolerate bigger skews, i.e. larger $\alpha$'s, achieve the same standard deviation by moving less items than those tolerating smaller skews. This behaviour is important for the idea of self-tuning algorithms that will be analysed in the following sections. In contrast to `karger` though, results of the 100 different test runs of the simulation on the alphabetical scenario with a given $\alpha$ vary much more which is probably due to the error rate being at 25% and the definition of *local load* in `mercury`.

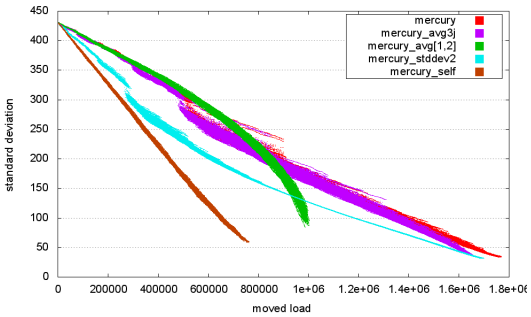
A fixed $\alpha$ of 1.42 is used for the following simulations because it results in the best imbalance at the end. The imbalance reached by this value and the number of moved items at the end of each simulation has been included in Table 4.7 on page 84.

#### With added global information

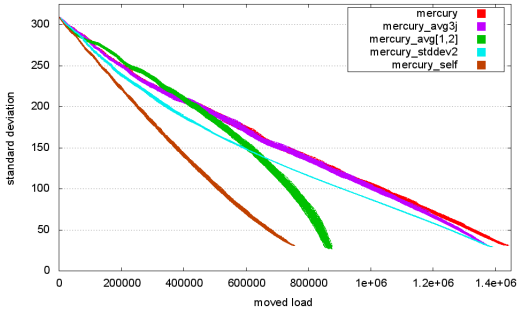
The results of the different algorithm variations that implement one of the variants introduced above are shown in Figure 4.10. In contrast to `karger` though the **`avg3j`** variant alone does perform a bit better than the original algorithm by moving less items and achieving only slightly worse imbalances at the end of the simulations in all three scenarios. This is probably due to *light nodes* (according to the definition in `mercury`) having lightly loaded neighbours with high probability and thus the additional restriction of `avg3j` not being hit that often as in `karger` (recall that only light nodes jump).

The performances of the two **`avg1`** and **`avg2`** variants are so similar when applied to `mercury`, that they could not be distinguished in the plots which is why they have been merged to a single scatter plot. Their results however are very similar to the ones they exhibited with `karger` although they do not show such bad results at the start of a simulation. The imbalance reached by these two variants at the end of the alphabetical simulations however is disappointing (ref. Table 4.7). This can only be explained by too many nodes getting a load that neither makes them *light* nor *heavy* and thus does not allow further balancing. Maybe not enough light nodes exist in order to balance the remaining heavy nodes. This however does not occur in the other two scenarios which is probably because there neighbouring nodes have a similar load with higher probability than in the alphabetical scenario. Averaging the load of three neighbouring nodes to a *local load* may thus lead to the node being neither heavy nor light although it is very heavily or lightly loaded.

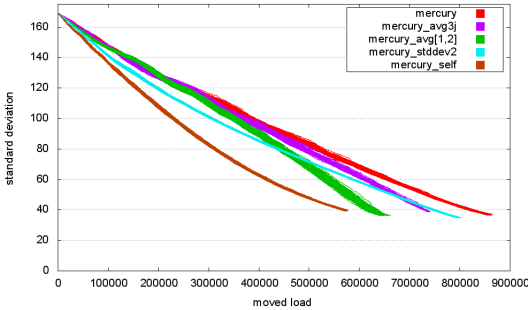
Results of the **`stddev2`** variant are as expected: a slightly better imbalance at the end of the simulations with less moved items than the original algorithm. In order to achieve that though, its $s$ parameter was set to 3.0 which seemed to be the best for `mercury_stddev2` during initial simulations. This will allow only such balance opera-




(a) Wikipedia page titles (en)




(b) Exponential - $Exp(\lambda = 6 \cdot 10^{-19})$)




(c) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$

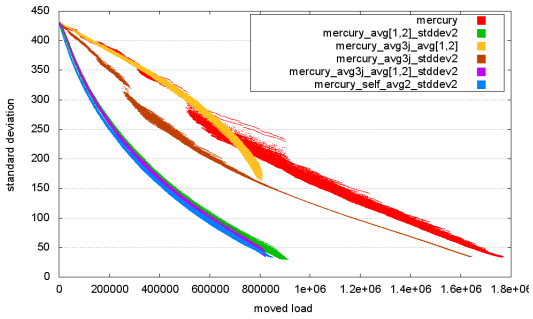
Figure 4.10.: *Balance results for the* `mercury` *variants with one variation, error rate* 25%.

tions, that improve the standard deviation by at least a factor of $^{3.0}/_{n}$ which is more selective than the $^{1.5}/_{n}$ used by the `karger` variants. In contrast to those however, the difference in the number of moved items compared to the original algorithm is not that high. Possibly most balance operations performed by `mercury` are already worth it and are thus not prevented by this variation.

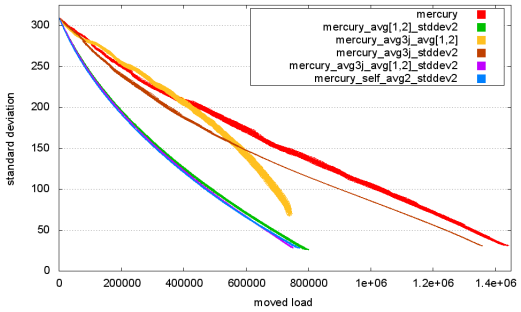
The **self-tuning** algorithm implemented for `mercury` looks quite superior to the other variants in the given plots - similarly to the results of the self-tuning variant of the `karger` algorithm. It moves a lot less items in all three scenarios and in the exponential scenarios it achieves an imbalance that is comparable to the one of the ordinary algorithm. In the alphabetical scenario however its final imbalance is disappointing. The huge variance of this value among the 100 test runs however indicates that the number of algorithm executions is not high enough for this variant which has been confirmed by additional simulations that reach better imbalances with an increased number of executions, e.g. an imbalance around 39 with a lower variance and around 830 000 moved items can be reached by doubling this number.

Results of algorithms with several variants combined are shown in Figure 4.11. Again **`avg1`** and **`avg2`** variants have been too similar to distinguish from each other and have been merged. Apparently the balance operations of mercury and its definition of local

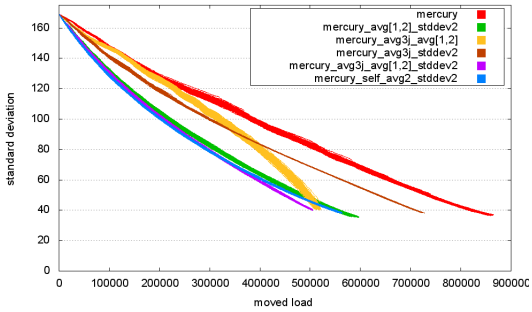
load have a greater influence on the imbalance than the number of items moved during such an operation which does not differ substantially in most cases with those two variants. Combining either of them with **avg3j** though is - as with karger - not a good idea since the resulting algorithm's performance is not as good as the other combinations and does not get close to the imbalance the original algorithm reaches at the end of the simulations. Only the last scenario exhibits good results with that combination which is probably due to its better imbalance at the simulation start. Adding **avg3j** to **stddev2** does not show much difference compared to the results of stddev2 alone which has been observed with the karger algorithm, too.




(a) Wikipedia page titles (en)




(b) Exponential - $Exp(\lambda = 6 \cdot 10^{-19})$)




(c) Exponential - $Exp(\lambda = 2 \cdot 10^{-19})$

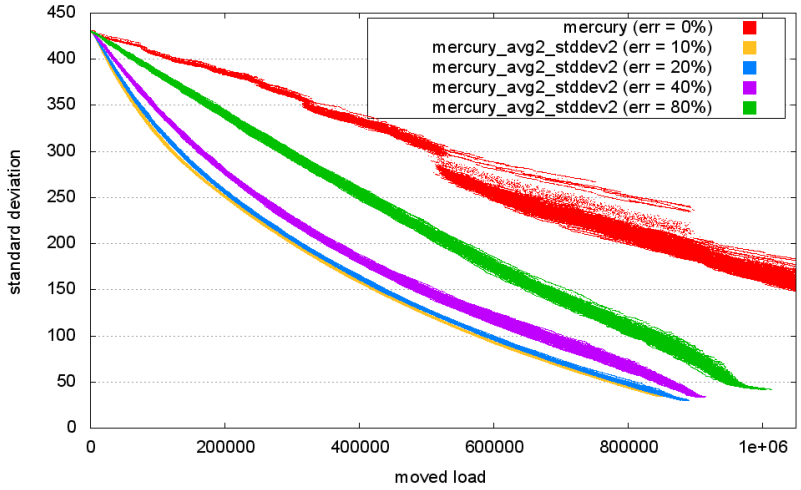
Figure 4.11.: *Balance results for the combined* **mercury** *variants, error rate* 25%. *Indistinguishable curves have been merged.*

The remaining algorithms are running shoulder to shoulder. Best final imbalances are achieved by the **avg1_stddev2** and **avg2_stddev2** variants which can be traded for a little worse imbalance by the advantage of moving slightly less items with the **avg3j_avg1_stddev2** and **avg3j_avg2_stddev2** variants. The **self-tuning** algorithm's performance is somewhere in between these two groups. In contrast to karger it does not outperform the other variants and the good results of self-tuning alone compared to other variants alone could unfortunately not be combined with the good results of another variant as this was the case with karger. Maybe $\alpha \geq \sqrt{2}$ is the limiting factor

here or the *local load* paradigm and its node classification in general. The fact that so many algorithms perform almost identically in contrast to the observations with `karger` would support that statement.

### Error rate

Simulations carried out with different error rates as above show that the `avg2_stddev2` variant applied to `mercury` performs similarly than the same variant on `karger`. The only difference that should be noted here is that the performance of the algorithm with an error of 10% is closer to the one with a 20% error than before. Also, the 10% scenario shows a worse imbalance result at the end of the simulations which is probably due to less items being transferred to another node than with 20% and heavy nodes thus not getting "normal" and stealing light nodes that otherwise would have been matched with more unbalanced nodes.




(a) `karger_avg2_stddev2` ($e = 0.24,\ s = 2$)

| Algorithm | | Error | Simulation results (avg) | | |
|---|---|---|---|---|---|
| *name* | *parameters* | | *moved load* | *stddev* | $\delta_{mal}$ |
| mercury | $a = 1.42$ | $\leq$ 0% | 1594504.07 $^{\pm 0.43\%}$ | 50.13 $^{\pm 0.91\%}$ | 3.23 |
| ..avg2_stddev2 | $a = 1.42, s = 3$ | $\leq$ 10% | 844367.93 $^{\pm 1.01\%}$ | 35.19 $^{\pm 2.92\%}$ | 3.18 |
| | $a = 1.42, s = 3$ | $\leq$ 20% | 883390.55 $^{\pm 1.36\%}$ | 30.48 $^{\pm 2.93\%}$ | 3.00 |
| | $a = 1.42, s = 3$ | $\leq$ 40% | 907052.92 $^{\pm 1.07\%}$ | 34.01 $^{\pm 2.42\%}$ | 2.81 |
| | $a = 1.42, s = 3$ | $\leq$ 80% | 998260.86 $^{\pm 1.60\%}$ | 42.36 $^{\pm 1.82\%}$ | 3.05 |

(b) Results at the end of the simulations

Figure 4.12.: *Results of the best* `mercury` *variant with different error rates. (Wikipedia (en), 100 test runs, 200 algorithm executions each, plot cut off at 1.05m moved items)*

| Scenario | Algorithm | Simulation results (avg) | | |
|---|---|---|---|---|
| *name* | *name* | *moved load* | *stddev* | $\delta_{mal}$ |
| Wikipedia (en) | mercury | 1761353.09 ±0.49% | 34.43 ± 2.36% | 2.87 |
| | ..._avg1 | 991167.62 ±1.21% | 105.80 ±17.72% | 95.34 |
| | ..._avg2 | 981093.10 ±1.57% | 105.99 ±20.78% | 95.04 |
| | ..._avg3j | 1658430.86 ±0.43% | 35.45 ± 2.31% | 3.13 |
| | ..._stddev2 | 1692875.04 ±0.46% | 31.81 ± 1.67% | 2.52 |
| | ..._self | 720993.42 ±5.89% | 72.92 ±25.88% | 49.45 |
| | ..._avg1_stddev2 | 904086.45 ±1.06% | 30.37 ± 7.82% | 3.58 |
| | ..._avg2_stddev2 | 887620.31 ±1.02% | 30.84 ± 4.72% | 3.22 |
| | ..._avg3j_avg1 | 804079.98 ±0.93% | 174.60 ± 8.97% | 126.44 |
| | ..._avg3j_avg2 | 791579.36 ±0.61% | 174.31 ± 6.65% | 126.61 |
| | ..._avg3j_stddev2 | 1638856.48 ±0.29% | 34.51 ± 1.33% | 2.86 |
| | ..._avg3j_avg1_stddev2 | 819585.53 ±0.83% | 38.27 ±21.87% | 16.85 |
| | ..._avg3j_avg2_stddev2 | 819585.53 ±0.83% | 38.27 ±21.87% | 16.85 |
| | ..._self_avg2_stddev2 | 841609.76 ±1.69% | 34.63 ± 9.47% | 3.80 |
| $Exp(\lambda = 6 \cdot 10^{-19})$ | mercury | 1436953.99 ±0.55% | 31.51 ± 2.24% | 2.48 |
| | ..._avg1 | 868365.51 ±0.98% | 29.12 ±16.86% | 9.22 |
| | ..._avg2 | 863661.82 ±1.27% | 29.49 ±17.17% | 9.02 |
| | ..._avg3j | 1367239.74 ±0.29% | 32.45 ± 1.54% | 2.76 |
| | ..._stddev2 | 1387009.40 ±0.31% | 29.54 ± 1.55% | 2.36 |
| | ..._self | 751612.57 ±0.70% | 31.32 ± 1.87% | 2.89 |
| | ..._avg1_stddev2 | 797138.00 ±0.67% | 26.47 ± 2.46% | 2.55 |
| | ..._avg2_stddev2 | 790562.53 ±0.79% | 26.73 ± 2.52% | 2.53 |
| | ..._avg3j_avg1 | 742164.69 ±0.74% | 74.06 ±10.63% | 27.45 |
| | ..._avg3j_avg2 | 736134.58 ±0.73% | 73.33 ±11.68% | 27.50 |
| | ..._avg3j_stddev2 | 1355825.40 ±0.24% | 31.07 ± 1.23% | 2.44 |
| | ..._avg3j_avg1_stddev2 | 749505.40 ±0.47% | 28.48 ± 2.03% | 2.83 |
| | ..._avg3j_avg2_stddev2 | 749505.40 ±0.47% | 28.48 ± 2.03% | 2.83 |
| | ..._self_avg2_stddev2 | 768556.88 ±0.62% | 28.14 ± 1.50% | 2.70 |
| $Exp(\lambda = 2 \cdot 10^{-19})$ | mercury | 858594.53 ±1.03% | 36.98 ± 1.50% | 2.69 |
| | ..._avg1 | 649470.11 ±2.05% | 36.76 ± 1.64% | 2.71 |
| | ..._avg2 | 644264.44 ±1.68% | 36.98 ± 2.42% | 2.72 |
| | ..._avg3j | 737164.36 ±1.03% | 39.15 ± 1.43% | 3.02 |
| | ..._stddev2 | 795476.38 ±0.88% | 35.11 ± 1.52% | 2.58 |
| | ..._self | 570593.90 ±1.26% | 39.80 ± 1.86% | 2.95 |
| | ..._avg1_stddev2 | 590802.90 ±1.62% | 35.88 ± 2.29% | 2.64 |
| | ..._avg2_stddev2 | 583538.73 ±1.18% | 36.10 ± 1.50% | 2.65 |
| | ..._avg3j_avg1 | 516676.43 ±1.12% | 41.81 ± 8.50% | 9.72 |
| | ..._avg3j_avg2 | 510947.23 ±1.11% | 41.87 ± 7.15% | 9.21 |
| | ..._avg3j_stddev2 | 720274.42 ±1.02% | 38.80 ± 1.69% | 2.76 |
| | ..._avg3j_avg1_stddev2 | 501536.74 ±1.54% | 40.71 ± 1.99% | 3.13 |
| | ..._avg3j_avg2_stddev2 | 501536.74 ±1.54% | 40.71 ± 1.99% | 3.13 |
| | ..._self_avg2_stddev2 | 558778.87 ±1.07% | 38.53 ± 1.51% | 2.84 |

Table 4.7.: *Results of the different* `mercury` *variants, best variants for each scenario marked in yellow (error rate* = 25% *unless otherwise stated,* $\alpha = 1.42$, $s = 3.0$ *where appropriate,* 100 *test runs with* 200 *algorithm executions each).*

### Scalability

As can be seen from the results in Table 4.8, `mercury` does scale linearly with the overall load as does its `avg2_stddev2` variant. It also continues to show the same improvements in terms of percentages compared to the original algorithm.

| Algorithm | Total load | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 500 000 | | 1 000 000 | | 2 000 000 | | 4 000 000 | |
| | *moved l.* | *stddev* | *moved l.* | *stddev* | *moved l.* | *stddev* | *moved l.* | *stddev* |
| mercury | 878334 ±0.58% | 17.28 ±1.79% | 1761353 ±0.49% | 34.43 ±2.36% | 3527486 ±0.57% | 68.83 ±2.30% | 7056172 ±0.63% | 137.71 ±1.76% |
| ..._av2_st2 | 443357 ±1.36% | 15.56 ±5.05% | 887620 ±1.02% | 30.84 ±4.72% | 1776176 ±1.02% | 61.52 ±5.09% | 3549476 ±1.34% | 123.04 ±4.42% |

Table 4.8.: *Average results of the* `mercury` *variants with different total loads (Wikipedia (en), error rate* 0% *with* `karger`*, otherwise* 25%*,* $\alpha = 1.42$*,* $s = 3.0$ *where appropriate,* 100 *test runs,* 200 *algorithm executions each,* 10 000 *nodes, moved load rounded down to the nearest integral, abbreviated algorithm names).*

Also these algorithm's performances in scenarios with more number of nodes, shown in Table 4.9, show similar results than what has been observed with `karger`. `mercury` moves about 10% more items in scenarios with twice as many nodes just like `karger` did and achieves nearly halved imbalances as well. The `avg2_stddev2` variant however moves only slightly more items comparing the simulations with 5 000 and 10 000 nodes but already starts to show the effect of not being executed often enough with 20 000 nodes which is supported by the variance that is shown by the results. This is similar to the observations with `karger` although there this effect with the non-self-tuning variant did start to occur with 40 000 nodes.

| Algorithm | Number of nodes | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 5 000 | | 10 000 | | 20 000 | | 40 000 | |
| | *moved l.* | *stddev* | *moved l.* | *stddev* | *moved l.* | *stddev* | *moved l.* | *stddev* |
| mercury | 1615172 ±0.69% | 69.68 ±2.99% | 1761353 ±0.49% | 34.43 ±2.36% | 1941249 ±0.40% | 17.05 ±1.57% | 2113715 ±0.47% | 8.31 ±1.33% |
| ..._av2_st2 | 869918 ±1.27% | 62.97 ±2.87% | 887620.31 ±1.02% | 30.84 ±4.72% | 873597 ±3.39% | 24.80 ±24.67% | 724608 ±17.11% | 29.05 ±38.04% |

Table 4.9.: *Average results of the* `mercury` *variants with different system sizes (Wikipedia (en), error rate* 0% *with* `karger`*, otherwise* 25%*,* $\alpha = 1.42$*,* $s = 3.0$ *where appropriate,* 100 *test runs,* 200 *algorithm executions each, total load* 1 000 000*, moved load rounded down to the nearest integral, abbreviated algorithm names).*

### Summary of Results

Based on the simulations on the three main scenarios carried out and analysed above, the best `mercury` variant is `mercury_avg2_stddev2` without self-tuning. It achieves an up to 15% lower imbalance at the end of the simulations by moving up to 50% fewer items than the original algorithm (depending on the scenario). This confirms the results that have been observed with this variant on the `karger` algorithm above and indicates that the possible improvements are not limited to the presented algorithms. Only the superiority of this variant being equipped with a self-tuning parameter could not be confirmed. However the implementation of this on `mercury` is different to the implementation on `karger` which is why the results can not necessarily be transferred. There are however indications that `mercury` itself is limiting any more improvements since several algorithms that have previously showed different results now almost perform identically.

The robustness of the algorithm variants that has been observed with `karger` though still exist here. `mercury`'s best variant `mercury_avg2_stddev2` shows the same minor influence to changed error rates as `karger_avg2_stddev2` does. The same can be said about the scalability of this variant in regard to increased overall loads and increased number of nodes. It seems that those effects can be transferred to another algorithm if the algorithm itself already shows them (which `mercury` does).

# 5. Conclusion

## 5.1. Achievements

At the beginning in Chapter 2 an overview of the field of research was given. It introduced Distributed Hash Tables (DHTs) and some of their representatives, e.g. CAN, Pastry and Chord. This general concept has then been extended to such DHTs that support range queries among their stored data, i.e. they do not only allow the retrieval of the value of a set of single keys but also support queries for ranges of them. Among several implementations of such DHTs, Mercury and Scalaris have been introduced that show only few or no overhead to ordinary DHTs without range queries. Additionally, gossiping techniques have been presented that are able to aggregate global information of a DHT with high confidence and low overhead. Among those estimated values is the system's minimum, average and maximum load, the standard deviation of the load among the nodes as well as the number of nodes in the system.

The problem that arises by the way range-queriable DHTs store their data is the increased variance of *load* among different nodes. They thus apply some sort of load balancing scheme. Chapter 2 also introduced several novel load balancing algorithms that have been developed in recent research. These algorithms have been arranged into 4 different categories since most of them make use of a common set of techniques and only differ in details.

Two of the 18 presented algorithms have then been chosen and equipped with (additional) estimates of global information with the objective of improving their performances. Chapter 3 introduced the system model that is used for the evaluation and presents the two algorithms in more detail. It concludes by introducing five different techniques of using information such as the average and maximum load, the standard deviation of the load among the nodes and system size and describes the ideas behind.

These techniques have then been integrated into the algorithms and evaluated by simulation. Chapter 4 described the simulator that has been set up for this evaluation and defined the metrics that have been used in order to rate the different algorithms. It further presented the simulation scenarios the algorithms should master and finally evaluated their performances. During these evaluations an algorithm variant combining several of the ideas introduced above has been found that significantly increases the

performance of both original algorithms. When applied to any of the given scenarios, the imbalance reached at the end of the simulations is about $15-30\%$ lower and the number of moved items has even been decreased by about $30-50\%$. For this achievement three global estimates are used: the average load, the standard deviation of the load among the nodes and the number of nodes. Further simulations also verified that this variant is quite robust regarding the accurateness of the estimates and also scales linearly to scenarios with higher overall loads or increased number of nodes. With the algorithm described by Karger and Ruhl, the number of moved items can even be further reduced by using a so-called self-tuning variant that sets the algorithm's $\epsilon$ parameter according to the system's state. This variation then moved only $40-65\%$ of the amount the original algorithm moves by achieving a $15-30\%$ lower imbalance. Unfortunately this success was not observed with the self-tuning variant developed for the second algorithm.

This evaluation supports the thesis from the beginning that load balancing algorithms can profit from added global information such that they show better performances. As can also be seen, major improvements can be expected by such variations of an algorithm.

## 5.2. Future Work

As already mentioned in the evaluation above, some of the algorithm's aspects need further investigation. There is at first the concept of *time* which has been omitted in the current system model but is needed for real-world applications. It needs to be evaluated how much more often the new algorithm variants need to be executed in order to show the improved results they exhibited here. It will then need to be evaluated whether the additional operations performed by the omitted balance operations, e.g. getting random nodes, are still negligible in terms of impact on the network.

Another aspect that still needs further attention is a different definition of *load*. The system model used here assumes that the load of a node is proportional to the number of items it stores and so is the transfer cost. This equals a real-world scenario where every item in the system has the same size and the storage on the nodes is to be balanced. More often, another aspect of the stored items is crucial for the system's performance/availability: the popularity of the stored items and the resulting number of item accesses. Other definitions of load may take into account the nodes' (potentially different) capacities of network bandwidth and latency towards other nodes. Systems with heterogeneous nodes in general would also need to be further observed.

Finally different (additional) global estimates may further improve the algorithms and deploying the introduced values in other way might potentially be useful as well. Especially the concept of self-tuning parameters seems promising and may be further extended to different algorithms, too, or be used to create a new algorithm from scratch.

# Bibliography


[1] Gnutella protocol specification. http://wiki.limewire.org/index.php?title=GDF. version from: 04/05/2009.

[2] Marble desktop globe. http://edu.kde.org/marble. Version 0.7.1.

[3] Scalaris. http://code.google.com/p/scalaris/.

[4] Wikipedia database dumps. http://download.wikimedia.org/backup-index.html.

[5] Wikipedia: Gnutella. http://en.wikipedia.org/w/index.php?title=Gnutella&oldid=315766949. version from: 23/09/2009, 19:36.

[6] Wikipedia: Space-filling curve. http://en.wikipedia.org/w/index.php?title=File:Hilbert_curve.svg&oldid=313758608. version from: 14/09/2009, 06:03.

[7] *Secure Hash Standard*. National Institute of Standards and Technology, Washington, 2002. Federal Information Processing Standard 180-2.

[8] Karl Aberer, Luc Onana Alima, Ali Ghodsi, Sarunas Girdzijauskas, Manfred Hauswirth, and Seif Haridi. The essence of p2p: A reference architecture for overlay networks. In *Proceedings of 5th IEEE International Conference on Peer-to-Peer Computing*, pages 11–20, Konstanz, Germany, 2005.

[9] Artur Andrzejak and Zhichen Xu. Scalable, efficient range queries for grid information services. *Peer-to-Peer Computing, IEEE International Conference on*, 0:33–40, 2002.

[10] James Aspnes, Jonathan Kirsch, and Arvind Krishnamurthy. Load balancing and locality in range-queriable data structures. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 115–124, New York, NY, USA, July 2004. ACM.

[11] Kevin Atkinson. Scowl (spell checker oriented word lists) rev 6. http://wordlist.sourceforge.net/. last access: 23/09/2009.

[12] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4):353–366, 2004.

[13] Marcin Bienkowski, Miroslaw Korzeniowski, and Friedhelm Meyer auf der Heide. Dynamic load balancing in distributed hash tables. In *Peer-to-Peer Systems IV*, volume 3640 of *Lecture Notes in Computer Science*, pages 217–225. Springer Berlin / Heidelberg, 2005.

[14] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables. In *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 80–87. Springer Berlin / Heidelberg, 2003.

[15] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony I. T. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 103–107. Springer Berlin / Heidelberg, 2003.

[16] Michel Charpentier, Gérard Padiou, and Philippe Quéinnec. Cooperative mobile agents to gather global information. In *Fourth IEEE International Symposium on Network Computing and Applications*, pages 271–274, July 2005.

[17] Yatin Chawathe, Sriram Ramabhadran, Sylvia Ratnasamy, Anthony LaMarca, Scott Shenker, and Joseph Hellerstein. A case study in building layered dht applications. *SIGCOMM Comput. Commun. Rev.*, 35(4):97–108, 2005.

[18] Chyouhwa Chen and Kun-Cheng Tsai. The server reassignment problem for load balancing in structured p2p systems. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):234–246, 2008.

[19] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, pages 46–66. Springer Berlin / Heidelberg, 2001.

[20] Free Software Foundation. GPL. http://www.gnu.org/licenses/gpl.html. last access: 20/08/2009.

[21] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 444–455. VLDB Endowment, 2004.

[22] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One torus to rule them all: multi-dimensional queries in p2p systems. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 19–24, New York, NY, USA, 2004. ACM.

[23] Ali Ghodsi, Seif Haridi, and Hakim Weatherspoon. Exploiting the synergy between gossiping and structured overlays. *SIGOPS Oper. Syst. Rev.*, 41(5):61–66, 2007.

[24] George Giakkoupis and Vassos Hadzilacos. A scheme for load balancing in heterogenous distributed hash tables. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 302–311, New York, NY, USA, 2005. ACM.

[25] P. Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard M. Karp, and Ion Stoica. Load balancing in dynamic structured p2p systems. In *INFOCOM*, 2004.

[26] P. Brighten Godfrey and Ion Stoica. Heterogeneity and load balance in distributed hash tables. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 1, pages 596–606, 2005.

[27] Mikael Högqvist, Seif Haridi, Nico Kruber, Alexander Reinefeld, and Thorsten Schütt. Using global information for load balancing in dhts. In *Workshop on Decentralized Self Management for Grids, P2P, and User Communities*, volume 0, pages 236–241, Los Alamitos, CA, USA, October 2008. IEEE Computer Society.

[28] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.

[29] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Mark Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[30] David R. Karger and Matthias Ruhl. Simple efficient load-balancing algorithms for peer-to-peer systems. *Theory of Computing Systems*, 39(6):787–804, November 2006.

[31] Krishnaram Kenthapadi and Gurmeet Singh Manku. Decentralized algorithms using both local and random probes for p2p load balancing. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 135–144, New York, NY, USA, 2005. ACM.

[32] Jonathan Ledlie and Margo Seltzer. Distributed, secure load balancing with skew, heterogeneity and churn. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 2, pages 1419–1430, March 2005.

[33] Daniel Mark Lewin. Consistent hashing and random trees : algorithms for caching in distributed networks. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, May 1998.

[34] Gurmeet Singh Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 197–205, New York, NY, USA, July 2004. ACM.

[35] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. Replication, load balancing and efficient range query processing in DHTs. In *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 131–148. Springer Berlin / Heidelberg, March 2006.

[36] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in structured P2P systems. In *Peer-to-Peer Systems II*,

volume 2735/2003 of *Lecture Notes in Computer Science*, pages 68–79. Springer Berlin / Heidelberg, 2003.

[37] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, 2001.

[38] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350, 2001.

[39] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Range queries on structured overlay networks. *Computer Communications*, 31(2):280–291, February 2008. Special Issue: Foundation of Peer-to-Peer Computing.

[40] Qt Software. Qt 4.5.2. http://qt.nokia.com/. last access: 20/08/2009.

[41] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, August 2001.

[42] Dimitri van Heesch. Doxygen. http://www.stack.nl/~dimitri/doxygen/. last access: 20/09/2009.

[43] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.

[44] Zhiyong Xu and Laxmi N. Bhuyan. Effective load balancing in p2p systems. In *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06*, volume 1, pages 81–88. IEEE Computer Society, May 2006.

[45] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: a fault-tolerant wide-area application infrastructure. *SIGCOMM Comput. Commun. Rev.*, 32(1):81, 2002.

# A. Implemented algorithms in Pseudo-Code

## A.1. Generic helper functions

These are some of the generic helper functions that are used by every algorithm. They make use of a `calcBalancedLoad` method which must be defined for each algorithm.

```
1  // slide between n_i and its successor n_j
2  // return the actual load changes this operation will have
3  slideHelper(DHT d, Node n_i, Node n_j, bool simulate) {
4   // calculate the amount of load that should be moved between the nodes:
5   LoadMove loadMove = calcBalancedLoad(d, n_i, n_j);
6   if (simulate) {
7    return d.simulateSlide(n_i, n_j, loadMove);
8   } else {
9    return d.slide(n_i, n_j, loadMove);
10  }
11 }
12
13 // move n_j to support n_i
14 // return the actual load changes this operation will have
15 jumpHelper(DHT d, Node n_j, Node n_i, bool simulate) {
16  // calculate the amount of load that should be moved from n_i
17  // to the empty n_j after it has been moved:
18  LoadMove loadMove = calcBalancedLoad(d, n_i, emptyNode);
19  if (simulate) {
20   return d.simulateJump(n_j, n_i, loadMove);
21  } else {
22   return d.jump(n_j, n_i, loadMove);
23  }
24 }
```

## A.2. Variations of `calcBalancedLoad`

The following method can be used by algorithms which want to even out the load of two nodes they decided to balance with each other. It will try give both nodes half of the sum of their loads. The exact number of transferred items is dependent on their load and will be determined by the `slide` and `simulateSlide` operations.

```
1  calcBalancedLoad_half(DHT d, Node first, Node second) {
2    LoadMove loadToMove;
3    Node n_fat = NULL;
4    Node n_slim = NULL;
5    if (load(first) > load(second)) {
6      n_fat = first;
7      n_slim = second;
8      loadToMove.direction = FirstToSecond;
9    } else { // load(first) <= load(second)
10     n_fat = second;
11     n_slim = first;
12     loadToMove.direction = SecondToFirst;
13   }
14   loadToMove.load = (load(n_fat) - load(n_slim)) / 2;
15   return loadToMove;
16 }
```

Another implementation might want to try not to move more than the average load for which an estimate is retrieved from the DHT. It will otherwise do the same as `calcBalancedLoad_half`.

```
1  calcBalancedLoad_avg1(DHT d, Node first, Node second) {
2    LoadMove loadToMove;
3    Node n_fat = NULL;
4    Node n_slim = NULL;
5    if (load(first) > load(second)) {
6      n_fat = first;
7      n_slim = second;
8      loadToMove.direction = FirstToSecond;
9    } else { // load(first) <= load(second)
10     n_fat = second;
11     n_slim = first;
12     loadToMove.direction = SecondToFirst;
13   }
14   double avg = d.getAvgLoad();
15   loadToMove.load = min(avg, (load(n_fat) - load(n_slim)) / 2);
16   return loadToMove;
17 }
```

The third implementation only balances heavy nodes (those with a load higher than the average) with light nodes (less load than the average) and never make a light node heavy. It will also move no more items than are required to make the heavy node balanced (load equal to the average).

```
1  calcBalancedLoad_avg1(DHT d, Node first, Node second) {
2    LoadMove loadToMove;
3    Node n_fat = NULL;
4    Node n_slim = NULL;
5    if (load(first) > load(second)) {
6      n_fat = first;
7      n_slim = second;
8      loadToMove.direction = FirstToSecond;
9    } else { // load(first) <= load(second)
10     n_fat = second;
11     n_slim = first;
12     loadToMove.direction = SecondToFirst;
13   }
14   double avg = d.getAvgLoad();
15   if (load(n_fat) > avg && load(n_slim) < avg) {
16     loadToMove.load = min(load(n_fat) - avg, avg - load(n_slim));
17   } else {
18     loadToMove.load = 0;
19   }
20   return loadToMove;
21 }
```

## A.3. Variations of `getBest`

The first way of finding a best node among a list of candidates uses only local knowledge and decides for the node that improves the standard deviation the most (without knowing its exact value). As pictured in Section 3.2 only the change of the sum of the square of all loads needs to be examined which is done here.

```
1  getBest_stddev1(List<Node> candidates, Map<LoadChanges>
       results) {
2   Node bestNode = none;
3   double minSumLi2_change = 0;
4   foreach(Node n_j ∈ candidates) {
5    double currentChange = 0;
6    foreach(LoadChange lc ∈ results[n_j]) {
7     currentChange += lc.newLoad()^2 - lc.oldLoad()^2;
8    }
9    if (currentChange < minSumLi2_change) {
10    minSumLi2_change = currentChange;
11    bestNode = n_j;
12   }
13  }
14  return bestNode;
15 }
```

A second implementation will get an estimate of the old value of the standard deviation and an estimate of the size from the DHT and use them to calculate the new value. The best candidate is at first the one that improves the standard deviation the most, but additionally to the previous implementation, it is only used if by balancing this node the standard deviation would increase by at least $s/size$. Otherwise nothing is done.

```
1  getBest_stddev2(List<Node> candidates, Map<LoadChanges>
       results, double s) {
2   Node bestCandidate = getBest_stddev1(candidates, results);
3   if (exists(bestCandidate)) {
4    int size = d.getSize(); double oldStddev = d.getStddev();
5    double variance = oldStddev^2;
6    foreach(LoadChange lc ∈ results[n_j]) {
7     variance += lc.newLoad()^2 / size - lc.oldLoad()^2 / size;
8    }
9    double stddev = √variance;
10   if (stddev >= 0 && stddev < oldStddev * (1 - s / size)) {
11    return bestCandidate;
12   }
13  }
14  return none;
15 }
```

## A.4. Algorithms based on the item balancing scheme by Karger and Ruhl

### Basic algorithm

The following listing shows the basic algorithm as it is shared between most of the variations. Each algorithm implementation has to provide an implementation of the `calcBalancedLoad` and the `getBest` method and can override any of the given methods.

```
1  karger_item(DHT d, double e /*ϵ*/, int k /*samples*/) {
2   foreach (Node n_i ∈ d) {
3    // get k unique random nodes that are not equal to n_i:
4    List<Node> candidates = d.getUniqueRandomNodes(n_i, k);
5    Map<LoadChanges> results;
6    foreach (Node n_j ∈ candidates) {
7     results[n_j] = karger_helper(d, e, n_i, n_j, true);
8    }
9    Node n_j = getBest(candidates, results);
10   if (exists(n_j)) {
11    karger_helper(d, e, n_i, n_j, false);
12   }
13  }
14 }
15
16 karger_helper(DHT d, double e, int k, bool simulate) {
17  if (load(n_i) ≤ e * load(n_j)) { // load(n_j) > load(n_i)
18   return karger_balance(d, n_j, n_i, simulate);
19  } else if (load(n_j) ≤ e * load(n_i)) { // load(n_i) > load(n_j)
20   return karger_balance(d, n_i, n_j, simulate);
21  }
22  return []; // no changes
23 }
24
25 karger_balance(DHT d, Node n_i, Node n_j, bool simulate) {
26  if (n_i == n_{j+1}) {
27   return slideHelper(n_i, n_j, simulate);
28  } else {
29   if (load(n_{j+1}) > load(n_i)) {
30    return slideHelper(n_j, n_{j+1}, simulate);
31   } else { // load(n_{j+1}) ≤ load(n_i) -> move n_j, balance with n_i
32    return jumpHelper(n_j, n_i simulate);
33   }
34  }
35  return []; // no changes
36 }
```

### Karger variations

The original `karger` algorithm uses `calcBalancedLoad_half` and `getBest_stddev1` as its implementations for `calcBalancedLoad` and `getBest` respectively. Variations of the original algorithm include the names of the used implementations in their own name, e.g. `karger_avg1_stddev2` uses `calcBalancedLoad_avg1` and `getBest_stddev2` and thus has an additional parameter `s`. It follows the pseudo-code of variants that need to be implemented inside `karger`'s main methods.

#### Variant `avg3j`

The implementation of `avg3j` only changes one method from the original algorithm:

```
1  karger_balance(DHT d, Node n_i, Node n_j, bool simulate) {
2   if (n_i == n_{j+1}) {
3    return slideHelper(n_i, n_j, simulate);
4   } else {
5    if (load(n_{j+1}) > load(n_i)) {
6     return slideHelper(n_j, n_{j+1}, simulate);
7    } else if (load(n_j) + load(n_{j+1}) ≤ d.getAvgLoad()) {
8     return jumpHelper(n_j, n_i simulate);
9    }
10  }
11  return []; // no changes
12 }
```

#### Self-tuning

The self-tuning variants of the `karger` algorithm only set a different value of the epsilon parameter for each node at each execution and then continue as the ordinary `karger`. Its main method is thus changed to:

```
karger_item(DHT d, int k /*samples*/) {
 foreach (Node n_i ∈ d) {
  double avgL = d.getAvgLoad(); double maxL = d.getMaxLoad();
  double stddev = d.getStddev();
  double e = bound(0.01, avgL / max(avgL+stddev, maxL-stddev), 0.24);
  // continue as before...
 }
}
```

## A.5. Algorithms based on Mercury's load balancing scheme

### Basic algorithm

As with Karger, the following listing presents Mercury's basic algorithm shared between most of its variations which need to provide implementations of the `calcBalancedLoad` and the `getBest` methods and can override any other method.

```
1  mercury(DHT d, double a /*α*/, int k /*samples*/) {
2   foreach (Node n_i ∈ d) {
3    if (isLight(n_i)) {
4     if (isHeavy(n_{i+1})) {
5      slideHelper(d, n_i, n_{i+1}, false);
6     } else if (isHeavy(n_{i-1})) {
7      slideHelper(d, n_{i-1}, n_i, false);
8     }
9    } else if (isHeavy(n_i)) {
10    // get k unique random nodes that are not equal to n_i:
11    List<Node> candidates = d.getUniqueRandomNodes(n_i, k);
12    Map<LoadChanges> results;
13    foreach (Node n_j ∈ candidates) {
14     if (isLight(n_j)) {
15      results[n_j] = mercury_helper(d, a, n_i, n_j, true);
16     } else {
17      results[n_j] = []; // no changes (do not balance!)
18     }
19    }
20    Node n_j = getBest(candidates, results);
21    if (exists(n_j)) {
22     mercury_helper(d, a, n_i, n_j, false);
23    }
24   }
25  }
26 }
27
28 mercury_helper(DHT d, double a, int k, bool simulate) {
29  // n_i may be lightly loaded ⇒ use most loaded node of n_i, n_{i-1}, n_{i+1}
30  Node n'_i = getMostLoaded(n_i, n_{i-1}, n_{i+1});
31  if (n_j.isNeighbourOf(n'_i)) {
32   return slideHelper(d, n'_i, n_j, simulate);
33  } else if (n'_i ≠ n_j) {
34   return jumpHelper(d, n_j, n'_i, simulate);
35  }
36 }
```

### Mercury variations

Similar to `karger`, the original `mercury` algorithm uses `calcBalancedLoad_half` and `getBest_stddev1` as its implementations for `calcBalancedLoad` and `getBest` respectively. Variations of the original algorithm include the names of the used implementations in their own name, e.g. `mercury_avg1_stddev2` uses `calcBalancedLoad_avg1` and `getBest_stddev2` and thus has an additional parameter `s`. It follows the pseudo-code of variants that need to be implemented inside `mercury`'s main methods.

#### Variant `avg3j`

The implementation of `avg3j` only changes one method from the original algorithm:

```
1 mercury_helper(DHT d, double a, int k, bool simulate) {
2  // n_i may be lightly loaded ⇒ use most loaded node of n_i, n_{i-1}, n_{i+1}
3  Node n'_i = getMostLoaded(n_i, n_{i-1}, n_{i+1});
4  if (n_j.isNeighbourOf(n'_i)) {
5   return slideHelper(d, n'_i, n_j, simulate);
6  } else if (n'_i ≠ n_j && load(n_j) + load(n_{j+1}) ≤ d.getAvgLoad()) {
7   return jumpHelper(d, n_j, n'_i, simulate);
8  }
9 }
```

#### Self-tuning

The self-tuning variants of the `karger` algorithm only set a different value of the alpha parameter for each node at each execution. Its main method is thus changed to:

```
mercury(DHT d, int k /*samples*/) {
 foreach (Node n_i ∈ d) {
  double avgL = d.getAvgLoad();
  double stddev = d.getStddev();
  double alpha = bound(1.42, (avgL + stddev) / avgL, 10.00);
  // continue as before...
 }
}
```