9. Reliability and Load Balancing in Distributed Hash Tables

Simon Rieche, Heiko Niedermayer, Stefan Götz, Klaus Wehrle
(University of Tübingen)

After introducing some selected Distributed Hash Table (DHT) systems, this chapter introduces algorithms for DHT-based systems which balance the storage data load (Section 9.1) or care for the reliability of the data (Section 9.2).

9.1 Storage Load Balancing of Data in Distributed Hash Tables

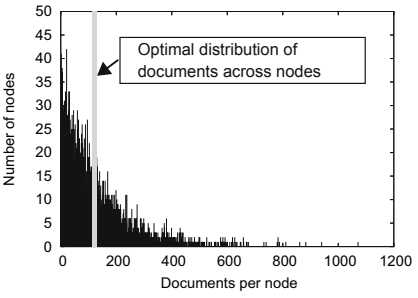
DHTs are used increasingly in widely distributed applications [52, 573]. Their efficient, scalable, and self-organizing algorithms for data retrieval and management offer crucial advantages compared with unstructured approaches. However, the underlying assumption is a roughly equal data distribution among the cooperating peers of a DHT. If there is a significant difference in the load of nodes in terms of data managed by each peer, i.e., data is concentrated on just a few peers, then the system may become less robust.

Alongside their crucial advantages, DHTs still show one major weakness in the distribution of data among the set of cooperating peers. All systems usually rely on the basic assumption that data is nearly equally distributed among the peer nodes. In most DHT approaches this assumption is based on the use of hash functions for mapping data into the DHT's address space. Generally, one assumes that hash functions provide an even distribution of keys and their respective data across the DHT address space. If there is a significant difference in the load of nodes in terms of data managed by each peer, the cost for distributed self-organization of such systems may increase dramatically. Therefore, appropriate mechanisms for load-balancing are required in order to keep the complexity of DHT search algorithms in the intended range of $O(logN)$ or less, where N is the number of nodes in the DHT.

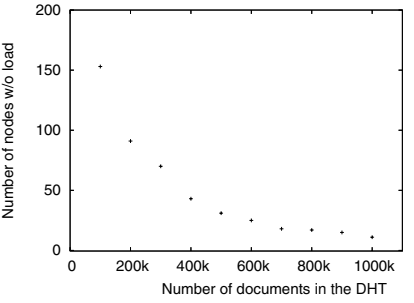
But, as proved in many papers [99, 501, 513], the simple assumption of getting an equally distributed value space simply by using hash functions does not hold. Therefore, several approaches for balancing the data load between DHT peers have been developed.

The following figure [513] shows simulations by simply hashing data into the target address space of a Chord ring. The distribution of documents among the nodes was analyzed. For each scenario, a simulated DHT with 4,096 nodes was subjected to multiple simulation runs. The total number of

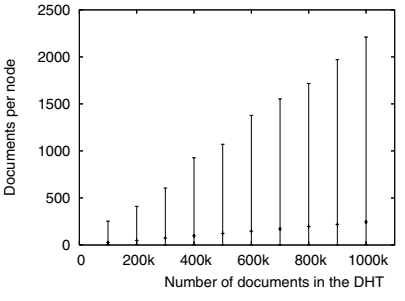
R. Steinmetz and K. Wehrle (Eds.): P2P Systems and Applications, LNCS 3485, pp. 119-135, 2005.



(a) Frequency distribution of DHT nodes storing a certain number of documents



(b) Number of nodes not storing any document



(c) Minimum, average, and maximum number of documents per node

Fig. 9.1: Distribution of data among a Chord DHT without load-balancing mechanisms.

documents to be stored ranged from 100,000 to 1,000,000 and for this purpose the Chord ring's address space had a size of $m = 22$ bits. Consequently, $2^{22} = 4{,}194{,}304$ documents and/or nodes could be stored and managed in the ring. The keys for the data and nodes were generated randomly. The load of a node was defined by the number of documents it stored.

The graphs in Fig. 9.1 clearly show that the assumption of an equal distribution of data among peers by simply using a hash function does not hold. For example, Fig. 9.1(a) shows how many nodes (y-axis) store a certain number of documents (x-axis). It is obvious that there is an unequal distribution of documents among the nodes. For an easier comparison, the grey line indicates the optimal number in the case of equal distribution – approximately 122 documents per node in this example. Additionally, Fig. 9.1(b) plots the number of nodes without a document.

Fig. 9.1(c) shows the distribution of documents in a Chord DHT without load-balancing. Between 100,000 and 1,000,000 documents were distributed across 4,096 nodes. The upper value indicates the maximum number of documents per node,and the lower value (always zero) the minimum number. The optimal number of documents per node is indicated by the marker in the middle. Even with a large total number of documents in the whole DHT, there are some nodes not managing any document and, consequently, are without any load. Some nodes manage a data load of up to ten times the average.

9.1.1 Definitions

Before discussing approaches for load-balancing in DHT systems, a clear definition of the term *load* has to be given [501]. Also, it must be made clear under what conditions a node may be referred to as *overloaded*, and a system as *optimally balanced*.

The data load of a peer node is usually determined by the amount of stored data per node. In the following, the load of a node is the sum of all data stored in this node. The total data load of a Peer-to-Peer (P2P) system is defined as the sum of the loads of all nodes participating in the system. The load of a system with N nodes is optimally balanced, if the data load of each node in the system is around $1/N$ of the total load. A node is referred to as *overloaded* or *heavy* if it has a significantly higher load compared with one in an optimal distribution. A node is *light* if it stores significantly less data than the optimum.

9.1.2 A Statistical Analysis

Before discussing algorithms for load-balancing, this section takes a look at the underlying statistics [450]. At the end of this section, theoretical evidence is provided for the empirical behavior. In the following, N is the number of nodes in the DHT, and m the number of data items.

Distribution of Data Items Among Peers with Equal-Sized Intervals
A simple model for load distribution is to consider all nodes to be responsible for intervals of equal size. Thus, when distributing a data item, each node is hit with a probability of $\frac{1}{N}$. Focusing on one node, the distribution of the data items is a series of independent Bernoulli trials with a success probability of $\frac{1}{N}$. The number of data items on this node (successful Bernoulli trials) is therefore following the binomial distribution.

The binomial distribution p_b and its standard deviation σ_b are defined as follows:

$$p_b(load == i) = \binom{m}{i}\left(\frac{1}{N}\right)^i\left(1-\frac{1}{N}\right)^{(m-i)} \tag{9.1}$$

$$\sigma_b = \sqrt{\frac{m}{N}\left(1-\frac{1}{N}\right)} \tag{9.2}$$

As an example, more than 300,000 collected file names from music and video servers were hashed, and the ID space was divided into intervals according to the first bits of an ID, e.g., 8 bits for 256 intervals. The load of each of these intervals is distributed closely around the average with the empirical standard deviation ($\sigma_{Experiment} = 34.5$) being close to the theoretical one ($\sigma_{Binomial} = 34.2$).

However, the assumptions of this model are not realistic for DHTs because interval sizes for nodes are not equal. The next section will deduce the interval size distribution.

Distribution of Peers in Chord

This section looks at the distribution of nodes on the Chord ring, or any other system randomly assigning node IDs. For the sake of simplicity, we use a continuous model, i.e., we consider the ID space to be real-valued in the interval $[0, 1)$. The number of nodes in a Peer-to-Peer network (at most, say a billion, i.e., roughly 2^{30} nodes) is small compared to the 2^{160} IDs in Chord's ID space.

– n-1 experiments with U(0,1).
– Determine the distribution of the IDs of the peers in the experiments.

The rationale for using this continuous model is that it is easier than a discrete one and the ID space is large compared with the number of nodes in it.

The continuous uniform distribution is defined as:

$$U(x) = \begin{cases} 0 & x < 0 \\ x & 0 \le x < 1 \\ 1 & x \ge 1 \end{cases}$$

Let L be the distribution of the interval size. It is given as the minimum of $N-1$ experiments[1]:

[1] For our statistical analysis it does not matter if the node responsible for the data is at the beginning or the end of the interval.

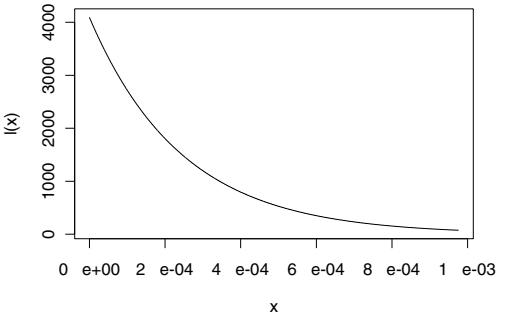


Fig. 9.2: Probability Density Function for Continuous Model with 4,096 nodes.

$$\begin{aligned} L(x) &= 1 - \prod_{i=1}^{N-1} (1 - U(x)) = 1 - (1 - U(x))^{N-1} \\ &= \begin{cases} 0 & x < 0 \\ 1 - (1-x)^{N-1} & 0 \le x < 1 \\ 1 & x \ge 1 \end{cases} \end{aligned}$$

And the probability density function (pdf) shown in Fig. 9.2:

$$l(x) = \frac{dL}{dx} = \begin{cases} (N-1)(1-x)^{N-2} & 0 \le x < 1 \\ 0 & else \end{cases}$$

It is interesting that the form of the probability density function and the load distribution are quite similar. This is even better illustrated when this calculation is done for a discrete ID space. To use this formula as an

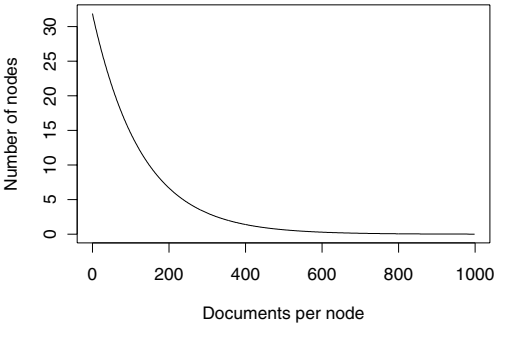


Fig. 9.3: Load Distribution (mean load = 128; 4,096 nodes) approximated with a scaled probability function from the Discrete Model.

approximation for the load, we use the number of data items as the ID space. Consequently, we get an approximation for the probability of a certain load (e.g., probability that a node has load = 10 items). If we multiply these probabilities with the number of nodes, we get the frequency distribution shown in Fig. 9.3.

9.1.3 Algorithms for Load Balancing in DHTs

To solve the problem of load-balancing, several techniques have been developed to ensure an equal data distribution across DHT nodes [512]. This chapter presents *Virtual Servers* [501], *Power of Two Choices* [99], *Heat Dispersion Algorithm* [513], and *a Simple Address-Space and Item Balancing* [338].

For clarity,the algorithms are mainly explained with Chord, but they are suitable for most of the different DHT systems.

The Concept of Virtual Servers

The virtual server approach [501] is based on the idea of managing multiple partitions of a DHT's address space in one node. Thus, one physical node may act as several independent logical nodes. Each virtual server will be considered by the underlying DHT as an independent node. Within a Chord system, one virtual server is responsible for an interval of the address space, whereas the corresponding physical node may be responsible for several different and independent intervals.

The basic advantage of this approach is the simplicity of displacement of virtual servers among arbitrary nodes. This operation is similar to the standard join or leave procedure of a DHT and content will be distributed as ranges of hash values across a the nodes. Every participating node manages virtual servers and has knowledge of all their neighbors from the fingers in the finger-table. For example with Chord, this relates to all fingers within the routing table. Now each node can transfer virtual servers to other nodes.

Transfer. The idea of the algorithm is to move a virtual server from a heavy to a light node. This transfer can be organized using three different schemes, known as: One-to-One, One-to-Many, and Many-to-Many.

In all of these schemes, the best virtual server which gets transferred is the one which satisfies the following three constraints. First, the transfer of a virtual server doesn't make the node which receives the virtual server heavy. Second, the virtual server is the lightest virtual server that makes the releasing node light. And third, if there is no virtual server whose transfer can make a node light, then the heaviest virtual server from this node gets transferred.

The third constraint results in the transfer of the largest virtual server that will not make the receiving node heavy, therefore, the chance of finding another light node in the next round which can receive a virtual server of this heavy node is increased.

One-to-One Scheme. This scheme is the simplest one. Two nodes are picked at random and a virtual server is transferred from a heavy node to a light one. Each light node periodically selects a node and indicates a transfer if that node is heavy, and if the above tree rules hold.

One-to-Many Scheme. This scheme allows a heavy node to consider more than one light node at a time. Each heavy node transfers a virtual server to one node of a known set of light nodes. For each light node of this set, the best virtual server is computed as described above and only the lightest virtual server of these will be transferred.

Many-to-Many Scheme. This scheme matches many heavy nodes to many light nodes. In order to get many heavy nodes and many light nodes to interact, a global pool of virtual servers is created – an intermediate step in moving a virtual server from a heavy node to a light node. The pool is only a local data structure used to compute the final allocation.

In three phases (unload, insert, and dislodge) the virtual servers to be transferred are computed. In the first one (unload) each heavy node puts the information about its virtual servers into a global pool until this node becomes light.

The virtual servers in the pool must then be transferred to nodes in the next step (insert). This phase is executed in rounds, in which the heaviest virtual server from the pool is selected and transferred to a light node, determined using the rules above. This phase continues until the pool becomes empty, or until no more virtual servers can be transferred.

In the final phase (dislodge), the largest virtual server from the pool is exchanged with another virtual server of a light node which is lighter and does not make the node heavy. If such a node is found, the insert step begins again, otherwise the algorithm terminates and the rest of the virtual servers in the pool stay at their current nodes.

Power of Two Choices

The algorithm *Power of Two Choices* [99] relies on the concept of multiple hash functions. These functions are used to map data into the address space of a DHT. For the processes of inserting and retrieving, the results of all hash functions are calculated. In the case of inserting a new document, all respective hash values are computed and the corresponding nodes are retrieved. Finally, the document is stored on the retrieved node with the lowest load in terms of stored data.

In formal terms, every node knows the universal hash function $h_1, h_2, \cdots, h_d$ which maps data onto the ring and so a node can compute

$h_1(x), h_2(x), \cdots, h_d(x)$ to insert the data x. For each of these computed results, the node responsible for this ID in the DHT is located. The data is now placed on the peer with the lowest load.

There are two ways to implement the search. A simple implementation requires that all hash functions be recalculated. After all lookups are made to find the peers associated with each of these values, one node must have successfully stored the data. These searches can be made in parallel and thus enable searching in little more time than their classic counterparts since this approach uses a factor of d more network traffic to perform each search.

The second way of searching is to use redirection pointers. Insertion proceeds exactly as before, but in addition to storing the item at the least loaded peer, all other peers store a redirection pointer to this node. To retrieve document x, it is not necessary to calculate all possible hash functions $h_1, h_2, \ldots, h_d$, because each possible node $h_1(x), h_2(x), \ldots, h_d(x)$ stores a pointer to document x. Thus, each of these nodes can forward the request directly to the node which is actually storing the requested document. Hence, a request for a certain key has to be made only to one of the d possible nodes. If this node does not store the data, the request is forwarded directly to the right node via the pointer. Nevertheless, the owner of a key has to insert the document periodically to prevent its removal after a timeout (soft state). Lookups now take at most only one more step.

Load Balancing Similar to Heat Dispersion

Rieche et. al [513] introduce another load-balancing algorithm for DHTs. Content is moved among peers similar to the process of heat dispersion [521]. Usually, a material warmer than its environment emits heat to its surroundings until a balanced distribution is reached in the entire system. To deploy a similar algorithm for balancing load among peers in a DHT, [513] proposes a very simple approach which needs only three rules. But nodes in a DHT can not simply move documents arbitrarily to other nodes, e.g., their neighbors, because this would result in an inconsistent and inefficient search. This reduces the performance and advantages of a DHT. Therefore, the algorithm moves only complete intervals, or contiguous parts of them, between the nodes in the DHT.

It seems appropriate to summarize the algorithm based on the DHT system Chord [575]. Although the Chord system has been modified, the efficient Chord routing algorithms remain unchanged. First of all, any fixed positive number f is chosen. f indicates the minimum number of nodes assigned to a specific DHT interval. If more than f nodes are assigned to a specific interval, one or more of them may be moved to a different interval. In case of more than $2f$ nodes, the respective interval can be split almost evenly. Now, a node has to manage approximately half of the documents. Each node periodically checks the data loads in its neighborhood – mainly its successors and

predecessors – as well as destinations referenced in its DHT routing table. This number f helps to balance the load, but also to make Chord more fault tolerant.

The first node takes a random position in the Chord ring and a new node is assigned to any other existing node in the system. This receiving node announces each joining node to all other nodes responsible for the same interval. Following this, a portion of the documents located within this interval are copied to the new node. Then, the original methods to insert a node in Chord are performed. Now the nodes, located within the same interval, can balance the load with nodes of other intervals according to one of three various methods.

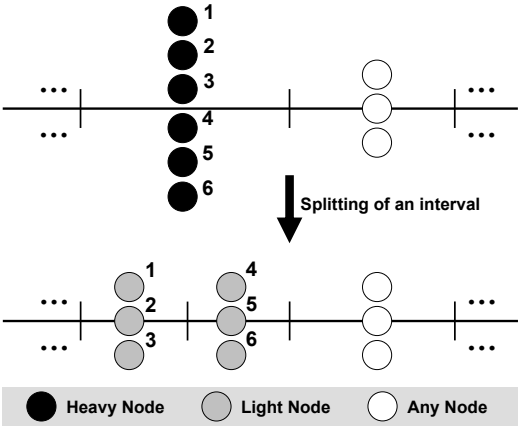


Fig. 9.4: Splitting of an Interval: Nodes 1 to 6 are assigned to the same interval and are overloaded in terms of data load. Since only three nodes are necessary to maintain an interval, this interval can be split.

$2f$ *Nodes with Excessive Load.* If $2f$ different nodes are assigned to the same interval, and each node stores significantly more documents than average, then this interval gets divided. The point of separation is the center of the interval. It can be easily computed as the half of the interval borders or the half of the hash values representing the stored documents. This implies that no load in terms of data has to be moved anywhere, and the respective nodes lose approximately half of their data load at once. Finally, the predecessors and successors will be adapted accordingly. Figure 9.4 shows an example of such an interval division.

More than f Nodes in an Interval. Intervals with more than f but less than $2f$ nodes can release some nodes to other intervals. If nodes within a particular interval are overloaded, they wait for additional nodes to join them. If

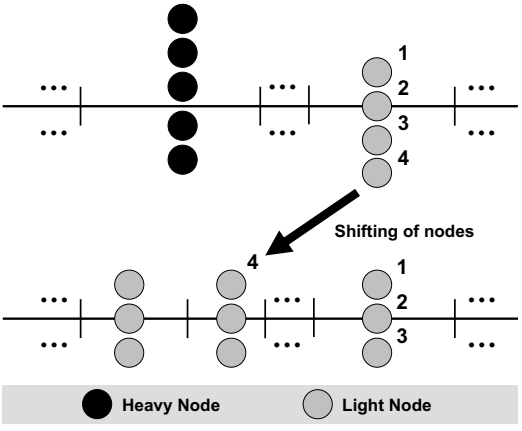


Fig. 9.5: Moving nodes: Nodes 1 to 4 are assigned to an interval. Since only three nodes are necessary to maintain an interval, node 4 can be transferred into another overloaded interval.

some nodes are very light, they periodically send this information to other nodes placed in different intervals. These desired destinations (intervals) can be found using routing entries of appropriate finger tables. Even if an interval with a heavy load exists, nodes can be moved to this interval. Based on the new situation, accumulated nodes within the new interval can try to split it by the rules described above.

Figure 9.5 shows an example of such a shifting of nodes to regions of higher load. Nodes 1 to 4 are very light in terms of data load and are responsible for the same interval. Since only three nodes are required, node 4 can be moved to an overloaded interval that should be divided.

No more than f Nodes within an Interval. As an additional alternative, interval borders may be shifted. Nodes can compare their load with the load of their immediate predecessors and successors. If its own interval shows more load than its neighbor's, part of the load can be released and thus interval borders will be shifted. Figure 9.6 shows an example of such a shifting of interval borders.

A Simple Address Space and Item Balancing

Karger and Ruhl introduce two protocols for load-balancing [338], especially for Chord [575]. The first balances the distribution of the key address space to nodes, the second directly balances the distribution of data among the nodes.

Address-Space Balancing. Each node has a fixed set of $O(\log N)$ possible positions in the Chord ring. These places are called virtual nodes (in com-

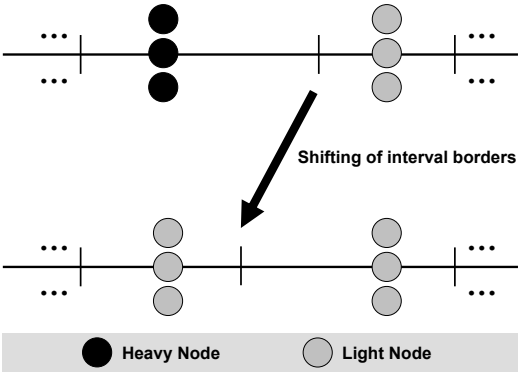


Fig. 9.6: Intervals adjusted between neighbors: The nodes within the right interval act together and are light, but the nodes located within the interval before are overloaded. Interval borders can be changed there.

parison to virtual servers in section 9.1.3) and are computed with different hash functions applied to their own ID. Each node chooses only one virtual node to become active. The address of a node is denoted as $(2b + 1)2^{-a}$ by $\langle a, b\rangle$, where a and b are non-negative integers and $b < 2^{a-1}$. This is an unambiguous notation for all addresses with finite binary representation. These addresses are ordered according to the length of their binary representation, so $\langle a, b\rangle < \langle a', b'\rangle$ if $a < a'$ or $(a = a'$ and $b < b')$.

Each node now chooses its ideal state. Given any set of active virtual nodes, each (possibly inactive) one spans a certain range of addresses between itself and the succeeding active virtual node. Each real node has activated the virtual node that spans the minimal possible (under the ordering just defined) address space. Thus, each node occasionally determines which of its virtual nodes spans the smallest address space and activates that particular virtual node.

Item Balancing. This also shifts interval borders. Nodes can compare their load with the loads of other nodes. If its own interval shows more load than its neighbor's, part of the load can be released and thus interval borders between two intervals will be shifted.

9.1.4 Comparison of Load-Balancing Approaches

To analyze load-balancing in a DHT, a complete Chord ring simulator was developed in [513] to investigate and to compare the load-balancing algorithms *Virtual Servers* [501], *Power of Two Choices* [99], and *Heat Dispersion Al-*

gorithm [513]. The focus was on the distribution of documents among the nodes.

Simulation Scenarios

In each scenario, a Chord DHT with 4,096 nodes was simulated and multiple simulations were run per scenario to confirm the results. The simulation in [513] shows that the results are comparable with the simulations presented in [99]. The total number of documents to be stored ranged from 100,000 to 1,000,000. The keys for the data and nodes were generated randomly. For this purpose, the Chord ring's address space had a size of $m = 22$ bits. Consequently, $2^{22} = 4{,}194{,}304$ documents and/or nodes could be stored and managed in the ring. In the simulation, the load of a node is defined as the number of documents it stores.

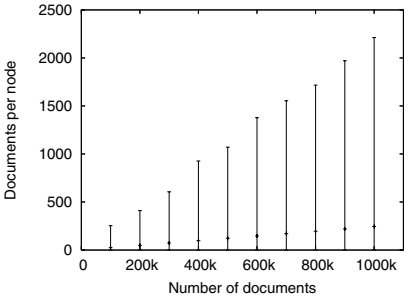
Simulation Results

Fig. 9.7(a) shows the distribution of documents in Chord without load-balancing. Between 10^5 and 10^6 documents were distributed across 4,096 nodes. The upper value indicates the maximum number of documents per node, the lower value the minimum number. The optimal number of documents per node is indicated by the marker in the middle.

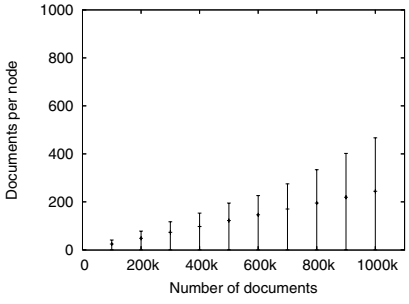
Even for a large number of documents in the DHT, there are some nodes not managing any documents and, consequently, without any load. Some nodes have a load of up to ten times above the optimum. Fig. 9.7(b) shows that *Power of Two Choices* works much more efficiently than the original Chord without load-balancing. However, there are still obvious differences in the loads of the nodes. Some are still without any document.

Applying the concept of *Virtual Servers* with the One-to-One scheme (cf. Fig. 9.7(c)) results in a more efficient load-balancing. Nevertheless, this is coupled with a much higher workload for each node because it has to manage many virtual servers. Additionally, the data of all virtual servers of one physical node has to be stored in the memory of the managing node.

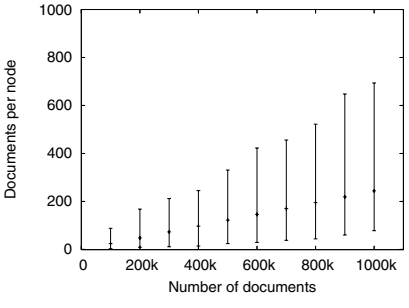
Fig. 9.7(d) shows that the best results for load-balancing are achieved by using the *heat dispersion algorithm*. Each node manages a certain amount of data and load fluctuations are relatively small. Documents are only moved from neighbor to neighbor. Using virtual severs, however, results in copying the data of a whole virtual server. As a result, the copied load is always balanced. In addition, more node management is necessary.



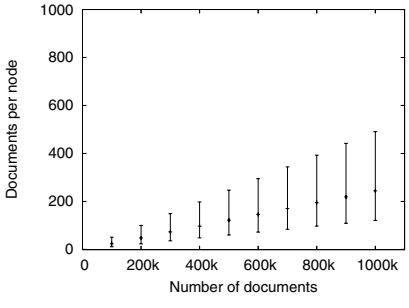
(a) Chord without load-balancing



(b) Chord with Power of Two Choices



(c) Chord with virtual server



(d) Chord with heat dispersion algorithm

Fig. 9.7: Simulation results comparing different approaches for load balancing in Chord.

9.2 Reliability of Data in Distributed Hash Tables

Through much research in the design and stabilization of DHT lookup services, these systems aim to provide a stable global addressing structure on top of a dynamic network of unreliable, constantly failing and arriving nodes. This will allow building fully decentralized services and distributed applications based on DHTs. This section shows algorithms for ensuring that data stored at failing nodes is available after stabilization routines of the Peer-to-Peer-based network have been applied.

There are two ways to store data in the DHT in a fault-tolerant manner. One is to replicate the data to other nodes, another is to split the data and make them more available through redundancy.

9.2.1 Redundancy

The idea to increase availability through redundancy is realized by splitting each data item into N fragments. Then K redundant fragments are computed by means of an erasure code. Thus, any N of the $N+K$ fragments will allow reconstruction of the original data. For each fragment, its place in the ring is computed. The data is split to $K+N$ different keys. Each fragment is stored using the standard Chord assignment rule. A read corresponds to $K+N$ recursive-style lookups and is successful if at least N parts are available. But, every time a node crashes, a piece of the data is destroyed, and after some time, the data may no longer be computable. Therefore, the idea of redundancy also needs replication of the data.

9.2.2 Replication

Another more fault-tolerant way to store the data is to replicate it to other nodes. This section describes two ways to replicate data in Chord.

Successor-List

The authors of Chord show in [575] a possibility to make the data more reliable in their DHT. The idea to make the data in Chord more fault-tolerant is to use a so-called *successor-list*. This list is also used to stabilize the network after nodes leave. The *successor-list* of any node consists of the f nearest successors clockwise on the Chord ring.

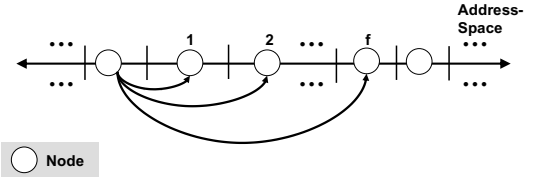


Fig. 9.8: Successor-list of a node with f nearest nodes clockwise on the Chord ring.

Because every node stores a successor-list of f nodes, the whole system has $n * f$ additional links in a network with n nodes. This implies that a lot of extra traffic is used just to keep the links consistent in case of node failures or arrivals.

Figure 9.8 shows an example for a node and its successor-list. A node stores pointers to the f nearest nodes clockwise on the Chord ring.

The reliability of data in Chord is an application task. Therefore, the successor-list mechanism helps a higher layer software to replicate inserted data to the next f nodes. Any application using Chord has to ensure that replicas of the data are stored at the f nodes succeeding the original node. Figure 9.9 shows replication of data by the application using the successive nodes in the ring.

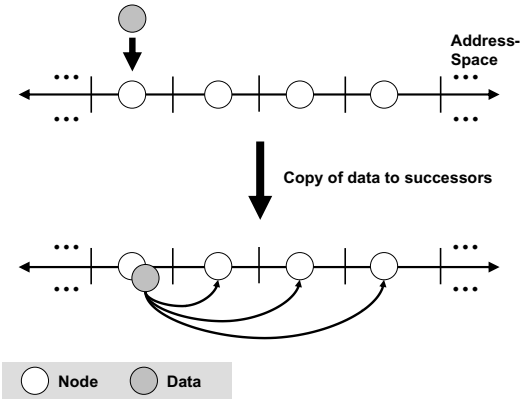


Fig. 9.9: Replication of inserted data in Chord by the application.

The application has to periodically check the number of replications of the inserted data and the stabilization routine has to repair the network successfully. Also, every node now has f intervals to store, hence the load of each node increases dramatically.

Multiple Nodes in One Interval

The approach [514] uses multiple nodes per interval.

Each interval of the DHT may be maintained by several nodes at the same time. In order to ensure correctness of this technique, each node stores additional pointers to nodes maintaining the same interval. In Chord, these additional pointers could be implemented as additional finger entries in another routing table. In CAN, they would be deployed as new neighbors. Let f be the minimum number of nodes assigned to a specific DHT region (or interval).

If a new node joins an existing interval, it announces itself to all other nodes responsible for the same interval. Additionally, all data associated with this interval is copied to the new node.

The first node takes a random position in the Chord ring and a new node is assigned to any existing node in the system. This receiving node announces

each joining node to all other nodes responsible for the same interval. Following this, parts of documents located within this interval are copied to this new node. Then, the original methods to insert a node in Chord are performed. To keep the complexity of the routing tables low, each node stores only one reference to the list of nodes for each finger within its own routing table to other intervals.

Figure 9.10 shows an example of the distribution of intervals on different nodes, where each interval has the minimum of two different nodes assigned to it.

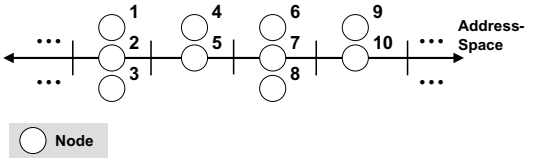


Fig. 9.10: Intervals with minimum of two nodes assigned to them.

According to [575], each node may maintain several virtual servers, but most of them store only one. This allows nodes with higher performance to store several virtual servers. Chord can take advantage of the high computational power of certain nodes. Such a node declares itself responsible for two or more intervals. Thus, it manages several virtual servers, and each virtual server is responsible for separate disjoint intervals. However, all intervals stored at the virtual servers in the same physical node shall not be identical in order to guarantee fault tolerance.

If new data has to be inserted into an interval, it will be distributed by one node to all other nodes responsible for the same interval. But, no copies of the data are replicated clockwise to the next n nodes along the ring, as in the original Chord DHT. Figure 9.11 shows the distribution of replicas of the inserted data to the neighbors responsible for the same interval.

If any node leaves the system and any other node takes notice of this, the standard stabilization routine of Chord is performed. The predecessors and successors are informed, and afterwards, inconsistent finger table entries are identified and updated using the periodic maintenance routine.

Thus DHT systems become more reliable and far more efficient due to the structured management of nodes. Generally, random losses of nodes are not critical because at least f nodes manage one interval cooperatively. The modified DHT system can cope with a loss of (f-1) nodes assigned to the same interval. In case of less than f nodes within one interval, the algorithm immediately merges adjacent intervals.

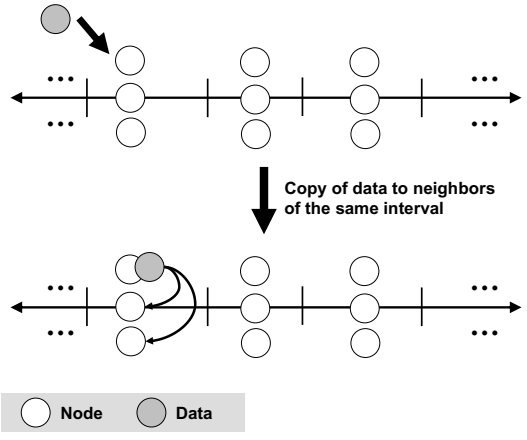


Fig. 9.11: Copy of data to the neighbors responsible for the same interval.

9.3 Summary

This chapter introduces algorithms for DHT-based systems which balance the storage data load or care for the reliability of the data.

All systems usually rely on the basic assumption that data is nearly equally distributed among the peer nodes. But, as shown, the simple assumption of getting an equally distributed value space simply by using hash functions does not hold. To achieve a continuous balance of data and, moreover, to ensure the scalability and efficiency in structured Peer-to-Peer systems, load-balancing mechanisms have to be applied. Therefore, this chapter introduces some algorithms for balancing storage load in DHTs. Although the algorithms are explained based on Chord, most of them can be easily adapted to other DHTs, such as CAN.

The second part of the chapter shows two ways to store data in the DHT in a fault-tolerant manner. One is to replicate the data to other nodes, and another is to split the data, thus making them more available through redundancy.