View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Biblioteca Digital do IPB

Toward a dynamically balanced cluster oriented DHT

José Rufino[1]* António Pina[2] Albano Alves[1] José Exposto[1]

[1]Polytechnic Institute of Bragança, 5300-854 Bragança, Portugal
{rufino,albano,exp}@ipb.pt
[2]University of Minho, 4710-057 Braga, Portugal
pina@di.uminho.pt

Abstract

In this paper, we present a model for a cluster oriented Distributed Hash Table (DHT). It introduces *software nodes*, *virtual nodes* and *partitions* as high level entities that, in conjunction with the definition of a certain number of invariants, provide for the balancement of a DHT across a set of heterogeneous cluster nodes. The model has the following major features: **a)** the share of the hash table handled by each cluster node is a function of its *enrollment level* in the DHT; **b)** the enrollment level of a cluster node in the DHT may change dynamically; **c)** cluster nodes are allowed to dynamically join or leave the DHT. A preliminary evaluation proved that the quality of the balancement of partitions of the hash table across the cluster, measured by the standard deviation with relation to the ideal average, surpass the one achieved by using another well known approach.

1 Introduction

Certain classes of applications often need to deal with very large amounts of data. As such, conventional, centralized data structures are no longer adequate, once a single machine may not be able to accommodate all the required data. One solution to this problem is to distribute the data across the nodes of a cluster. The distribution of data may provide for significant improvements in the data access throughput, derived from parallel and concurrent accesses to the data.

Complex, dynamical applications may also require the existence at the same time of several Distributed Data Structures (DDSs), each one dedicated to a specific domain of applicability. If we use a cluster to manage such complexity, we will finish having every node of a cluster running multiple applications, supported by several DDSs.

In this context, an issue often disregarded when developing cluster oriented DDSs is the possible heterogeneity of the cluster nodes. Another issue is related to the dynamism of applications running in a cluster, where nodes may be dedicated to several different tasks, with variable demands during its lifetime.

In this paper, we present a model for a cluster oriented Distributed Hash Table (DHT). The major features of our approach are: **a)** the share of the hash table handled by each cluster node depends on the node's capabilities and on the amount of resources bound to the DHT; **b)** each cluster node is allowed to dynamically change the amount of resources engaged in the DHT; **c)** cluster nodes may dynamically join or leave the DHT.

In what follows, section 2 characterizes the problem, section 3 introduces our model, section 4 describes the assignment of partitions to cluster nodes, section 5 presents a preliminary evaluation of the model, section 6 provides a comparison with related work, section 7 discusses some assumptions and limitations and section 8 concludes.

*Supported by PRODEP III, through the grant 5.3/N/199.006/00, and SAPIENS, through the grant 41739/CHS/2001.

2 Characterization of the problem

The dynamic balancement of a hash table across a set of cluster nodes is a complex problem, once it comprises many different and interrelated issues. As such, we chose to concentrate our efforts in a small (yet relevant) subset of those issues, and then establish a systematic pathway for the search of solutions. This view is conveyed by Table 1.

problem class	diversity of cluster nodes		usage policy of cluster nodes		access pattern and storage utilization	
	none	some	exclusive	shared	uniform	diverse
1	×		×		×	
2		×	×		×	
3	×		×			×
4		×	×			×
5	×			×	×	
6		×		×	×	
7	×			×		×
8		×		×		×

Table 1: Characterization of the DHT balancing problem.

Table 1 classifies our dynamic balancing problem accordingly with: **a)** the diversity of cluster nodes enrolled in the DHT; **b)** the usage policy of those cluster nodes; **c)** the access pattern to data and the storage utilization.

Typically, the diversity of nodes in a cluster is low: clusters are more often made of homogeneous nodes, which makes their administration easier. There are, however, valid reasons for a cluster to have some heterogeneity: **a)** economical factors may dictate the coexistence of ma-

chines from different hardware generations; **b)** some tasks may require more specialized nodes.

In a cluster, nodes may run a single user task at a time, or they may run several concurrent user tasks. To maximize the cluster usage ratio and minimize the response time of tasks, a shared usage policy may be desirable over an exclusive one. However, a shared policy requires more sophisticated mechanisms to globally balance the load in face of several user tasks, each one with specific requirements.

Finally, in a DHT, the access pattern to data records and the storage consumed by each record are not necessarily uniform: there may be situations where some data records are more popular; it is also possible that some data records consume much more/less storage space than the average. These situations may be transient or permanent. In the later case, imbalances may arise if such “anomalies” are not evenly spread.

In Table 1, each problem class requires a solution with a different complexity level. In general, higher level classes require more complex solutions. The model we present in this paper deals with both classes 1 and 2, and paves the way for the investigation of the remaining problem classes.

3 The basic model

In our model, a DHT is organized as a set of *software nodes* (or simply *snodes*), which are software instances that manage subsets of the hash table.

Although problem classes 1 and 2 prevent cluster nodes to support more than one DHT at a time, we have incorporated in the model some of the mechanisms that will be necessary to do so. Specifically, the naming convention for snodes is compatible with a cluster node hosting several snodes, each one relative to a different DHT: a snode is uniquely identified in the cluster by a 32 bit integer.

Figure 1 shows the internal organization of a snode.

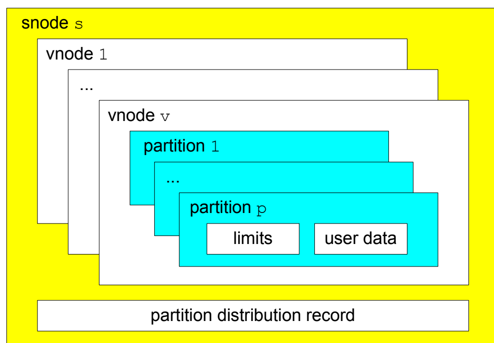


Figure 1: Anatomy of a typical snode.

3.1 Vnodes

Each snode may host several *virtual nodes* (or simply *vnodes*), which are sets of *partitions* of the DHT. The quantity of vnodes per snode may vary dynamically. This feature allows for the definition of the *enrollment level* of a snode as the number of vnodes it hosts at a specific time.

The idea behind the definition of vnodes comes from **a)** the need to distribute the hash table accordingly with the capabilities (CPU power, main memory, disk storage and network bandwidth) of each cluster node and **b)** the need to support the dynamic adjustment of the amount of resources that each cluster node dedicates to the DHT.

If nodes are homogeneous (both at the hardware and operating system level), we may define the enrollment level of each node in a DHT by the percentage/amount of resources bound to the DHT.

If nodes are heterogeneous, the definition of the enrollment level should be complemented by taking into account the relative capabilities of each node. One way to assess them is to execute a benchmark, on one node of each kind, before running the DHT. The benchmark tool selected should be oriented to the way in which data records are going to be locally stored (*e.g.*, in main memory and/or in the file system, in a database, etc.).

Every vnode is a local snode entity, named by an integer which corresponds to the sequence order of its creation (*e.g.*, vnode `v-1` was created before vnode `v`). In the global context of the cluster, in order to identify a specific vnode, we use a canonical/fully qualified name, that follows the format `snode_id.vnode_id`. With respect to Figure 1, the canonical name of vnode `v` is `s.v`.

3.2 Partitions

A *partition* is a contiguous subset of the range of the hash function, with specific left and right *limits*. There may be *user data* bound to each partition: user data with a specific key is bound to a partition if by feeding the hash function with the key the outcome is an hash index that belongs to the partition.

A partition is named, in the local context of its vnode, by an integer which corresponds to the order in which the partition was created (*e.g.*, partition `p-1` was created before partition `p`). In the global context of the cluster, a partition is named using a canonical/fully qualified name, following the generic format `snode_id.vnode_id.partition_id`. In Figure 1, `s.v.p` is the canonical name of partition `p`.

3.3 Partition Distribution Record

Every snode hosts a copy of a global *partition distribution record* (PDR) which is a table used to dynamically registry the number of partitions of each one of the vnodes created across all the nodes of the cluster enrolled in a specific DHT. Every entrance of the table follows the format `(snode_id.vnode_id, nparts)`.

The PDR table is used during the creation/deletion of vnodes, to assist on the process of deciding which vnodes

will loose/gain partitions, with the objective of keeping the hash table balanced across all the vnodes.

3.4 Invariants

Let h be a hash function of range $R_h = \{i \in \mathbb{N}_0 : 0 \leq i < 2^{B_h}\}$, where B_h is the (fixed) number of bits of any hash index i. Also, let P_{min} be a (fixed) power of 2. In our model, the following invariants must hold:

1. R_h is fully divided in mutually exclusive (non–overlapping) partitions;
2. the overall number of partitions, P, is always a power of 2;
3. every partition has the same size $S = 2^{B_h}/P$;
4. for any vnode v, its number of partitions, P_v, is bounded: $P_{min} \leq P_v \leq P_{max}$ and $P_{max} = 2.P_{min}$;
5. when the overall number of vnodes, V, is a power of 2, any vnode will have P_{min} partitions.

4 Assignment of partitions to vnodes

The main objective of the model is to perfectly balance the distribution of the partitions of a DHT, that conforms to the defined invariants, across all the nodes of the cluster enrolled in the DHT. As so, it provides for **a)** *coarse–grain balancing*, by letting snodes increase or decrease its number of vnodes, and for **b)** *fine–grain balancing*, by letting the number of partitions bound to each vnode to fluctuate. This approach is effective only during the creation or deletion of vnodes.

The creation/deletion of vnodes are synchronization points (or barriers) that have to be supported by a synchronization protocol that minimizes the underlying communication overhead. This paper does not address that issue.

Also, due to space constraints we only present the mechanism for the creation of vnodes. Details about the deletion of vnodes may be found in [1].

4.1 Quality of the assignment

We use $\sigma(P_v, \overline{P}_v)$, the standard deviation of all values of P_v from the (ideal) average $\overline{P}_v$, to measure the quality of the assignment of partitions to vnodes.

By taking this approach, the assignment of partitions to vnodes should minimize $\sigma(P_v, \overline{P}_v)$. To achieve this objective, it is necessary to carefully select the vnodes that will handover partitions (and in what number) whenever a new vnode is created, or which vnodes will receive partitions (and in what number) from a deleted vnode.

4.2 Creation of a single vnode

A snode starts the creation of a vnode by issuing a *creation request* to the totality of the snodes of the DHT. The request will be completed only when a consensus about the contents of the PDR in all snodes is attained and all the necessary transfer of partitions have been performed.

4.2.1 Reassignment of partitions

There is a common algorithm executed by all snodes to handle the creation of a new vnode that acts as follows.

First, a new pair `(snode_id.vnode_id, nparts)` is inserted in the (local) PDR table; the 1st field identifies the new vnode using its canonical name and the `nparts` field is initially set to zero. Second, the value of $\sigma(P_v, \overline{P}_v)$ is computed. Third, the entrances of the PDR table are sorted by the `nparts` field to find the vnode (the *victim vnode*) having the greater number of partitions. Fourth, when the value of $\sigma(P_v, \overline{P}_v)$ may decrease by moving one partition from the victim vnode to the new vnode, a *victim partition* belonging to the victim vnode is chosen and its transfer to the new vnode is scheduled. The third and fourth steps are repeated until $\sigma(P_v, \overline{P}_v)$ no longer decreases by removing partitions from the victim vnode.

At the third step, if more than one vnode may be elected as a victim, the correspondent entries in the PDR are resorted using the `snode_id` component of the vnode identifiers, to find the one that has the minimum value for `snode_id`. Afterward, if several entries having the same minimum value for `snode_id` still exist, those entries are further reordered by the `vnode_id` component of the vnode identifiers. The victim vnode will then be the one with the minimum value for `vnode_id`.

For instance, after the third step, the original PDR in Table 2.a) will result on the PDR presented in Table 2.b), leading to the selection of `a.1` as the victim vnode.

snode_id.vnode_id	nparts
a.1	12
a.2	8
b.1	10
a.3	12
b.2	10
b.3	12

a) PDR unsorted

snode_id.vnode_id	nparts
a.1	12
a.3	12
b.3	12
b.1	10
b.2	10
a.2	8

b) PDR sorted

Table 2: The same PDR, a) unsorted and then b) sorted.

To select the victim partition, the convention is to choose the one (pertaining to the victim vnode partition set) having the greater value currently used in the `partition_id` component of its canonical name. That value belongs to the range $\{P_{min} + 1, ..., P_{max}\}$. The reason for this resides on the need to conform to invariant 4, which states that a vnode must have at least P_{min} partitions; one way to ensure that the invariant is respected is to handover partitions only outside the range $\{P_1, ..., P_{min}\}$.

For instance, if the vnode `a.1` has partitions `a.1.1`, `a.1.2`, `a.1.3` and `a.1.4` (and $P_{min} = 2$), it will first handover `a.1.4` and subsequently `a.1.3`.

The identification of the victim partition will change to reflect its new snode and vnode hosts, as well as the creation order relative to the other partitions of the new vnode.

For instance, if `a.2.3` becomes the 6th partition of vnode `b.5`, the partition would be renamed to `b.5.6`.

The reassignment algorithm is deterministic and is executed by all snodes enrolled in a DHT. Thus, it is possible to identify, at each snode, **a)** the vnodes that will loose partitions, **b)** the partitions that will be transfered and **c)** its future name in the newly created vnodes.

4.3 Creation of several vnodes

As a result of the consecutive creation of vnodes, the number of partitions contained at each vnode evolves following a pattern imposed by the invariants of the model and by the use of the reassignment algorithm, previously presented.

To conform to invariant 5, when V (the overall number of vnodes) is a power of 2, all vnodes must have P_{min} partitions, ensuring that R_h is perfectly balanced across all the vnodes. In this situation, when a new vnode is created, some of the older vnodes will have to handover some of its partitions, accordingly to the assignment algorithm. Because invariant 4 does not allow $P_v < P_{min}$, all the older vnodes need to binary split their own partitions, doubling its number to $P_v = P_{max}$, which is the maximum number of partitions per vnode allowed by the invariant.

As more vnodes are created, the number of partitions of the existing vnodes will decrease toward P_{min}. At some moment, the overall number of vnodes, V, will double, reaching the next power of 2. At that point, for any vnode v, the number of partitions will be exactly $P_v = P_{min}$ and R_h will be again perfectly balanced across all vnodes.

4.3.1 Example

Table 3 shows the evolution of the number of partitions of each vnode, and which partition transfers take place, in a scenario where 4 vnodes are consecutively created, with $P_{min} = 4$. We choose to locate all vnodes in the same snode and so it suffices to name vnodes only by the `vnode_id` component of the canonical name. The values of P_v enclosed in a circle relate to victim vnodes. Partition transfers are denoted by `old_canonical_name` → `new_canonical_name`.

When vnode `1` is created, $V = 1$ (which is a power of 2), and so $P = P_1 = P_{min} = 4$ equal sized, non-overlapping, partitions of R_h (`1.1`, ..., `1.4`); the PDR is initially set to {`(1,4)`}.

When vnode `2` is created, vnode `1` must hand it over some partitions; first, however, invariant 4 mandates that vnode `1` binary splits its partitions, after which $P_1 = P_{max} = 8$ partitions (`1.1`, ..., `1.8`); then, the PDR is updated to {`(1,8);(2,0)`} and the assignment algorithm is applied, making vnode `1` to transfer 4 partitions to vnode `2` (`1.8` → `2.1`, ..., `1.5` → `2.4`); the choice of the victim partitions and their renaming follow the conventions previously established; the PDR becomes {`(1,4);(2,4)`}.

creation event	P_1	P_2	P_3	P_4	partition transfer
vnode 1	4				
vnode 2	8	0			
	⑦	1			1.8 → 2.1
	⑥	2			1.7 → 2.2
	⑤	3			1.6 → 2.3
	④	4			1.5 → 2.4
vnode 3	8	8	0		
	⑦	8	1		1.8 → 3.1
	7	⑦	2		2.8 → 3.2
	⑥	7	3		1.7 → 3.3
	6	⑥	4		2.7 → 3.4
	⑤	6	5		1.6 → 3.5
vnode 4	5	6	5	0	
	5	⑤	5	1	2.6 → 4.1
	④	5	5	2	1.5 → 4.2
	4	④	5	3	2.5 → 4.3
	4	4	④	4	3.5 → 4.4

Table 3: Assignment of partitions when creating 4 vnodes.

When vnode `3` is created, it will receive partitions from vnode `1` and/or vnode `2`; because $P_1 = P_2 = P_{min} = 4$, invariant 4 mandates that both vnodes `1` and `2` binary split their partitions, after which $P_1 = P_2 = P_{max} = 8$ partitions (`1.1`, ..., `1.8`, `2.1`, ..., `2.8`); then, the PDR is updated to {`(1,8);(2,8);(3,0)`} and the reassignment of partitions takes place as shown (`1.8` → `3.1`, ..., `1.6` → `3.5`); the PDR becomes {`(1,5);(2,6);(3,5)`}.

Finally, when vnode `4` is created, there are enough partitions for it, among the other vnodes, without the need for any partition split; the PDR is updated to {`(1,5);(2,6);(3,5);(4,0)`} and partitions are reassigned (`2.6` → `4.1`, ..., `3.5` → `4.4`); in the end, the PDR becomes {`(1,4);(2,4);(3,4);(4,4)`}; once $V = 4$ (a power of 2), a new point of perfect equilibrium (where $P_v = P_{min}$, for any vnode v) is reached.

5 Evaluation

We now present results from a preliminary evaluation of the algorithm used by our model to assign partitions to vnodes, under the following conditions: **a)** the overall number of vnodes, V, grows from 1 to 1024; **b)** there is only one vnode per snode; **c)** the minimum number of partitions per vnode, P_{min}, is set to 32. We present two graphics per each parameter evaluated. The second graphic zooms the first and the zoom level varies as convenient.

5.1 Average number of partitions per vnode

Figure 2 shows the evolution of the average number of partitions per vnode, $\overline{P}_v = P/V$, where P is the overall number of partitions and V is the overall number of vnodes.

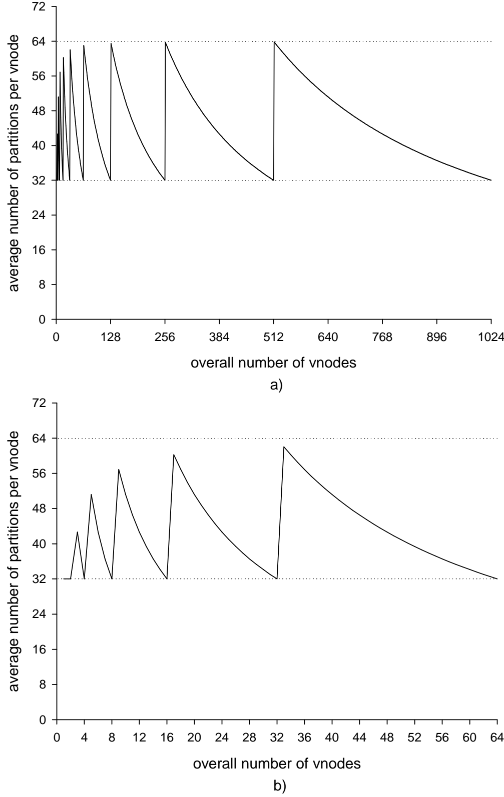


Figure 2: Evolution of $\overline{P}_v$

As a consequence of invariant 4, $P_{min} \leq \overline{P}_v \leq P_{max}$. To understand the way in which $\overline{P}_v$ evolves, we recall that P doubles every time V grows and the previous value of V was a power of 2 (only this way it is possible to ensure a minimum of P_{min} partitions per vnode, in accordance with invariant 4).

When $V = 1$ (a power of 2), $P = P_{min} = 32$ and $\overline{P}_v = 32/1 = 32$; then, when V increases to 2, we leave a power of 2 behind and so P doubles to 64, meaning that $\overline{P}_v = 64/2 = 32$; when V increases to $V = 3$, we also leave a power of 2 behind and so P doubles to 128, meaning that $\overline{P}_v = 128/3 = 42.6(6)$; when V increases to $V = 4$, we do not leave a power of 2 behind and so P remains 128, meaning that $\overline{P}_v = 128/4 = 32$; when V increases to $V = 5$, we leave a power of 2 behind and so P doubles to 256, meaning that $\overline{P}_v = 256/5 = 51.2$.

The evolution pattern above repeats, making the maximum value of $\overline{P}_v$ to converge toward P_{max}. To understand why this happens, suppose V' is a power of 2; the creation of one more vnode makes all previous vnodes to double their number of partitions to P_{max}; as such, $\overline{P}_v = P/V = (V'.P_{max})/(V'+1)$, which turns closer to P_{max} as V' increases.

5.2 Quality of the partition assignment

The quality of the partition assignment is measured by the standard deviation $\sigma(P_v, \overline{P}_v)$. This value depends not only on the actual number of vnodes, V, but also on the minimum number of partitions per vnode, P_{min}, constant for the lifetime of the DHT.

In order to also assess the influence of P_{min} on the quality of the partition assignment, we made P_{min} assume several values and computed $\sigma(P_v, \overline{P}_v)$ accordingly. However, we choose to plot (in percentage) the *relative* standard deviation, $\overline{\sigma}(P_v, \overline{P}_v) = \sigma(P_v, \overline{P}_v)/\overline{P}_v$, once this metric makes comparisons easier.

We expected the quality of the partition distribution to improve, by increasing P_{min} (always trough powers of 2): higher values for P_{min} translate in more, smaller partitions per vnode; this allows for a more fine grained partition distribution, thus with smaller deviations of each P_v from the ideal average $\overline{P}_v$. The reverse should also hold: if P_{min} decreases, there will be less, bigger partitions and so deviations from the average should grow. The predicted behavior may be observed in Figure 3, where P_{min} varies through the values $\{16, 32, 64\}$.

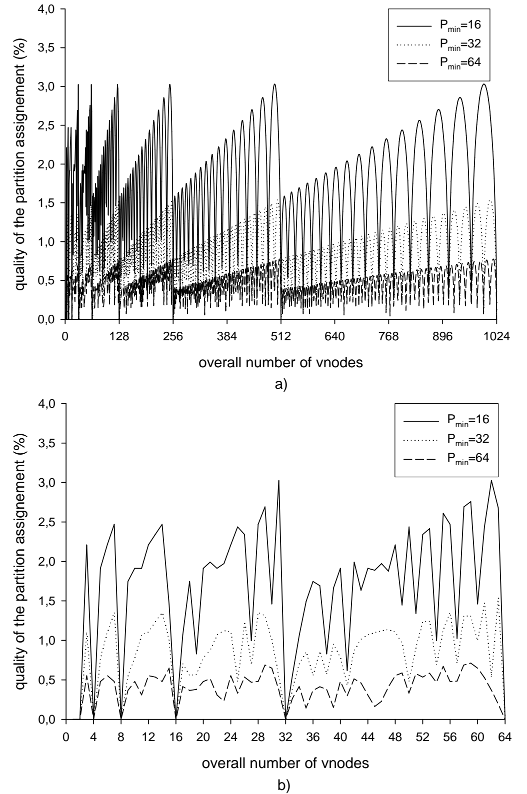


Figure 3: Evolution of $\overline{\sigma}(P_v, \overline{P}_v)$.

More precisely: doubling P_{min} makes the maximum value of $\overline{\sigma}(P_v, \overline{P}_v)$ to halve; halving P_{min} makes the maximum value of $\overline{\sigma}(P_v, \overline{P}_v)$ to double. With regard to each curve, the minimum is reached whenever V is a power of 2, meaning each node has exactly P_{min} partitions; adding more vnodes, until V doubles, makes $\overline{\sigma}(P_v, \overline{P}_v)$ to grow, following an oscillatory pattern, which becomes more regular with more nodes.

5.3 Number of victim vnodes

Figure 4 shows the evolution of V_t, the number of victim vnodes (vnodes that will transfer at least one partition to a new vnode).

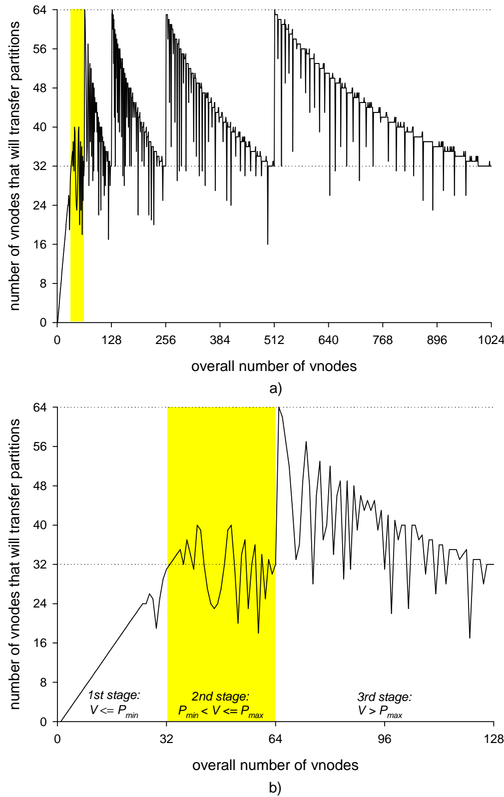


Figure 4: Evolution of V_t

It may be observed that the evolution of V_t has 3 different stages. In the 1st stage, $V \leq P_{min}$, meaning there are less vnodes than the minimum number of partitions per vnode, and so every vnode will contribute with partitions to a new vnode; in general, in this stage $V_t \approx V$, except for a small number of cases.

In the 2nd stage, $P_{min} < V \leq P_{max}$; once $V > P_{min}$, only a subset of vnodes will loose partitions to a new vnode; V_t oscillates around an average of $31.25 \approx P_{min}$.

The 3rd stage begins when $V > P_{max}$; from there on, V_t will decrease from a maximum value, near P_{max}, to a minimum value, near P_{min}; this variation is cyclic: it begins after V becomes the successor of a power of 2 and ends when V reaches the next power of 2.

We thus conclude that V_t scales linear with V when $V \leq P_{min}$, and V_t remains approximately bounded between P_{max} and P_{min} when $V > P_{min}$. Thus, we may say that in the 1st stage V_t is of order $O(V)$ with relation to V and that in the remaining scenarios V_t is of order $O(1)$ with relation to V.

Figure 5 presents a complementary perspective about the evolution of V_t, by plotting V_t/V (in percentage), which is the fraction of the number of vnodes that will transfer partitions, from the overall number of vnodes. As expected, V_t/V diminishes exponentially, with an approximate minimum bound of P_{min}/V and an approximate maximum bound of P_{max}/V.

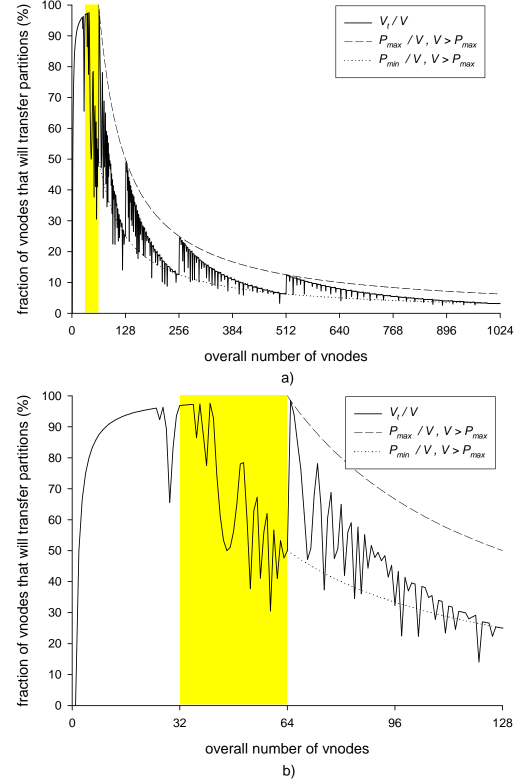


Figure 5: Evolution of V_t/V

5.4 Average number of partitions transfered

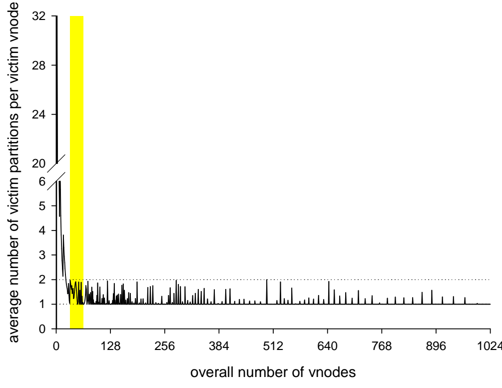
Figure 6 plots $\overline{P}_t$, the average number of partitions transfered by each victim vnode to a new vnode.

The evolution of $\overline{P}_t$ goes trough the same 3 different stages of the evolution of V_t. In the 1st stage ($V \leq P_{min}$), $\overline{P}_t$ has a maximum value of P_{min} that takes place with $V = 2$: when the 2nd vnode is created, the 1st vnode binary splits its partitions, doubling its number from $P_{min} = 32$ to $P_{max} = 64$; it then transfers $\overline{P}_t = 32$ partitions to the 2nd vnode and both vnodes end up with $P_{min} = 32$ partitions. Then, $\overline{P}_t$ decreases toward a minimum value of 1,03 when the 32nd vnode joins the DHT (and, accordingly with Figure 4.a), $V_t(32) = 31$ vnodes transfer at least one partition to the new vnode).

In the 2nd stage ($P_{min} < V \leq P_{max}$), $\overline{P}_t$ oscillates between a minimum of 1 and a maximum near 2; once $V \leq P_{max}$, many vnodes must transfer more than one partition.

In the 3rd stage ($V > P_{max}$), we recall that V_t is typically greater than in the previous stages and so the victim vnodes seldom need to transfer more than one partition.

We finally conclude that $\overline{P}_t$ quickly stabilizes around a maximum of 2 and a minimum of 1 and may be referred to as being of order $O(1)$ with relation to V.



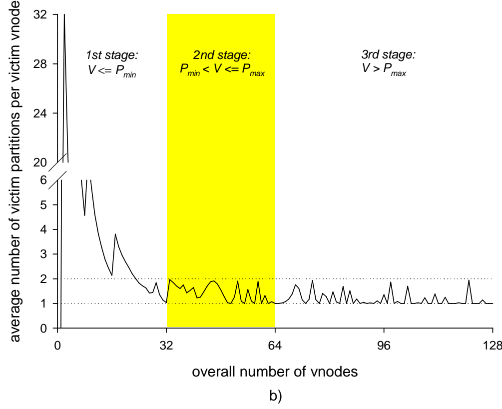
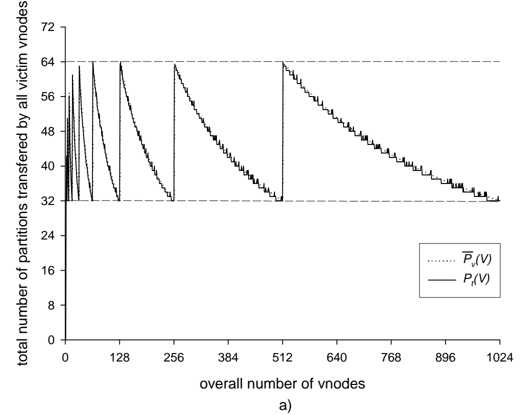


Figure 6: Evolution of $\overline{P}_t$



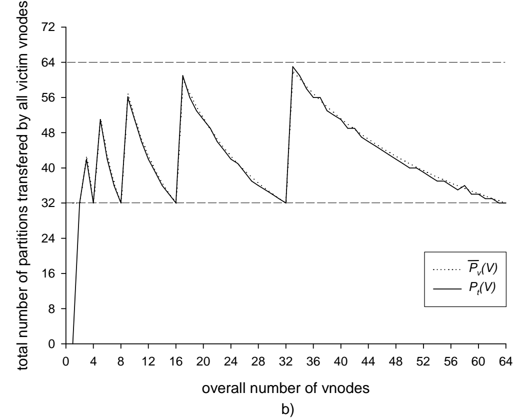


Figure 7: Evolution of P_t.

5.5 Total number of partitions transfered

Figure 7 shows $P_t = V_t.\overline{P}_t$, the total number of partitions transfered by all victim vnodes to a new vnode.

As expected, $P_{min} < P_t < P_{max}$ (for $V > 1$). Thus, P_t evolves with order $O(1)$, with relation to V. It may also be observed that P_t evolves close to $\overline{P}_v$, in accordance with the small values of $\overline{\sigma}(P_v, \overline{P}_v)$ presented in Figure 3 for $P_{min} = 32$.

6 Related Work

In this section we demonstrate the potential of our model by comparing its balancing capabilities with those of the well known Consistent Hashing (CH) approach. This approach was used both in the context of Web Caching [2] and in the context of Chord [3], a design for a P2P [4] lookup scheme.

In CH, the hash table is divided into partitions and each (computational) node hosts a certain number of virtual nodes. The hash table is distributed across the nodes by assigning one partition per virtual node. For the balancing to be effective, each node must host at least $k.log_2 N$ virtual nodes, if the expected maximum number of nodes is N [5].

CH uses partitions that have random sizes and virtual nodes that doesn't fit our own definition. For these reasons we cannot measure the quality of its partition assignment using the $\sigma(P_v, \overline{P}_v)$ metric. Thus, we introduce Q_n, the quota of R_h handled by each node n, calculated by dividing the sum of the ranges of all partitions hosted at that node, by 2^{B_h}. We may then use $\sigma(Q_n, \overline{Q}_n)$, which is the standard deviation of Q_n with relation to the (ideal) average $\overline{Q}_n$, to compare the balancement of a DHT when using the two different approaches.

In general, if X_i and Y_i represent two series of numbers, such that $Y_i = c.X_i$, for any i, with c constant, then $\sigma(Y_i, \overline{Y}_i) = c.\sigma(X_i, \overline{X}_i)$ and $\overline{\sigma}(Y_i, \overline{Y}_i) = \overline{\sigma}(X_i, \overline{X}_i)$. In our model, if we consider the special case where there is a sole virtual node per cluster node n, then $Q_n = (P_n.S)/2^{B_H} = c.P_n$. In this context, it is possible to reuse the values shown by Figure 3 to compare our model with CH, because $\overline{\sigma}(Q_n, \overline{Q}_n) = \overline{\sigma}(P_v, \overline{P}_v)$.

In Figure 8 we present the results of the comparison by showing the evolution of $\overline{\sigma}(Q_n, \overline{Q}_n)$ for the two approaches as the number of nodes enrolled in the DHT grows from 1 to 1024. To deal with the random nature of its partitioning policy, the values for CH reflect an average of 100 simulations. Because the number of partitions per node is fixed in CH, but varies in our model, from $P_{min} = 32$ to $P_{max} = 64$, we have considered in CH both the scenarios for 32 and 64 partitions per node.

The analysis of Figure 8 confirms a significant improvement of the quality of the distribution of our model when compared with CH. We further may conclude that the quality of the distribution on CH increases when the number of partitions per node also increases. The same conclusion applies to our model, as it was already observed

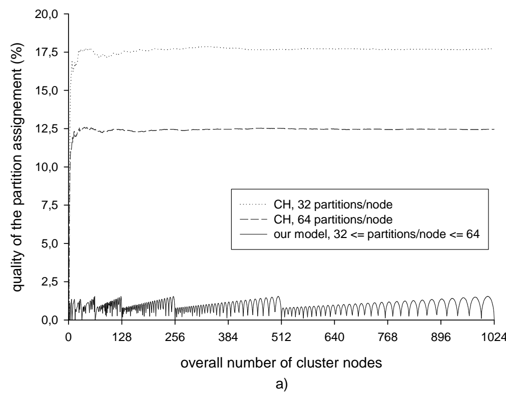


Figure 8: Evolution of $\overline{\sigma}(Q_n, \overline{Q}_n)$.

in Figure 3.

7 Discussion

We now discuss some of the assumptions and constraints of the model, along with issues that remain to be investigated.

The first assumption is that the hash function h uniformly maps its input in its range, restricting the choice of the hash function to the one that best suits the data type and size of the input keys (see [6, 7] for some related work).

The model also assumes that, although commodity clusters are not immune to failures, that context of utilization is stable, in contrast with other application scenarios for DHTs (*e.g.*, P2P environments). This gives good reasons for not incorporate in the model fault tolerance mechanisms.

The model contemplates two levels of balancing: the first, supported by vnodes, provides for coarse–grain balancing; the other one, based on the assignment of partitions to vnodes, provides for fine–grain balancing. We have only discussed the mechanisms used for the (re)assignment of partitions, that is, to achieve fine–grain balancing. Thus, it remains to be investigated the algorithms and metrics to assist on the assignment of vnodes to snodes.

To ensure that the PDR state is globally synchronized, the creation and the deletion of vnodes need to be global serialized events, thus limiting the degree of dynamism. This problem calls for the investigation of alternatives for distributed balancing (which may result on the diminishing of the quality of the assignment of partitions to vnodes).

No lookup method is specified, in order to locate the partition where a hash index belongs. However, to avoid the typical problems derived from centralized approaches, a distributed lookup scheme seems to be the most appropriate. When taking this approach, each snode should maintain a bounded amount of information about the location of partitions and the lookup process should need to visit a bounded number of snodes.

There are several alternatives for distributed lookup, including the ones offered by P2P oriented DHTs [8]. The same overlay network assumed by those approaches could be used to define neighborhoods for distributed balancing.

8 Conclusions

The main contribution of this paper is the definition of a model that paves the way for dynamically balanced, cluster oriented Distributed Hash Tables.

A preliminary set of tests allowed to study the evolution of some parameters and to demonstrate that the model responds as expected. Experience also proved that the balancing mechanism produces results that make the model competitive with another well known design for a DHT.

Finally, we point out some of the important issues that remain to be investigated and which will be the focus of our future work: **a)** mechanisms for coarse–level balancing; **b)** a scalable distributed scheme for the balancing of the hash table; **c)** a compatible scalable lookup scheme; **d)** solutions for the other problem classes, above class 2.

References

[1] J. Rufino, A. Pina, A. Alves, and J. Exposto. A cluster oriented model for dynamically balanced DHTs. Technical report, Dep. of Informatics and Communications, Polytechnic Institute of Bragança, Portugal, 2003.

[2] D. Kargeer, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web Caching with Consistent Hashing. In *Proceedings of the 8th International WWW Conference*, 1999.

[3] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balkrishnan. Chord: A Scalable Peer–to–Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM'01*, 2001.

[4] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP Labs, 2002.

[5] D. Kargeer, E. Lehman, F. Leighton, D. Levine, and R. Panigrahy. Consistent Hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 1997.

[6] B.J. McKenzie, R. Harries, and T. Bell. Selecting a Hashing Algorithm. *Software - Practice and Experience*, 2(20):209–224, 1990.

[7] J. Rufino. Selection of Hash Functions. Technical report, Dep. of Informatics and Communications, Polytechnic Institute of Bragança, Portugal, 2002.

[8] H. Balakrishnan, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, 46(2):43–48, 2003.