Multiple Order-Preserving Hash Functions for Load Balancing in P2P Networks

Maeva Antoine, Fabrice Huet

Multiple Order-Preserving Hash Functions for Load Balancing in P2P Networks

Maeva Antoine* and Fabrice Huet

University of Nice Sophia Antipolis, CNRS I3S UMR 7271
2000 route des Lucioles, Batiment Euclide A
06900 Sophia Antipolis - France
E-mail: ma.antoine06@gmail.com
E-mail: fabrice.huet@unice.fr
*Corresponding author

Abstract: Hash functions are at the heart of data insertion and retrieval in DHT-based overlays. However, a standard hash function destroys the natural ordering of data. To perform efficient range queries processing, in a minimum number of hops, more and more systems opt for an order-preserving hash function to place data. Unlike a standard hash function, this technique cannot evenly distribute data among peers. In this paper, we propose a novel approach to improve the data dissemination using several order-preserving hash functions. We describe a protocol that allows an overloaded peer to change its hash function at run-time. Then, we show that all peers should not necessarily use the same hash function in an overlay to insert or look for an item. Finally, we demonstrate through simulations that this strategy greatly improves the dissemination of data items. As this technique does not require any overall coordination, it is well suited to overloaded P2P systems facing unstable topologies. To our knowledge, this is the first load balancing solution using multiple order-preserving hash functions in structured overlays.

Keywords: hash functions; load balancing; structured overlays; skewed data; lexicographic order.

Reference to this paper should be made as follows: xxxx (xxxx) 'xxxx', *xxxx*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Maeva Antoine received her PhD in Computer Science from the University of Nice Sophia-Antipolis, France in 2015. She received a MSc in Databases and a MSc in Computer Science applied to Business Management from the University of Nice Sophia-Antipolis in 2012. She is currently working as a research project manager in the private sector. Her research interests include distributed databases for Big Data management, data mining and the Semantic Web.

Fabrice Huet is an associate professor at the University of Nice Sophia-Antipolis, France. He is a member of SCALE, a joint research group between INRIA, I3S and CNRS. He received his PhD in Computer Science from the University of Nice Sophia-Antipolis in 2002. His research interest lies in middleware for large scale distributed systems. He currently works on peer-to-peer overlays, Map Reduce and data stream analysis.

This paper is a revised and expanded version of a paper entitled 'Dealing with Skewed Data in Structured Overlays using Variable Hash Functions' presented

at The 15th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), Hong Kong, 9-11 December 2014.

1 Introduction

Many Peer-to-Peer (P2P) systems like Chord (Stoica et al. 2001), Content Addressable Network (CAN) (Ratnasamy et al. 2001), Pastry (Rowstron & Druschel 2001) or Kademlia (Maymounkov & Mazieres 2002) face unstable topologies (Dhurandher et al. 2009), making it difficult to insert and locate data items in the network. In such systems, each node only has a partial knowledge of the other peers located in the overlay. To improve file search in P2P systems, several techniques have been proposed in the literature (Dhurandher et al. (2011), Navimipour & Milani (2015)). Hashing is commonly used in structured overlays to determine which node should store an item. When hashing, a peer that wants to insert or retrieve an item applies a hash function on this item and looks at its Distributed Hash Table (DHT) in order to find the coordinates corresponding to where in the overlay the item should be stored. Consequently, the hash function being used is directly responsible for both determining the efficient retrieval of resources in the overlay and also the good distribution of data across the nodes.

Hashing can be an efficient load balancing technique to uniformly distribute data among peers. Given a set of i items and a set of n possible nodes to store these items, a perfect hash function should map each item to a distinct hashed value and all the nodes should be responsible for the same number of hashed values so that each node stores i/n items. However, such a hash function destroys the semantics of data and a uniform distribution usually implies the loss of the possible links between the items. The drawbacks associated with this random distribution of data affect, for instance, the time required for the execution of range queries, as the matching results are possibly distributed all over the network. This is why some distributed systems for data storage and retrieval tend to use an order-preserving hash function to distribute data across nodes, as proposed by Cai et al. (2004) and more recently for Big Data management systems (Escriva et al. 2012, Filali et al. 2011). Data is no longer randomly distributed, and syntactically close items can be retrieved in a minimum number of hops. Peers still use a hash function to determine an item's location: the function uses the natural value of the item (e.g. its lexicographic value) to generate its coordinates in the overlay's identifier space. Unfortunately, this technique cannot always guarantee a satisfying distribution of data among peers since such a hash function cannot efficiently disseminate skewed values. Indeed, when hashing similar keys using an order-preserving hash function, the hashed values are consequently similar as well and may be mapped to a very small area of the identifier space. Such a poor data dissemination may have major impact on the performance of a system, if large workloads are sent to very few nodes. This is a typical challenge for Big Data management systems, as the datasets they manage very often contain skewed values. For instance, Kotoulas et al. (2010) have shown a DBpedia (Auer et al. 2007) dataset, made of extracted information from Wikipedia, may contain the same key (a popular term) in up to 55% of its records.

As hash functions are at the heart of data distribution for DHT-based overlays, we present in this paper a new load balancing technique relying on the use of several order-preserving

hash functions at the same time in a network, to improve the skewed data dissemination. To our knowledge, this is the first load balancing solution using multiple order-preserving hash functions in structured overlays. Our goal is to allow an overloaded peer to modify the hash function it applies on data, in order to be responsible for a smaller interval of values and hence store less items.

To summarise, our work makes the following contributions:

- We describe a protocol that allows a peer to change its hash function at run-time.
- We show that all peers should not necessarily use the same hash function in an overlay to determine an item's coordinates.
- We demonstrate through simulations that this strategy greatly improves the dissemination of Unicode-encoded data items.

The rest of the paper is structured as follows. In Section 2, we present existing load balancing strategies based on multiple hash functions in structured P2P networks. Then, in Section 3, we introduce our approach using variable order-preserving hash functions to dynamically distribute skewed datasets in structured overlays. In Section 4, we detail our solution along with the rules to follow when implementing it in order to maintain an overlay consistent. In Section 5, we present the experiments we ran in CAN and Chord overlays using highly skewed values from real world datasets. Finally, Section 6 concludes this paper.

2 Related work

P2P systems are an efficient and scalable solution for data storage and retrieval in large distributed environments. However, many P2P systems for data management face the problem of load imbalance between their nodes. These imbalances may be caused by different factors, like churn (Ho et al. 2013) or the lack of information availability (Misra et al. 2009). Many different solutions have been proposed to address these issues (Felber et al. 2014), including data replication (Pitoura et al. 2006), node replication (Gupta et al. 2004) or keys reassignment (Taank & Bharati 2013). In this section, we focus on load balancing schemes based on multiple hash functions.

The use of multiple hash functions is mainly suggested in the literature as a solution to address two load imbalance issues: it is usually proposed as a replication scheme for popular items, and more rarely as a way to distribute data items among peers.

2.1 Multiple hash functions for popular items

A data item may be more popular than others and, consequently, the peer storing this item has to face an important number of queries, which affects its processing capacity. To speed up response time when a request for a popular item is made, some load balancing strategies replicate a popular item by applying n different hash functions on this item, in order to replicate it at n different locations. For example, the original CAN paper proposes to allow a peer looking for an object obj to apply n hash functions on obj. Each hash function maps obj to a different CAN coordinate and a peer looking for obj can choose the hash function associated with the nearest coordinate to its location in order to minimize the number of

hops to retrieve obj. However, this technique requires that all peers know all the different hash functions used to place data in order to be able to choose from where to retrieve it afterwards.

Other papers propose similar replica placement strategies based on multiple hash functions to deal with popular items. Xia et al. (2009) consider a P2P system for file storage where a peer can replicate a popular file at a location found by applying on the file a uniform hash function currently unused in the overlay. At most m uniform hash functions may be used to replicate a popular file. If a file f, currently replicated at i different locations corresponding to hash functions $h_1(f), ..., h_i(f)$, exceeds a popularity threshold on a peer p, p has to replicate f at another location using $h_{i+1}(f)$. To find f's current number of replicas (i.e. the value of i), the paper proposes an algorithm that works as follows: p applies x hash functions $h_1, ..., h_x$ on f until h_x points to a peer that does not store f, i.e. $x = i + 1$. When a peer wants to retrieve f, it uses a random binary search algorithm: the peer applies $h_r(f)$ where r is a random value comprised between 1 and m. If f is not found when using this hash function, the peer applies $h_{r2}(f)$ where $r2$ is comprised between 1 and r, and so on until the peer finds f.

Mu et al. (2009) present their approach to dynamically increase or decrease the number of replicas for a popular object. The authors assume a peer can evaluate which of its items are the most popular, in terms of load consumption for a set of predefined load criteria. Every time a node reaches a given load threshold, it creates a i^{th} replica of its most popular item by using a hash function h_i and sends the replica to the corresponding peer. Each original object embeds a counter number (equal to i) indicating the number of replicas existing for this object. The authors also assume a peer can distinguish its own original items from those that are replicas received from other peers. Whenever the load of a peer caused by one of its own popular objects decreases under a given threshold (i.e. the object becomes less popular), the peer can launch a removal process of one of its replicas. Concerning the retrieval of items, a peer looking for an object obj for the first time can only use the default hash function known by all peers in the overlay. When retrieving the original item, the peer also retrieves its associated counter number and thus knows the range of hash functions used to replicate this item. If, later, the same peer wants to retrieve obj again, it will be able to contact one of the i locations of obj by computing a hash function between h_1 and h_i.

Wu & Wang (2009) use the concept of multiple hashes for data storage and retrieval in the Kademlia P2P network. The authors consider a system where objects are indexed in the overlay according to a keyword value. If all peers use the same hash function to index objects, a bad dissemination of objects occurs when most of them are associated with the same subset of popular keywords. The authors propose that the same keyword may be hashed different times with the same hash function, in order to produce different keys. Each of these keys refers to the same keyword but points to a different peer and hence objects associated with the same keyword can be indexed on different peers instead of only one. For example, $hash(word)$ is the default hash function of the system for keyword $word$ and derived hash functions h_2 and h_3 would respectively correspond to $hash(hash(word))$ and $hash(hash(hash(word)))$. Up to N hash functions can be used, thus the same keyword may be stored by N peers.

2.2 Multiple hash functions for data dissemination

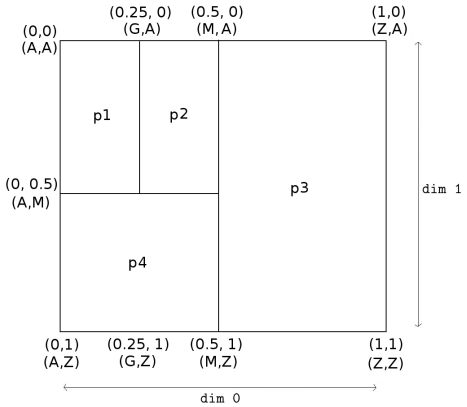
Byers et al. (2003) suggest the use of multiple hashing functions for a different purpose than popular items replication. Their approach, presented on the Chord P2P network, aims at addressing load imbalances in terms of items stored per peer, by trying different hash functions on an item to insert it at the least loaded location. To do so, the paper presents a variant of the power of two choices paradigm (Mitzenmacher 2001). The power of two choices paradigm consists in applying two hash functions picked at random on an item's key to eventually pick the least loaded node out of the two possible locations to store this item. This paper extends this paradigm to the use of two or more hash functions to compute the potential locations of an item. A node that wishes to insert an item applies d hash functions picked at random on the item's key and gets back d identifiers (each hash function is assumed to map items onto a ring identifier). Afterwards, a probing request is sent for each identifier computed previously and the peers managing the identifiers answer with their load. Once this load information is retrieved, the peer with the lowest load p_{low} is adopted for indexing the item. The other $d-1$ peers that were contacted but not selected receive a redirection pointer (key space identifier) to p_{low} for the corresponding item. When a peer wants to retrieve an object obj but does not know which hash function has been used to index obj, the peer applies on obj's key a random hash function $h_r(obj_{key})$ among the d possible hash functions that were applied previously to store obj. Even if h_r does not point to the peer storing obj, the peer receiving this query necessarily owns a redirection pointer to obj. Thus, a lookup can be achieved by using only one hash function among d at random and it may take only one more hop to reach the desired resource. However, this technique implies that all peers are aware of the d possible hash functions applied when indexing resources, in order to use one of them, albeit randomly chosen, when looking for an item.
The power of two choices paradigm is a load balancing solution that can be used with various types of distributed systems. Recently, Nasir et al. (2015) presented the direct applicability of this paradigm on distributed data stream processing systems, in order to address load imbalances in terms of sub-streams managed by each worker.

3 Multiple Hash Functions

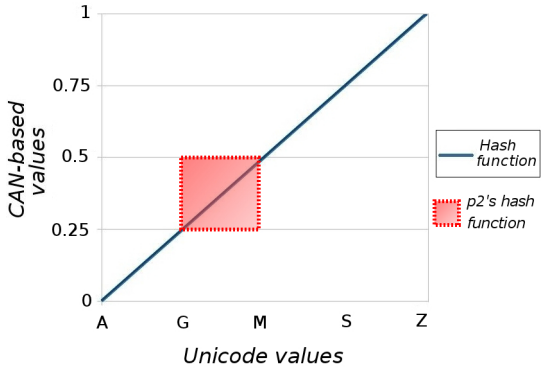
Unlike the existing strategies, we propose a solution based on multiple hash functions where each node can use its own hash function to place and retrieve data. Moreover, it is not required that all the nodes learn about all the hash functions that can be used in the overlay. To show the interest of our load balancing strategy, we present in this section how the use of multiple order-preserving hash functions can improve the distribution of Unicode-encoded data items according to the lexicographic order.

3.1 Order-preserving hash function

Let us consider an overlay made of n nodes (denoted as *peers*). Each peer p is responsible for an interval of coordinates $[min_p; max_p]$ in the overlay's identifier space ranging from O_{min} to O_{max}. For instance, Figure 1(a) presents a 2-dimensional CAN overlay where each peer manages a zone bounded by an interval on each dimension. Each peer's interval is constant and can only be modified during *join* or *leave* node operations.



(a) Example of a 2-dimensional Unicode CAN storing items in the form of $< attribute^0; attribute^1 >$. Dimension 0 is dedicated to $attribute^0$ values and dimension 1 to $attribute^1$ values.



(b) Standard hash function on dimension 0.

Figure 1 Standard hash function of a CAN, that can be used to determine the CAN coordinate between $[0; 1]$ to be associated with a Unicode value (between $[A; Z]$).

Minimum and maximum Unicode values U_{min} and U_{max} are set to determine the Unicode range that can be managed within the overlay: in Figure 1, U_{min} is equal to A and U_{max} is equal to Z (for more clarity, we use a single character to represent each bound's value). A Unicode value is associated with each bound of a peer and a peer is responsible for storing the items whose Unicode value falls between these bounds. For example, in Figure 1(a), $p1$ is responsible for data between $[A; G[$ (coordinates included in $[0; 0.25[$) on the horizontal dimension, and $[A; M[$ (coordinates included in $[0; 0.5[$) on the vertical one.

The mapping relation between overlay-based coordinates and Unicode-based values can be seen as a form of hash function. Indeed, to each Unicode value corresponds a coordinate between O_{min} and O_{max}, obtained by applying a given hash function on the Unicode value. This hash function provides coordinates that determine where a data item should be stored. The default hash function for the CAN of Figure 1(a) is shown in Figure 1(b). The graph describes which CAN coordinate is associated with which Unicode value, hence each peer is associated with a segment of the function ($p2$'s segment is highlighted in

Figure 1(b)). For instance, "G" corresponds to coordinate 0.25 on the hash function graph, which means this value is managed by $p2$ on dimension 0 because its interval is $[0.25; 0.5[$.

When a peer receives a new data item to insert or a query to execute, it has to convert the Unicode-encoded values into coordinates to check whether it is responsible for this item/query or not. For example, the hashed value of string $ProductType1$ in the context of an overlay storing worldwide data (wide Unicode range, up to codepoint value 2^{20}, as depicted in Figure 2) would be equal to coordinate 0.00004673 (i.e. at the far-left in the identifier space). By default, strings made of Latin characters have a low hashed value, whereas strings made of any East-Asian characters have high values (close to O_{max}) because such characters are located towards the end of the Unicode table. If the hashed value does not match the peer's coordinates, it means the peer is not responsible for the corresponding item. In this case, the peer forwards the item/query to a neighbour managing an interval closer to the requested one.

3.2 Skewed data, skewed distribution

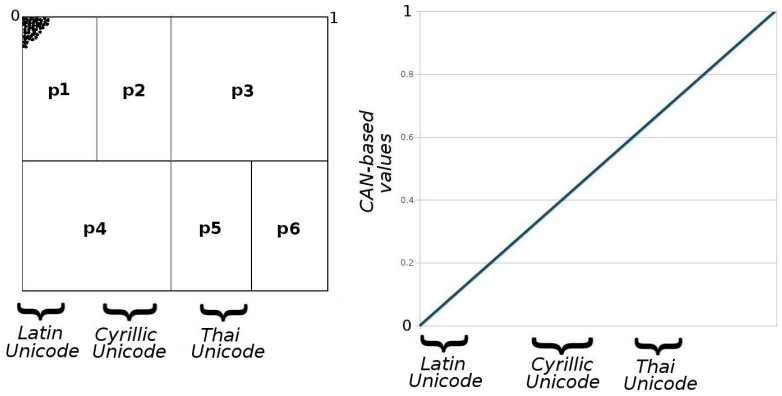


Figure 2 Default hash function inefficient to disseminate data items (represented as black dots at the top-left corner of the CAN).

The order-preserving storage technique presented above suffers from a major drawback regarding data distribution. Figure 2 shows a system where only items made of Latin characters are stored, which means only a very small area of the overlay is always targeted when storing or querying data. The most basic (no accent) Latin Unicode range only includes about 100 different characters whereas there are about one million characters in the whole Unicode table, which makes Latin-encoded values very skewed. In consequence, peer $p1$, whose zone includes the small Latin interval, may become overloaded, while the rest of the network stores nothing as it is dedicated to other Unicode characters. A solution could be to dedicate the entire network to Latin Unicode data storage. However, being able to manage characters in any language has now become essential, typically for any application using Web data. Data published by the social networks or in the Linked Data (Heath & Bizer 2011) format can be written in any language, using any Unicode character. For instance, the DBpedia datasets contain characters from 125 different languages. Nowadays, as more and

more data is produced, often to be processed in real time, it becomes crucial to dynamically adapt the allocation of resources depending on the incoming data, whose language may vary according to the current *trends* or even time of day because of the different time zones.

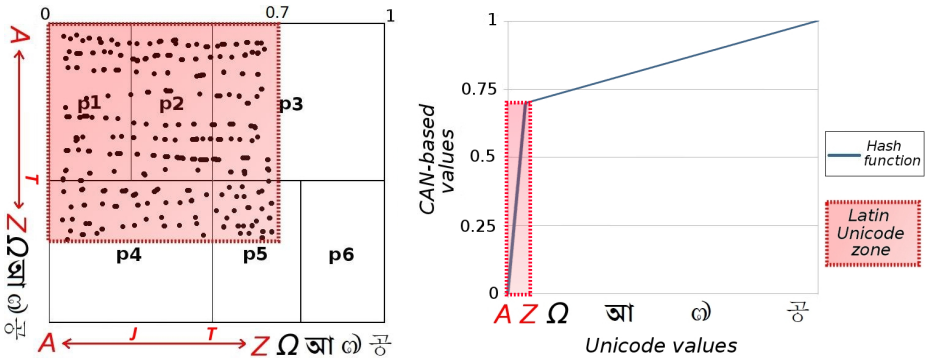
Based on this observation, our goal is to allocate more space for skewed values in the identifier space, while maintaining the lexicographic order as well as an area for all the other Unicode scripts.

3.3 Variable hash functions for overloaded zones

Enlarging the zone dedicated to the Latin data in the identifier space would result in an improved data dissemination among peers, as shown in Figure 3. Concretely, this means that each peer being fully included in the highlighted area becomes exclusively responsible for a given subset of Latin-encoded values. Thus, extending the interval of coordinates dedicated to a given Unicode subset requires that the peers in this interval reduce the Unicode interval they manage.

As this mapping between Unicode values and coordinates differs from the standard mapping previously introduced, a *non-standard* hash function must be used to place the Latin items. The corresponding hash function for the overlay in Figure 3(a) (to be applied on any "enlarged" dimension) is shown in Figure 3(b). Extending the Latin interval in the overlay leads to a different hash function for all the peers in the highlighted area, as the end of this interval (represented by the character Z) is now located at coordinate 0.7.

Figure 3 Possible enlargement of the interval for Latin Unicode data storage, in order to disseminate skewed values over several peers while still dedicating a part of the overlay to other Unicode scripts.



(a) Extension on both dimensions (dots represent data items).

(b) Non-standard hash function to place data items over the enlarged area for skewed values (highlighted on the graph).

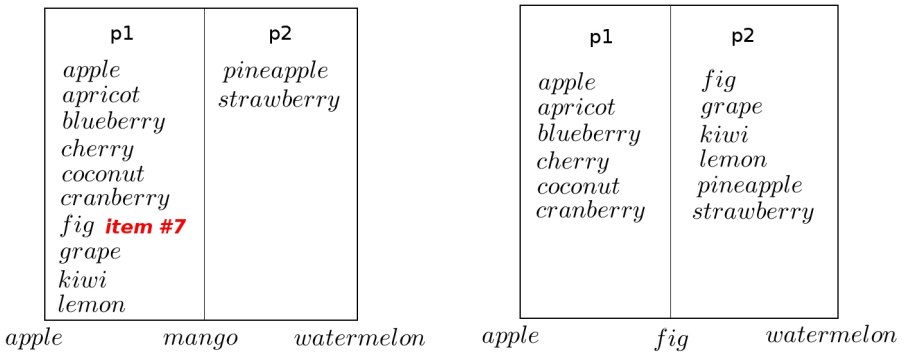
3.4 Computing a new hash function

In this subsection, we explain how a new hash function can be calculated to manage the load imbalance. We consider the number of items stored by a peer as the imbalance criterion, assuming all items are of a similar size.

Let p be a peer storing a given number of items noted as $load_p$ in its Unicode interval

$[Umin_p; Umax_p[$. If p is overloaded, it has to send all its items above a certain limit, lim, to its neighbour(s), such as the new value of $load_p \leq lim$. Basically, p has to reduce its Unicode interval, such as only at most lim data remain in p's zone. This is achievable by simply changing p's hash function from $h1$ to $h2$ such that $h2(Umax_p) = max_p$ and the new value of $Umax_p$ is equal to the Unicode value of p's $(lim + 1)^{th}$ item (as we consider p can sort its items lexicographically). Thus, by deciding how many items it wants to move, an overloaded peer also implicitly determines its new hash function. An example is shown in Figure 4, where a peer is considered as overloaded if it stores more than 6 items. In Figure 4(a), $p1$ is overloaded and has to change its hash function on max_{p1}, associated with the Unicode value *mango*. A new hash function $h2$ should transform $p1$'s 7^{th} triple fig (as lim is equal to 6) into max_{p1}. In Figure 4(b), $p1$ and $p2$ have changed their hash function to $h2$ respectively on max_{p1} and min_{p2}, and their load is evenly balanced.

Figure 4 Load distribution before and after rebalancing. In this example, a peer is said to be overloaded if it stores more than 6 items.



(a) Load imbalance between $p1$ and $p2$.

(b) Even data distribution after $p1$ has induced an hash function update.

4 Rules to Update a Hash Function

As there is no central coordination in an overlay, it is very difficult to change the overall hash function applied in the network. However, it is not necessary for all the peers to use the same hash function to get a fully functional overlay. Routing is done on the go, from one peer to another, and each peer applies its own hash function when receiving a message, to determinate its next recipient (as each peer should know the hash function used by its neighbours). Therefore, the initial sender of the message potentially has no precise idea about where in the system the message will be processed. Overall, different hash functions may be used as long as the rules presented below are observed.

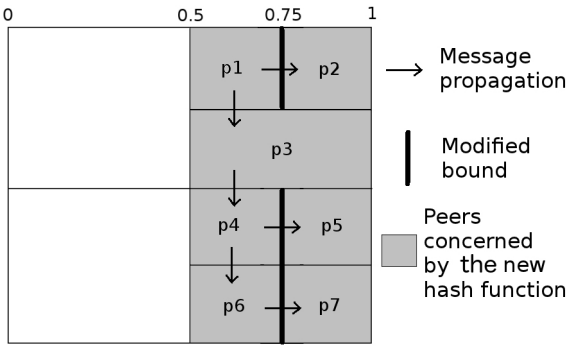
4.1 Message propagation

A peer should know the hash function of all its neighbours, and should notify them as soon as it changes its hash function. This ensures the arrival of new peers (division of an

existing zone between two peers) is correctly handled. Also, when routing an item/query in the network, this allows a peer to choose among its neighbours which one will be the most efficient for routing this message.
To ensure the overall consistency of the overlay, two neighbouring peers should share a common Unicode-based value, even if they use different hash functions. Given two neighbour peers $p1$ and $p2$ using $h1$ and $h2$ as their respective hash functions, if $max_{p1} = min_{p2}$, then $h1(Umax_{p1}) = h2(Umin_{p2})$. This rule prevents concurrent hash function changes from creating overlap or empty areas no one is responsible for in the identifier space.

In d-dimensional overlays, like CAN, all peers having a bound at a given coordinate should apply the same hash function on this bound, through the whole dimension. To maintain the routing optimal, peers whose interval includes this bound should also know this hash function, without needing to apply it. Figure 5 describes the impact of this rule by showing the scope of a hash function change in a CAN. Peer $p1$ decides to change its hash function to reduce its Unicode interval on its bound at coordinate 0.75. Thus, all peers on 0.75 (the grey area in Figure 5) should also apply this new hash function. A multicast message is sent by $p1$ and routed to all the concerned peers, which also includes $p2$. Indeed, $p2$ will not apply the new hash function but must be aware of it, as its CAN interval $[0.5; 1]$ comprises 0.75. This way, if a new peer wants to join $p2$ and split on its horizontal interval, $p2$ and the new peer will both know which Unicode value to apply on their new 0.75 CAN bound. Also, $p2$ must continue routing the multicast message in order to reach $p3$ and the other concerned peers. The overloaded peer cannot predict if this change will overload some of the concerned peers since they may not be its neighbours. However, this can be easily addressed, as the peers applying this new hash function can also trigger their own hash function change later on.

Figure 5 Hash function update message routing in a 2-dimensional CAN overlay.

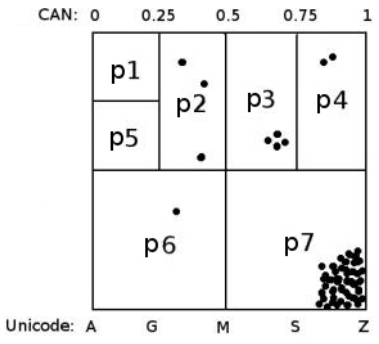


4.2 Bound reduction

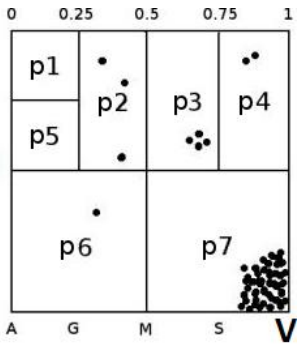
Peers having their upper bound in the identifier space equal to the maximum authorized coordinate O_{max} should also be able to reduce their upper Unicode bound. Otherwise,

if, for example, a large amount of non-Latin items is sent into the system, they would probably be stored by the peer at the far right in the identifier space, with no possibility for it to balance its load. To avoid this situation, we allow the peers located at the far right of the identifier space to reduce their upper Unicode bound like any other peer. To do so, we take advantage of the tore-ring topology of many overlays like CAN and Chord, and consider O_{min} = O_{max}, which means all Unicode updates associated to O_{max} must also be applied to the peer(s) on O_{min}. As a consequence, the peer(s) on O_{min} can become responsible for two Unicode intervals within a single interval of coordinates. This is shown in Figure 6, representing a CAN where the minimum Unicode value allowed U_{min} is A and the maximum U_{max} is Z. In Figure 6(c), peers $p1$, $p5$, and $p6$ own two different intervals: respectively $[[V;Z];[A;M[]$ for $p6$ and $[[V;Z];[A;G[]$ for $p1$ and $p5$. Therefore, when a peer is responsible for two Unicode intervals, the first interval must stop at U_{max} and the second one must start at U_{min}.

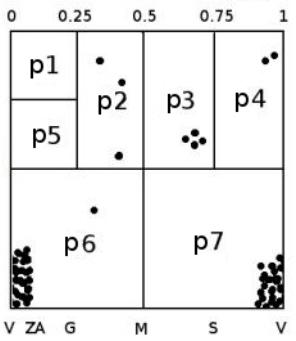
Figure 6 Unicode reduction process on the O_{max} bound.



(a) p7 gets overloaded.



(b) p7 reduces its upper bound from Z to V, so does p4.



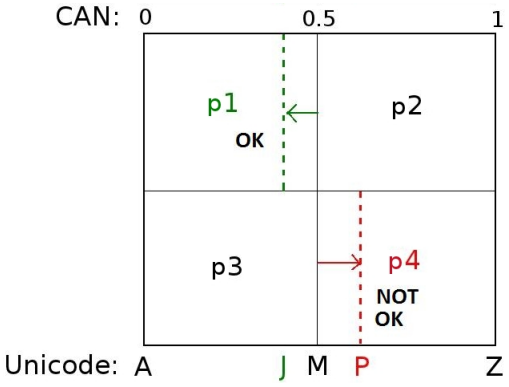
(c) p1, p5 and p6 now own two Unicode intervals.

4.3 Concurrency and message filtering

For a given bound, a peer p receiving an update message will allow the new Unicode value only if it is lower than p's current Unicode value for the same overlay-based bound. Thus,

it is forbidden to ask for a forwards reduction of the minimum bound, in order to avoid inconsistent concurrent reductions (i.e. on different directions). For example, in Figure 7, if two peers ($p1$ and $p4$) decided in a short time interval to modify their Unicode value (respectively $Umax_{p1}$ and $Umin_{p4}$) on the same coordinate (0.5) but on different directions (backwards for $p1$ and forwards for $p4$), it would become difficult to tell which one has priority over the other, especially since their multicast messages may not be received in the same order by all the concerned peers ($p1$, $p2$, $p3$, $p4$). Therefore, only $p1$ should be allowed to reduce its bound.
Moreover, a peer is not allowed to modify its hash function while it has received update messages it has not processed yet. Also, if update messages asking for different Unicode values are sent by different peers at the same time and for the same overlay-based bound, we ensure all peers will end up choosing the same value: the lowest one. Thus, the peers will reject the other messages and stop their multicast routing. This rule is very efficient as it guarantees the same value will be picked by all the concerned peers without any mutual consultation. The only case where a higher Unicode value would be accepted is when the reduction is made by a peer already containing two continuous intervals on one dimension (as discussed above in Section 4.2) and the new value is included in the first interval. In Figure 6(c), if, later, $p1$ wants to reduce from G to Y, it would no longer be responsible for two Unicode intervals (only $[V;Y[$), but its neighbour $p2$ would ($[[Y;Z];[A;M[]$).

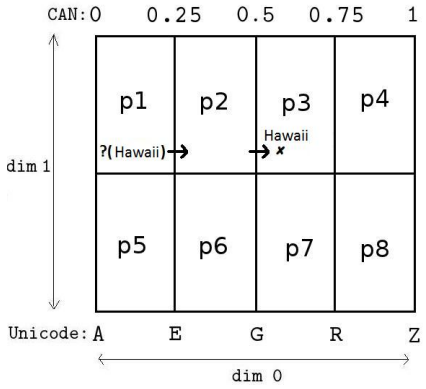
Figure 7 Inconsistent reductions for two bounds on the same coordinate (0.5).



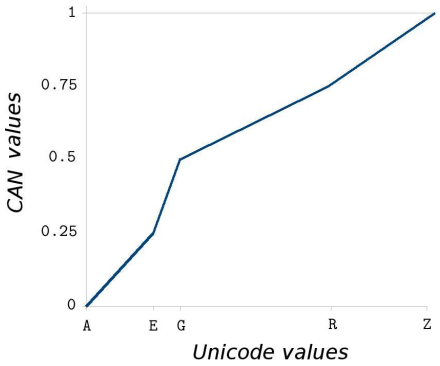
4.4 Data movement

After a peer p has changed its hash function, it has to send all the items it is no longer responsible for to its neighbour(s). To do so, p has to wait until they apply the update as well (it should be done quickly as they are only one hop away from p). In the meantime, p still processes queries for the items it wants to move. After sending them, p waits until it receives a storage acknowledgement message from its neighbour(s) before deleting these items, in order to ensure no data is lost during this process.

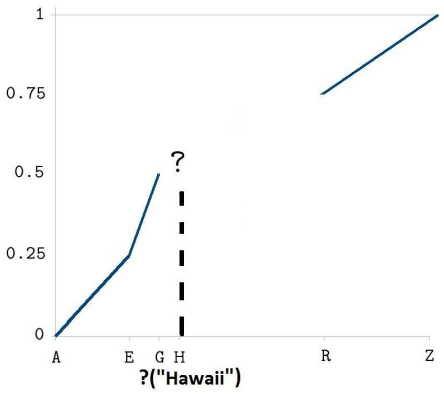
Figure 8 Routing process in an overlay using different hash functions.



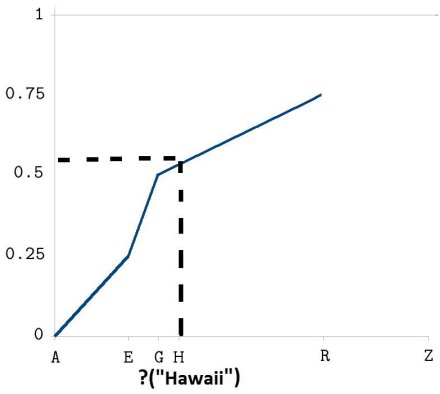
(a) Example of a CAN overlay where peer p1 is looking for value "Hawaii".



(b) Overall hash function for dimension 0.



(c) p1's hash function knowledge.



(d) p2's hash function knowledge.

4.5 Data retrieval

Using variable hash functions implies peers may not know where exactly in the overlay an item should be located. However, this does not affect data retrieval, if the rules set out in Section 4 are respected. Let us consider $p1$ initiates in Figure 9(a) a lookup query to retrieve a data item containing the value $Hawaii$ on dimension 0. Figure 9(b) represents the overall non-linear function used in the overlay, while Figure 9(c) shows $p1$'s knowledge about this hash function. As stated in 4.1, $p1$ knows the hash functions used by its neighbours $p2$ and $p4$ (torus topology). However, $p1$ has no idea about the hash function used between CAN coordinates 0.5 and 0.75 on dimension 0. As $p1$ cannot know to which CAN coordinates $Hawaii$ is associated, $p1$ will route its query towards the closest coordinate to $Hawaii$ it knows of. In our case, this corresponds to the value G associated to CAN value 0.5 on neighbour $p2$. Then, $p2$, whose hash function knowledge includes $p3$ (Figure 9(d)), will be able to estimate item $Hawaii$ should be stored by $p3$ and will route the query to $p3$.

5 Experiments

5.1 Experimental set-up

To validate our approach, we ran extensive experiments in the cycle-based simulator PeerSim (Montresor & Jelasity 2009), in order to perform large-scale experiments with many peers concurrently modifying their hash function at the same time. A cycle represents the time required by a peer to perform a basic operation (message routing, load checking, etc.). We simulate a network composed of 1000 peers. The maximum Unicode value supported U_{max} is set to character number 2^{20}, to encompass the whole Unicode table of characters. In order to make realistic experiments, we used a dataset made of real world data from two different sources. The first one is only composed of Latin Unicode characters, from the Berlin BSBM benchmark (Bizer & Schultz 2009). The second source is extracted from DBpedia and contains Japanese Unicode characters. Each data item takes the form of a *triple* of three Unicode-encoded elements (<*subject;predicate;object*>) according to the Resource Description Framework (RDF) (Lassila & Swick 1999) format used to represent Semantic Web data. The *subject* value represents a resource, the *predicate* is a property of this resource and the *object* is the value of this property. One million triples are inserted into the network in the space of 15 cycles, to simulate bursty traffic with a continuous insertion of data.

5.2 Load balancing policies

Various load balancing policies may be conceived to implement the multiple hash functions strategy, regarding how a peer evaluates its load state and how many items should be moved. Table 1 summarises the different policies we used during our experiments for these two behaviours.

5.3 CAN experiments

We simulated a 3-dimensional CAN for RDF data storage with each dimension being associated with one of the three parts of an RDF triple (Filali et al. 2011), as shown in Figure 9. Let us consider z_{smin}, z_{smax}, z_{pmin}, z_{pmax}, z_{omin} and z_{omax} the minimum (min)

Table 1 Load balancing policies used during our experiments. "Load state" refers to how a peer checks whether it should induce a rebalancing or not, and "Limit" describes how the new Unicode interval of an overloaded peer is calculated.

Strategy	Load State: p overloaded if	Limit: new value of $Umax_p$
Internal	$load_p$ > 8000 (CAN) or 1000 (Chord)	Unicode value of p's 8000^{th} (CAN) or 1000^{th} (Chord) item
Median	n/a	Value of p's median item
Local	$load_p$ > (30000+avg neighbours load)	Value of p's item at position corresponding to avg neighbouring load (including $load_p$)
Overall	$load_p$ > overall avg load $\times 15$	n/a

and maximum (max) borders of a peer's zone according to the subject axis (z_{smin}, z_{smax}), the predicate axis (z_{pmin}, z_{pmax}) and the object axis (z_{omin}, z_{omax}). We consider that a triple $t =< s, p, o >\in z$ only if $z_{smin} \leq s < z_{smax}$, $z_{pmin} \leq p < z_{pmax}$ and $z_{omin} \leq o < z_{omax}$, where s, p, o respectively represent the subject, the predicate, and the object of the triple t and z is a zone of the CAN overlay. For better understanding, consider the example presented in Figure 10. Suppose that node n_0 receives an $insert(t)$ request aiming to insert the RDF triple t in the network. Since no element of t belongs to the zone of n_0, and as s fits into the zone of n_1, n_0 routes the insert message to its neighbour n_1 according to the subject axis ($S - axis$). The same process will be performed by n_1, by means of the predicate axis ($P - axis$) which will forward the message to its neighbour n_3. Once received, n_3 checks whether one of its neighbours is responsible for a zone such as o belongs to this zone. Since the target peer is not found, the message will be forwarded at each step according to the object axis ($O - axis$) to the neighbour whose object coordinates are the closest to o. The idea behind this indexing mechanism is sketched in Algorithm 1.
Additionnally, in order to efficiently propagate the messages sent by the peers, an optimal broadcast algorithm (Henrio et al. 2013) is used. The goal of such algorithm is to ensure a message will be received by each peer only once.

Figure 9 3-dimensional CAN for RDF data storage.

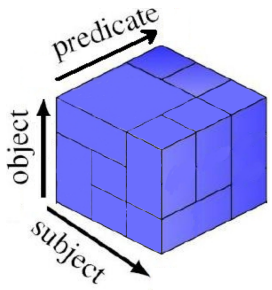
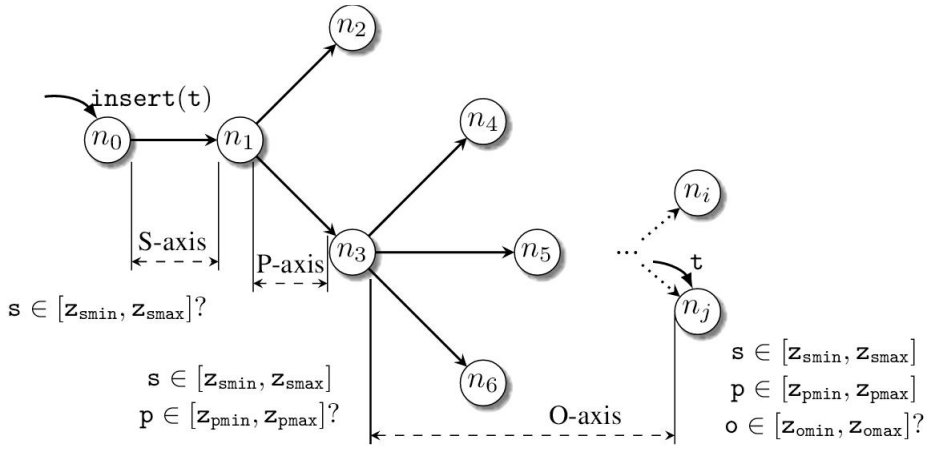


Figure 10 Insertion of RDF triples using an order-preserving storage technique.



Algorithm 1 Indexing algorithm

```
1:                                                    ▷ Code for peer P_i
2: upon event <Insert | t> from P_j
3:     if t ∉ Z_i then
4:         if s or p or o closer to one of my neighbours' zone then
5:             send <Insert | t> to Neighbour_dim
6:         end if
7:     end if
8: end event
```

5.3.1 Reference algorithm

In order to evaluate the efficiency of our solution, we ran a first series of experiments to compare our results with those obtained when using a well-known load balancing strategy relying on a unique and linear hash function. The scheme consists in adding a new peer at the location of an overloaded peer. This technique is the default CAN load balancing strategy and is used by famous P2P storage systems such as Meghdoot (Gupta et al. 2004). To perform experiments using this reference algorithm, we consider an overlay composed of a single peer, at the beginning. Then, periodically, a new peer is added at the most loaded peer's location. When a new peer p_{new} joins an existing zone managed by an overloaded peer p_{old}, p_{old} hands over half of the zone it manages in the coordinate space to p_{new}. Hence, the data located in this area does not belong to p_{old} anymore and is moved to p_{new}. For this different strategy, there is no particular balance to achieve as the peers do not compare their load with a threshold. The experiment consists in starting with one peer receiving the one million triples, then new peers are periodically added until having a network made of 1000 peers. Load balancing is triggered by an external event (a new peer joining), thus the experiments were stopped once all peers were added.

5.3.2 Results

Table 2 Data dissemination over peers and cost of each load balancing strategy to achieve balance.

Strategy	Peers storing data	Changes of hash function	Items moved	Cycles to achieve balance	Standard deviation
None	2	0	0	n/a	235702
Add peers	407	0	9334233	n/a	1487
Internal	652	156	10333076	80	1618
Local	818	170	5153092	100	2675
Overall&Median	602	90	4626023	45	1900

Table 2 summarizes the results obtained for each load balancing strategy at the end of simulations. Results are expressed in terms of number of peers storing data (out of 1000), hash functions updates sent by overloaded peers, cumulated total number of triples moved from peers in order to balance their load, cycles to achieve balance (after all triples are stored by the very first peer) and standard deviation in terms of RDF triples stored.

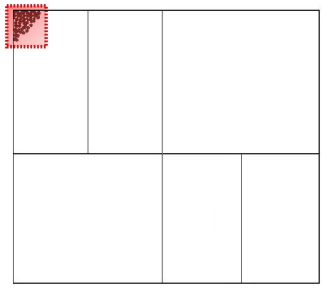
Without applying any load balancing strategy, results show that only 2 peers would store data, because of the large bias between datasets (Latin vs. Japanese characters). These two peers respectively correspond to the one at the far-left of the CAN on each dimension, responsible for storing all Latin triples, and the peer at the far-right of the CAN on each dimension, responsible for storing all Japanese triples.
When using multiple hash functions, data was disseminated over up to 818 peers (with the "local" policy). A perfect distribution would correspond to all 1000 peers storing 1000 triples. However, this is not achievable in practice for two reasons. First, the CAN topology may not be perfect (some zones larger than others). Secondly, real RDF data is often very skewed, due to the large scope of the whole Unicode and the fact that the RDF triples may contain very similar values, like ID values that differ by a single character. Also, the *predicate* values are often not very distinct. For instance, thousands of different triples could share the same predicate: *typeOf*, specifying the RDF type of the subject. Overall, our datasets contained less than 50 different values of predicate, which explains the impossibility to perfectly distribute data on the *predicate* dimension of the CAN.
When adding peers at the most loaded locations, i.e. when using a linear hash function shared by all peers, data is far less well distributed (407 peers). The main reason for this lies in the fact that the skewed data is mapped to an extremely small area in the identifier space, as shown in Figure 11(b), depicting a simplified version of the CAN after several peers have joined the most loaded peer. By adding new peers, data is moved from one peer to another but the items' coordinates still remain the same as the hash function applied by all peers on data does not change. This considerably limits data dissemination and, at some point, becomes ineffective.

Using variable hash functions required to move an important volume of data to balance the load. However, our results show that a more conventional strategy fails to achieve better results in terms of data items moved between peers: around 9 million triples were

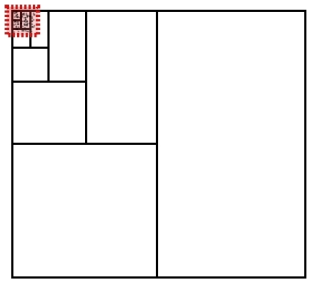
moved when adding peers against approximately 4.5 and 5 million for the local and overall schemes. Therefore, our solution, although not perfect, is not more costly regarding data transfer than other existing load balancing strategies. Moreover, it has the advantage that it does not require data to be moved far away in the overlay because only one hop is necessary to move data to a direct neighbour. To date, generally speaking, it remains impossible to address the load imbalance issue regarding data storage without having to move large amounts of data.

To ensure that the load balancing operations do not affect the consistency of the overlay, 200 random queries were sent to random peers throughout the experiments. For all of them, the correct result was received within a reasonable time (an average of 15 hops to route queries or results, hence 15 cycles). The number of update bound messages created by peers and the fact that these updates may not be applied by all peers at the same time did not affect the resolution of queries within our system. This is because an item is always stored at least on one peer. Moreover, moving data is only done once an update bound message is applied by both the sender and the receiver. Thus, reaching the new responsible peer for a given item should not require more than a few hops in the network, at the very most.

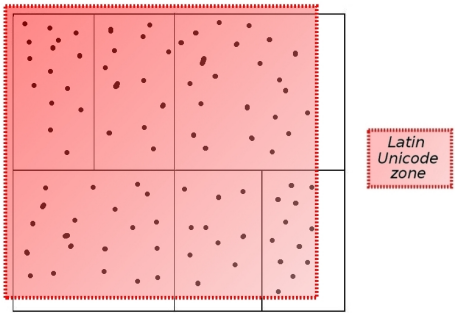
Figure 11 Load distribution depending on the strategy used.



(a) Standard CAN.



(b) CAN with peers iteratively added at the most loaded location.



(c) CAN using variable hash functions.

We ran a second experiment to evaluate the efficiency of our load balancing policies depending on the two behaviours chosen for estimating the load state of peers and selecting

Table 3 Variation regarding the data distribution when using different load balancing policies. This table shows the number of peers (out of 1000) storing data at the end of experiments depending on how a peer evaluates its load state, associated to how an overloaded peer defines its new Unicode interval ("Load to move").

Load to Move \ Load State Estimation	Internal	Local	Overall
Internal	652	433	508
Local	715	818	945
Median	672	577	602

the load to move. Table 3 presents the different results regarding data distribution among peers. Results show that the data distribution can be notably improved depending on these two behaviours. It is interesting to note that a better distribution occurs when taking into account the load information of other peers to calculate the amount of load to move. When being associated to an overall knowledge of the network load to estimate a peer's load state, the distribution is almost perfect, with 945 peers storing data out of 1000. As such knowledge might not be achievable in practice, relying only on the load information received from neighbours or even no external information to estimate a peer's load state also seems to offer a convincing data dissemination (respectively 818 and 715 peers concerned). This means we could consider a strategy with no load information exchange before a peer gets oveloaded and, only in this case, messages would then be exchanged with neighbours to estimate their load and hence, the amount of load to move. Such strategy could be regarded as efficient for a system whose peers only know a very small subset of the network (their neighbours), and have few or no contact with other peers. Indeed, with this strategy, no communication between peers is needed to exchange load information as long as no peer gets overloaded. Moreover, the load information exchange would be restricted as an overloaded peer would only contact its neighbours to determine the best amount of load to move.
These results show that the efficiency of this technique is strongly linked with the policies used to determine a peer's load state and the new hash function to be applied by the overloaded peer. Many variants are possible depending on what are the key issues for a particular system: we tried to distribute the load as much as possible but it is also conceivable to opt for a variant offering a lower dissemination in order to offload some machines while avoiding large data transfers.

5.4 Chord experiments

Finally, we ran another experiment to show the multiple hash functions strategy can also be applied to a Chord overlay for Unicode-encoded data storage.

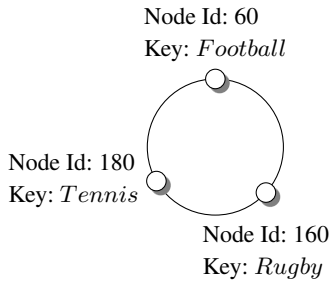
5.4.1 Multiple hash functions in a Chord overlay

We consider the same context that was presented in Section 5.1. Each node n, represented by a fixed *Node Id*, is associated with a *key* identifier, and should store all items whose hashed value is comprised between $]key_{n.predecessor}; key_n]$. The default hash function used in Chord destroys the natural ordering of data that we want to preserve. Thus, our default hash function will maintain the lexicographic order. On the ring, this means that

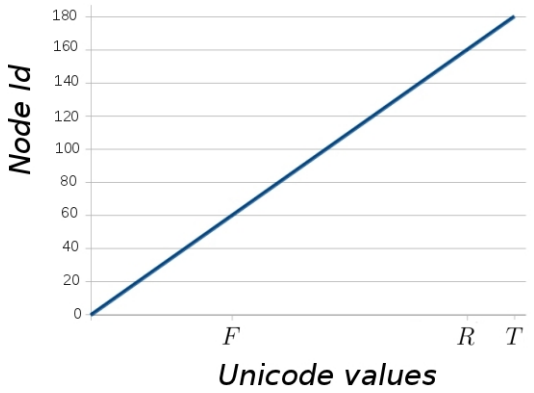
nodes are assigned key identifiers ordered according to the lexicographic order. A key refers to an attribute of the data being stored, hence this technique may be applied more generally for any NoSQL data storage for instance. In our example, the key corresponds to the value of an RDF triple's object. We chose the *objects* as their values are not as biased as *predicates*, hence this should result in a better distribution. A triple t will be located at $successor(t_{object})$: the successor peer of t's object value. As keys correspond to only one part of a triple, routing a query or a triple can be made according to this part only.
We use the node identifier (*Node Id*) of a node as the fixed coordinate to be associated with the node's key identifier, which can vary, to establish the hash function of the overlay. Unlike the default Chord implementation that assigns node and key identifiers using a consistent hash function, we separate the assignment of node identifiers from the assignment of key identifiers. By default, the key identifier of a node is computed using its node identifier and, very often, the ambiguous notion of *identifier* is used when referring to either a node or key identifier. In our implementation, both terms refer to two different notions. A *node identifier* refers to a node and its value can be computed by applying a consistent hash function on the node's IP address (we consider this value should not change), like in any Chord overlay. The *key identifier* of a node refers to the maximum key value managed by this peer. We allow a peer's key identifier to change over time by using variable hash functions. This implies, among other things, that peers should dissociate node and key identifiers when routing a message: in our overlay, routing should only be done according to a peer's key identifier. Therefore, the knowledge maintained by peers about their neighbours or finger table entries should include both node and key identifiers. More precisely, a finger table entry in our Chord implementation should contain the node identifier and the IP address of a peer (as in any Chord overlay) to identify a node, but also the key identifier associated to this peer in order to resolve lookups.

Whenever a peer is overloaded, according to a given load balancing strategy (the same as those described in Table 1), this peer can reduce its key identifier value in order to store less data. For example, in Figure 12, $Node\ 160$ ($N160$) has reduced its key identifier to only store values between interval $]Footboll; Golf]$. As a consequence, its successor $Node\ 180$ has a wider Unicode interval to manage ($]Golf; Tennis]$). As presented in Section 4.1, a peer updating its hash function must notify its neighbours. This means $N160$ has to notify all its successor(s). As there is no notion of dimension in Chord, there is no need to propagate this information any farther to maintain the overlay consistent. The new hash function, depicted in Figure 12(d), requires $N180$ to capture part of $N160$'s keys (hence also data items) between $]Golf; Rugby]$. However, other peers might still include $N160$ in their finger table associated with key value $Rugby$. As the finger table link is unidirectional, $N160$ does not know which peers are concerned, thus cannot directly inform them of its new key identifier (i.e. its new hash function). A similar problem is experienced by the default Chord overlay when a new peer joins the system and captures some keys but the existing nodes' finger table entries are not up-to-date. To address this issue, Chord uses a stabilization protocol to help peers maintain correct information about their successor, predecessor and finger table entries. By default, each node periodically runs this protocol to learn about new nodes. In our case, they also learn about new hash functions (i.e. new key identifiers).

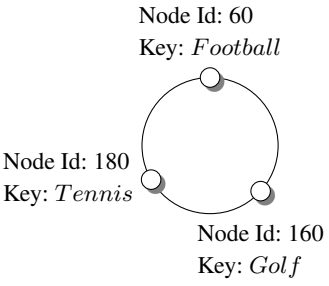
Figure 12 Multiple hash functions in a Chord overlay.



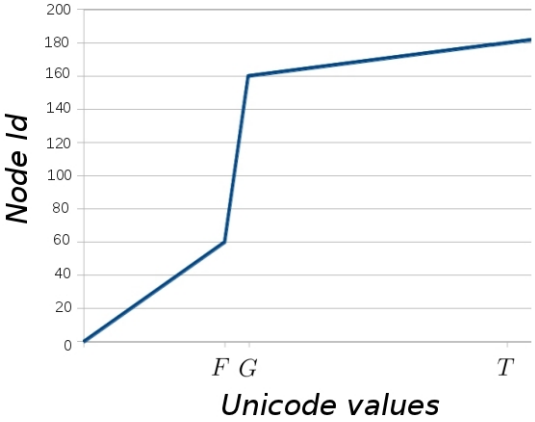
(a) Default Chord.



(b) Default hash function.



(c) Chord after $Node$ 160 has reduced its key from $Rugby$ to $Golf$.



(d) New hash function.

5.4.2 Reference algorithm

In order to evaluate the efficiency of our solution, we also ran a series of experiments using another load balancing strategy, relying upon the migration of underloaded nodes. We chose this solution as it is commonly used (Bharambe et al. (2004), Konstantinou et al. (2011)) and suited to the constraints of order-preserving storage systems. This mechanism is based on low overhead random sampling to create an estimate of the data value and load distribution. Basically, each peer periodically sends a probing request to another peer using random routing. This offers a global system load assessment whose values are collected into histograms maintained on peers. Using this information, a heavily loaded node can contact a lightly loaded node and request it to leave its location in the routing ring (and hence hand over its data to a neighbour) and re-join near the location of the heavy node (*leave-join* mechanism), to become its predecessor. Finally, the underloaded peer captures half of the keys the heavy peer (its successor) is responsible for. To perform experiments using this reference algorithm, we consider a Chord ring composed of a 1000 peers and use some of the policies presented in Table 1. We apply the overall strategy to consider a peer as overloaded and consider a peer as underloaded if its load is lower than the overall average load. The median strategy is used to determine which items should be moved by an overloaded peer.

5.4.3 Results

Table 4 Load balancing results for each strategy applied on a Chord ring.

Strategy	Peers storing data	Changes of hash function	Items moved	Peers moved	Standard deviation
None	2	0	0	0	235702
Leave-join	712	0	30938483	323	1438
Internal	1000	1328	23589693	0	743
Local	806	1874	23098537	0	6353
Overall&Median	760	1339	21313525	0	1141

Table 4 summarises the results obtained for our experiments in a Chord overlay. When moving underloaded peers to overloaded zones (leave-join mechanism), the load is distributed over 712 peers. The amount of items moved between the peers is the highest out of all our experiments, as the underloaded peers must send their items to a neighbour before leaving their location, which increases data transfer between nodes. At the end of our experiments, 323 underloaded peers had left their initial location, which implies many links had to be repaired between the peers.
Conversely, the multiple hash functions strategy did not require to modify the network topology and was able to distribute the items over more peers. The threshold strategy allows to share workload among all peers as we consider only one dimension based on the object of triples, whose value is much more distinct than the predicate value. Such perfect distribution was obtained as the volume of data to be inserted was known beforehand, which helped us estimate the perfect threshold. The goal of such experiment was to show how

efficient variable hash functions can be to disseminate data in a Chord overlay. However, in practice, it might be harder to distribute data over 100% of the nodes if the amount of data to store is not known in advance.
The local strategy, which measures the load of a peer's successors to decide whether to induce a rebalance or not, was able to distribute data among 806 peers. The overall strategy offers the lowest standard deviation between the 760 peers storing data at the end of experiments. These two schemes were less effective to distribute data as too low threshold/coefficient values would have caused an infinite oscillation, hence the need to use higher values that may not trigger load balancing as often as expected to obtain a perfect distribution.

For all these strategies, the number of triples to move and the number of hash functions changes were higher than those obtained when using a CAN overlay, for several reasons. First, the better distribution among peers consequently involves more triples to be moved between peers. More importantly, triples are moved from successor to successor, as they have to apply the new hash function, which may lead them to rebalance their load as well. Finger tables, that would help save some hops, are not updated as quickly and hence are not as efficient as usually. Finally, in a CAN overlay, a hash function update is propagated on the whole dimension, which means many peers may potentially benefit from this change. Indirectly, a single overloaded peer may prevent many other peers from being overloaded when asking them to modify their hash function, and hence these peers will not need to initiate their own update later on. In our Chord approach, such saving is impossible and each overloaded peer has to trigger its own hash function update.

6 Conclusion

When storing highly skewed values in an order-preserving manner, the distribution of data among peers is consequently highly skewed as well. Since hash functions are at the heart of data distribution in DHT-based overlays, we proposed to address this issue by allowing an overloaded peer to modify the hash function it applies on data, so that it can reduce the interval of values it is responsible for. This results in the coexistence of several different hash functions used by different peers. In order to maintain optimal routing and consistency in the overlay, we have shown that it is not necessary for all the peers to use the same hash function as long as some rules are observed. To our knowledge, this is the first load balancing solution for structured overlays based on the coexistence of multiple order-preserving hash functions to improve data dissemination. To evaluate the effectiveness of this solution, we simulated an overlay composed of 1000 peers and injected 1 000 000 highly skewed RDF triples (Semantic Web data). Results have shown that the number of peers storing data is 2 out of 1000 when using no load balancing strategy. When implementing this multiple hash functions solution, up to 945 peers store data at the end of experiments. We also compared our solution with a well-known strategy consisting in adding a peer at the location of an overloaded node. Our solution appeared more efficient to disseminate the load and also less costly regarding the amount of data to move to achieve balance. We performed our experiments on both CAN and Chord overlays but we believe that the use of multiple hash functions could be applied to other DHT-based overlays. Likewise, we only used RDF data in

our experiments but the same principles could be followed with any other Unicode-encoded data representation (e.g. key-value pairs, tuples).

References

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R. & Ives, Z. (2007), Dbpedia: A nucleus for a web of open data, *in* 'The semantic web', Springer, pp. 722–735.

Bharambe, A. R., Agrawal, M. & Seshan, S. (2004), Mercury: supporting scalable multi-attribute range queries, *in* 'ACM SIGCOMM computer communication review', Vol. 34, ACM, pp. 353–366.

Bizer, C. & Schultz, A. (2009), 'The berlin SPARQL benchmark', *International Journal on Semantic Web and Information Systems (IJSWIS)* **5**(2), IGI Global, 1–24.

Byers, J., Considine, J. & Mitzenmacher, M. (2003), Simple load balancing for distributed hash tables, *in* 'Peer-to-Peer Systems II', Springer, pp. 80–87.

Cai, M., Frank, M., Chen, J. & Szekely, P. (2004), 'MAAN: A multi-attribute addressable network for grid information services', *Journal of Grid Computing* **2**(1), Springer, 3–14.

Dhurandher, S. K., Misra, S., Obaidat, M. S., Singh, I., Agarwal, R. & Bhambhani, B. (2009), Simulating peer-to-peer networks, *in* '2009 IEEE/ACS International Conference on Computer Systems and Applications', IEEE, pp. 336–341.

Dhurandher, S. K., Misra, S., Pruthi, P., Singhal, S., Aggarwal, S. & Woungang, I. (2011), 'Using bee algorithm for peer-to-peer file searching in mobile ad hoc networks', *Journal of Network and Computer Applications* **34**(5), Elsevier, 1498–1508.

Escriva, R., Wong, B. & Sirer, E. G. (2012), 'Hyperdex: A distributed, searchable key-value store', *ACM SIGCOMM Computer Communication Review* **42**(4), 25–36.

Felber, P., Kropf, P., Schiller, E. & Serbu, S. (2014), 'Survey on load balancing in peer-to-peer distributed hash tables', *IEEE Communications Surveys & Tutorials* **16**(1), 473–492.

Filali, I., Pellegrino, L., Bongiovanni, F., Huet, F. & Baude, F. (2011), Modular P2P-based approach for RDF data storage and retrieval, *in* 'Proceedings of the international conference on Advances in P2P Systems', Citeseer, pp. 39–46.

Gupta, A., Sahin, O. D., Agrawal, D. & Abbadi, A. E. (2004), Meghdoot: content-based publish/subscribe over P2P networks, *in* 'Proceedings of the international conference on Middleware', Springer-Verlag New York, Inc., pp. 254–273.

Heath, T. & Bizer, C. (2011), 'Linked data: Evolving the web into a global data space', *Synthesis lectures on the semantic web: theory and technology* **1**(1), Morgan & Claypool Publishers, 1–136.

Henrio, L., Huet, F. & Rochas, J. (2013), An optimal broadcast algorithm for content-addressable networks, *in* 'International Conference on Principles of DIstributed Systems', Springer, pp. 176–190.

Ho, C.-Y., Chung, M.-C., Yen, L.-H. & Tseng, C.-C. (2013), Churn: a key effect on real-world p2p software, *in* '2013 42nd International Conference on Parallel Processing', IEEE, pp. 140–149.

Konstantinou, I., Tsoumakos, D. & Koziris, N. (2011), 'Fast and cost-effective online load-balancing in distributed range-queriable systems', *Parallel and Distributed Systems, IEEE Transactions on* **22**(8), 1350–1364.

Kotoulas, S., Oren, E. & Van Harmelen, F. (2010), Mind the data skew: distributed inferencing by speeddating in elastic regions, *in* 'Proceedings of the 19th international conference on World wide web', ACM, pp. 531–540.

Lassila, O. & Swick, R. R. (1999), Resource Description Framework (RDF) model and syntax specification, (Available from: http://www.w3.org/TR/REC-rdf-syntax/.) [Accessed on 20 November 2016].

Maymounkov, P. & Mazieres, D. (2002), Kademlia: A peer-to-peer information system based on the xor metric, *in* 'Peer-to-Peer Systems', Springer, pp. 53–65.

Misra, S., Dhurandher, S. K., Obaidat, M. S., Singh, I., Bhambhani, B. & Agarwal, R. (2009), On increasing information availability in gnutella-like peer-to-peer networks, *in* '2009 IEEE International Conference on Communications', IEEE, pp. 1–5.

Mitzenmacher, M. (2001), 'The power of two choices in randomized load balancing', *Parallel and Distributed Systems, IEEE Transactions on* **12**(10), 1094–1104.

Montresor, A. & Jelasity, M. (2009), Peersim: A scalable p2p simulator, *in* 'Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on', IEEE, pp. 99–100.

Mu, Y., Yu, C., Ma, T., Zhang, C., Zheng, W. & Zhang, X. (2009), Dynamic load balancing with multiple hash functions in structured p2p systems, *in* '2009 5th International Conference on Wireless Communications, Networking and Mobile Computing', IEEE, pp. 1–4.

Nasir, M. A. U., Morales, G. D. F., García-Soriano, D., Kourtellis, N. & Serafini, M. (2015), The power of both choices: Practical load balancing for distributed stream processing engines, *in* '2015 IEEE 31st International Conference on Data Engineering', IEEE, pp. 137–148.

Navimipour, N. J. & Milani, F. S. (2015), A comprehensive study of the resource discovery techniques in peer-to-peer networks, *in* 'Peer-to-Peer Networking and Applications', Vol. 8, Springer, pp. 474–492.

Pitoura, T., Ntarmos, N. & Triantafillou, P. (2006), Replication, load balancing and efficient range query processing in dhts, *in* 'Advances in Database Technology-EDBT 2006', Springer, pp. 131–148.

Ratnasamy, S., Francis, P., Handley, M., Karp, R. & Shenker, S. (2001), A scalable content-addressable network, *in* 'ACM SIGCOMM Computer Communication Review', Vol. 31, ACM, pp. 161–172.

Rowstron, A. & Druschel, P. (2001), Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems, *in* 'Middleware', Springer, pp. 329–350.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. & Balakrishnan, H. (2001), Chord: A scalable peer-to-peer lookup service for internet applications, *in* 'ACM SIGCOMM Computer Communication Review', Vol. 31, ACM, pp. 149–160.

Taank, C. & Bharati, R. (2013), Load balancing algorithm for dht based structured peer to peer system, *in* 'International Journal of Emerging Technology and Advanced Engineering Website: www. ijetae. com (ISSN 2250-2459, ISO 9001: 2008 Certified Journal, 3 (1)', Citeseer.

Wu, T.-T. & Wang, K. (2009), An efficient load balancing scheme for resilient search in kad peer to peer networks, *in* 'Communications (MICC), 2009 IEEE 9th Malaysia International Conference on', IEEE, pp. 759–764.

Xia, Y., Chen, S., Cho, C. & Korgaonkar, V. (2009), Algorithms and performance of load-balancing with multiple hash functions in massive content distribution, *in* 'Computer Networks', Vol. 53, Elsevier, pp. 110–125.