Technical Report RT/46/2010

Rollerchain: a DHT for High Availability

João Paiva
INESC-ID / IST
jgpaiva@gsd.inesc-id.pt

João Leitão
INESC-ID / IST
jleitao@gsd.inesc-id.pt

Luis Rodrigues
INESC-ID / IST
ler@ist.utl.pt

November 2010
Abstract

In this paper we present Rollerchain, a novel Distributed Hash Table that offers high availability of stored data in an efficient manner. Rollerchain is based on the combination of unstructured and structured overlay networks: its unstructured component maintains clusters of fully connected nodes, where each cluster acts as a virtual node in the structured component. Additionally, Rollerchain benefits from the availability of multiple replicas to balance the load among all the nodes of the system. We have implemented a prototype of Rollerchain that we have used to experimentally validate the benefits of this approach.
Thicket: A Protocol for Building and Maintaining Multiple Trees in a P2P Overlay

João PAIVA
INESC-ID / IST
jgpaiva@gsd.inesc-id.pt

João Leitão
INESC-ID / IST
jleitao@gsd.inesc-id.pt

Luís Rodrigues
INESC-ID / IST
ler@ist.utl.pt

Abstract

In this paper we present Rollerchain, a novel Distributed Hash Table that offers high availability of stored data in an efficient manner. Rollerchain is based on the combination of unstructured and structured overlay networks: its unstructured component maintains clusters of fully connected nodes, where each cluster acts as a virtual node in the structured component. Additionally, Rollerchain benefits from the availability of multiple replicas to balance the load among all the nodes of the system. We have implemented a prototype of Rollerchain that we have used to experimentally validate the benefits of this approach.

1 Introduction

Distributed Hash Tables (DHTs) [25, 23, 21, 9] have been widely studied as they can support several types of large-scale distributed applications and services, such as resource-location [1], publish subscribe [2], multicast [28], distributed storage [3], among others [22, 4].

DHTs are structured overlays, i.e., overlays where nodes organize themselves into a predefined topology, such as a ring. The relative position of nodes in the ring is a function of their identifiers (usually large random bit strings). One important advantage of structured overlays is that their deterministic topology can be leveraged to support very efficient application-level routing in the identifier space. In fact, DHTs can efficiently map any arbitrary key in this identifier space to a peer currently active in the system (named the key owner). This functionality allows to support store/lookup operations of key/value data pairs in a distributed and scalable manner.

In a large-scale system, the membership is bound to be dynamic, as nodes voluntarily join, leave, or simply fail. Often, membership changes occur at a high pace, a phenomenon that is known as churn [27]. To avoid key/value pairs from being lost when a node leaves, they need to be replicated in the DHT, typically by keeping copies in the \( k \) successors of the key owner in the ring [25, 23, 4]. Other alternative replication schemes have also been proposed, such as adding salt to the key to store the value in different locations on the ring [22, 14, 26, 15]. A cost comparison of some of these approaches is presented in [16].

Unfortunately, the cost of replication under churn may be excessively high. In order to tolerate \( f \) faults, \( f + 1 \) copies of the key/value pair must be preserved. This requires data to be moved from one node to another, every time there is a membership change in the DHT. This adds up to the already non-negligible cost of maintaining the DHT under churn [16]. As a result, existing replication schemes may perform poorly or have to relax the availability properties, by risking the replication degree to go below the desired threshold.

In the paper we present a novel DHT that offers high availability of stored data in an efficient manner. The idea is that physical nodes use an unstructured overlay to self-organize into small replica groups of approximately the
same size. The number of such groups is not defined a priori, and grows with the total number of peers in the system. Each group is fully connected, and its members coordinate to operate as a single virtual node that joins a structured layer, building a DHT ring. This approach has the following benefits with regard to previous work:

- The structured overlay becomes more stable, as the size of a group that constitutes a DHT virtual node can fluctuate without affecting the DHT topology. As a result, multiple peers need to fail or join in the same group in order to affect the membership of the DHT.
- Since each DHT virtual node is composed of multiple cooperative peers, the system has inherent replication capacity. Thus, the use of multiple DHT nodes (with the associated cost of additional DHT routing steps) is not required to maintain the availability of the stored data.
- Finally, peers of the same group can cooperate to balance the load of both maintaining the DHT structure and routing/answering queries.

The resulting system is extremely resilient to failures and churn, and also less expensive to maintain than previous replication schemes for DHTs. We have called our DHT Rollerchain, as it is able to create a “chord” where links are highly robust, making the resulting “chain” hard to break. We have experimentally evaluated Rollerchain against simple replication schemes for Chord, using simulations of a system composed by 10,000 nodes populated with 500,000 objects. Results show that Rollerchain is indeed harder to break and more resistant to churn, has a smaller signaling cost, and promotes better load balancing.

The rest of this paper is organized as follows. Section 2 discusses related work in more detail. Section 3 provides a brief overview on the operation of the building blocks of our solution, Section 4 describes the operation of Rollerchain, and Section 5 presents the results from the experimental evaluation. Finally, Section 6 concludes the paper.

2 Related Work

2.1 Data Replication on DHTs

Several alternatives have been explored in the literature to support data replication in DHTs. There are three main approaches to this problem, namely: neighbor replication, multi-publication replication, and path replication. Each of these techniques operates as follows:

- in neighbor replication [25, 23, 4], as the name implies, the owner of a key replicates its data to a number of its neighbors (for instance to an equal number of predecessors and successors);
- in multi-publication [22, 14, 26, 15], the same content is stored using multiple identifiers, generated deterministically from the key (for instance, by using different hash functions);
- in path replication [28], replicas are created as a result of query processing, by caching the results in the nodes that route the query.

Each of these approaches has its own advantages and disadvantages, as discussed below.

A significant advantage of neighbor replication (also known as leaf set replication) is that it allows to keep a tight control on the replication degree. In fact, in a DHT, a node usually needs to keep track of its neighbors, even when replication is not in place. Thus, when the neighbors change, the owner may trigger the creation of new replicas to ensure that the replication degree does not fall below a target threshold. Unfortunately, neighbor replication requires data to be constantly moved to the nodes closer to its owner, inducing a significant overhead when the system experiences churn. Furthermore, depending on the routing scheme used in the DHT, when using neighbor replication some replicas are more likely to be hit by lookups than other replicas. This results in a sub-optimal load-distribution among replicas.

Multi-publication also allows to keep a tight control of the replication degree. Initially, $k$ replicas of the data are placed in different positions of the DHT. Then, some mutual monitoring scheme needs to be implemented to detect the departure/failure of a replica and subsequently restore the replication degree. The main advantage
of multi replication is that it offers very good load balancing properties, as multiple queries may be diverted to different regions of the ring. On the other hand, monitoring becomes very expensive, because it needs to use DHT routing and a node may be forced to monitor a different set of nodes for each object that it stores. In the worst case, a node that stores $M$ objects, each replicated in $k$ nodes, has to periodically monitor $M \times (k - 1)$ peers.

Path replication has been mainly proposed as a technique to speedup lookups and perform load balancing. Since replicas are only created when lookups are executed (in the nodes that have been used to route the query), it is hard to guarantee that a minimum target replication degree is preserved. Also, since it is impractical to keep track of the location of replicas, replicas can be discarded in an uncoordinated manner, as the purpose of caches is primarily efficiency.

2.2 Load Balancing on DHTs

Although the main goal of our work is to achieve high availability, we are also interested in leveraging on the node redundancy required for fault-tolerance to improve load balancing. As such, we also briefly overview the main techniques that have been used to provide load balancing in DHTs.

From the load balancing point of view, one fundamental problem in DHTs is that nodes or, most likely, keys, may not be uniformly distributed in the address space. As a result, some nodes will be required to maintain (and answer queries for) many items while others may be relatively offloaded.

A common technique to circumvent this problem is to use virtual servers [3, 7]. In this technique, each physical node joins the DHT using multiple identities; each identity represents a virtual node maintained by that physical node. This technique, by itself, already tends to balance the load among nodes and may be further improved by allowing nodes to enable or disable virtual peers in face of the observed demand. However, the efficiency of this mechanism depends on how many virtual servers each node can handle, as a larger number of such servers will lead to a better distribution of load. On the other hand, having a larger number of virtual servers involves each node maintaining more routing information and monitoring more overlay neighbors, which may impose an excessive overhead. Also, this strategy amplifies the effect of churn, as the departure of a single physical node causes the simultaneous failure of multiple virtual nodes.

Other approaches rely on making a guided choice of node identifiers at join time, to select positions in the identifier ring such that the load is evenly distributed among all nodes. In order to achieve this, works such as [12] and [17] use probes in the system to determine the best identifier to use. These schemes allow balancing the load of object storage among all nodes in the system without additional routing information, by increasing the cost of join operations. However, these schemes may create a non-uniform distribution of nodes in the identifier space, which hinders the performance of some routing algorithms.

2.3 Combining Structured and Unstructured Overlays

As it will become clear later in the text, our solution relies on the combination of structured and unstructured overlay mechanisms. Some previous systems have already explored this idea, although with different goals and, as a result, with solutions that are structurally quite different from Rollerchain:

The work by Ghodsi, Haridi and Weatherspoon [6] makes the case for combining gossip-based and structured networks, and discusses several examples of successful synergies between both designs. This work claims that through this symbiosis, the current state of the art on DHTs can be improved with new overlay designs that offer better reliability, lower bandwidth costs, or better geometry.

In [20], the authors present an approach which combines the design of two overlays, one unstructured and one structured, in order to build both overlays at a cost similar to that of building a single one. This paper presents

\footnote{Contrary to our design, where several physical nodes map into a single virtual node, in these approaches each physical node maps into several virtual nodes.}
an example of how a DHT and an interest-based unstructured network may create a symbiosis, reducing their maintenance costs.

Kelips [9] is a DHT structured using virtual nodes composed of several physical peers. Unlike Rollerchain, Kelips is aimed at medium-sized systems. In this system, each virtual node is composed of a large number of physical nodes, and each of these nodes knows at least one contact in every other virtual node, creating a one-hop DHT. Kelips supports efficient lookups, but it does not present any solution for data replication and each node stores a pointer to every file owned by its virtual node, a property that severely limits its scalability.

More recently, a combination of structured and unstructured overlays has been proposed to improve complex resource location in peer to peer systems [1]. In that system, named Curiata, peers are organized in an unstructured overlay that is biased such that peers with similar content become close to each other. For each region of the unstructured space, a small number of nodes are elected to join a DHT, that is used to quickly guide queries to the regions of the unstructured overlay that are more likely to contain the desired resources. However, Curiata is only concerned with resource location and does not address to data placement or replication policies.

3 Rollerchain Overview and Building Blocks

Rollerchain achieves high availability by combining features of unstructured and structured overlay networks in an integrated design. More specifically, Rollerchain builds a DHT where each (virtual) node is materialized by a small group of peers, the size of which depends on the replication degree, $R$. Peers on each virtual node share among themselves the information required to maintain the DHT topology and the data stored on the virtual node. These groups are neither static nor defined a priori. Instead they are dynamically created by the unstructured component. In more detail:

- The unstructured component of Rollerchain is responsible for creating and maintaining the virtual nodes. Some of its mechanisms are inspired by Overnesia, an unstructured overlay that aggregates peers in clusters [18].

- The structured component of Rollerchain runs the DHT maintenance algorithms. Rollerchain creates a ring of virtual nodes similar to that of Chord [25].

In order to introduce the basic concepts of both layers of Rollerchain, before we describe its operation in detail we provide a brief overview of the two solutions that inspire the design of each of the layers identified above.

3.1 Overnesia

Overnesia [18] is an unstructured overlay network where nodes self-organize into fully connected clusters which, in turn, are highly and randomly connected among themselves. The target size of these clusters can be configured by the application to a given targetClusterSize value. However, the cost of ensuring that every generated cluster produced by the protocol has exactly the same size in highly dynamic and open environments can be prohibitively high. To overcome this challenge, Overnesia instead ensures that the size of clusters is distributed between minClusterSize and maxClusterSize, with a predominance of clusters with the targetClusterSize (evidently, minClusterSize < targetClusterSize < maxClusterSize). Each Overnesia cluster is assigned a random identifier that is known by all elements of that cluster.

When a new node wishes to join the Overnesia network, it first contacts a peer already in the system. Then, it uses the links that connect different clusters to perform a random walk in the network in an attempt to find a suitable cluster to join. If no appropriate cluster is found when the random walk terminates, that node simply joins a random cluster. As expected, this process might lead to situations where the size of a cluster surpasses maxClusterSize. In this case, the cluster is split into two new clusters by means of a division procedure. In most
cases, when a node leaves a cluster it is later replaced by another node that joins the network. Therefore, clusters tend to preserve their size. However, multiple leave operations and/or failures may cause the cluster size to fall below the \( \text{minClusterSize} \) threshold. In that case, individual nodes of that cluster initiate a disbanding procedure and relocate themselves to other existing, and more stable, clusters.

A gossip-based anti-entropy mechanism, in which elements of each cluster periodically exchange messages among themselves, is employed to ensure that cluster members converge on a consistent view of the cluster membership, despite concurrent joins and failures in the system. Furthermore, this process is used to provide a minimal amount of coordination required to increase the diversity of external links maintained by different cluster members.

Note that Overnesia does not offer any DHT support. As a result, the way links are established among clusters does not take any sort of routing requirements into account, other than attempting to maximize the connectivity of the network.

We describe how the unstructured layer of Rollerchain differs from Overnesia in Sec.4.2.1.

3.2 Chord

Chord [25] is a widely known DHT, used for resource location, that organizes the nodes in a ring-like topology. It places objects in different locations on the identifier ring using consistent hashing. Its ring-based geometry was shown to be one of the most resilient DHT geometries [8].

Chord’s ring is created by sorting nodes by their identifier modulus the size of the identifier space. Its maintenance is mostly proactive, such that each node keeps a predecessor and a successor node through periodic maintenance routines. More specifically, each node \( N \) periodically queries its successor \( S \) in order to obtain the predecessor \( P \) of \( S \) (naturally, in a stable scenario, we should have \( P = N \)); should \( P \neq N \) be a node with an identifier in the interval \( )N_{id}; S_{id}[ \), \( N \) will then switch its successor pointer to node \( P \). After this update, \( N \) informs \( P \) that it is now \( P \)’s predecessor. This simple routine allows the ring to converge and remain connected even in face of concurrent entry and exit of nodes.

In order to improve Chord’s reliability in a scenario with node failures, the typical approach is for each node to maintain a set of \( f \) alternative successors. This allows the node to contact one of the alternative successors should the current one fail. This mechanism may be implemented by having each node pass its own list of \( f - 1 \) successors to its predecessor in reply to the “Predecessor?” query (in a cascading fashion); or by having each node actively requesting its alternative successor’s successor list, leading to a faster update of the alternative successors list when under greater system dynamics.

Even though Chord’s ring would suffice for any node to reach any other node in the overlay, routing messages exclusively through this ring would be very inefficient. Thus, Chord’s protocol includes an efficient routing mechanism: each Chord node maintains a \textit{Finger Table}, from which it selects the closest node on the ring to route messages towards their destination. As the Finger Table contains pointers to nodes which are at exponentially increasing distances from the node’s position in the ring, this mechanism allows Chord to route messages in \( \log(n) \) network hops (since the distance to the destination can be halved with each hop).

We describe how the structured layer of Rollerchain differs from Chord in Sec.4.2.2.

4 Rollerchain Operation

In this section we describe the operation of Rollerchain in detail. We start by introducing some notation, that will help to make the description clearer. Then we detail the operation of each layer of Rollerchain. Finally, we discuss some aspects of the protocol where the operation of the unstructured and structured layers is more entwined.
4.1 Definitions and Basic Operation

We opted to preserve the nomenclature of the original Chord paper, where each peer is denoted a node. Therefore, in Rollerchain, each peer is called a physical node, or pnode, for short. The unstructured component of Rollerchain aggregates pnodes in groups. Pnodes that belong to the same group cooperate to behave collectively as a logical virtual node, or vnode. A vnode is a fully connected group of pnodes with a fluctuating size around \( R \) (the replication factor) which act as a single virtual node at the structured layer. To facilitate the coordination among pnodes in the same group, an anti-entropy gossip-based protocol is executed among them. The anti-entropy protocol allows pnodes to exchange information required for the operation of Rollerchain (which we will incrementally describe), including the membership of the group and key/value pairs maintained by pnodes. Also, to simplify coordination among members in several Rollerchain procedures, the group member with the lowest process identifier is elected as the vnode leader. Occasionally, it may happen that more than a single peer sees itself as the pnode leader. This does not affect the correctness of Rollerchain, as the leader is only used to lower the signaling costs of the algorithms. Similarly, if no node sees itself as the leader, this only delays the progress of the algorithms until the anti-entropy procedure enables one node to see itself as the leader.

When a virtual node grows to a large size, it is halved into two vnodes with the same size. The division creates a new vnode, which joins the structured layer (the first vnode in the system being the obvious exception: it serves as bootstrap for the DHT and starts as part of it).

Virtual nodes establish virtual links among themselves to create a logical ring. A virtual link between vnode \( A \) and \( B \) is materialized by establishing links among pairs of pnodes \((a_i, b_j)\) where \( a_i \in A \land b_j \in B \). The algorithm for establishing and maintaining virtual links is described later in the text.

4.2 Layer Operation Details

In this section we describe the high-level operation of each layer. Section 4.3 presents the details on how to integrate both layers.

4.2.1 The Unstructured Layer

The unstructured layer of Rollerchain is an overlay composed of clusters of fully connected nodes, where each cluster stores key/value pairs replicated by all of its members. This replication not only increases the resilience of data, but also allows pnodes to share the load of answering queries for objects stored the associated vnode. The operation of this layer is inspired by that of Overnesia [18], but with a significant number of changes in order to satisfy the Rollerchain’s requirements for object placement and load balancing.

When joining the unstructured layer, a pnode starts a random walk in the network, which probes suitable vnodes to join. As one of Rollerchain’s goals is to balance the load among the virtual nodes that compose the system, the new pnode chooses the most heavily loaded virtual node found in the random walk to join the network. This mechanism causes heavily loaded virtual nodes to attract new nodes in order to share their load.

When a vnode leader detects that its vnode has become too large, it coordinates a division procedure to split it into two vnodes. This division serves not only to reduce the costs of replicating data among members of the vnode, but also to reduce the number of objects it stores. When a virtual node is divided, one cluster keeps the identifier of the original virtual node (to avoid inducing artificial churn in the structured overlay) and the other generates an identifier that allows it to keep the remaining half of the key/value pairs (this is discussed further ahead). Note that, as a result of this strategy, if the keys are not uniformly distributed in the identifier space, node identifiers will follow a similar distribution and consequently will also not be uniformly distributed in the address space.

On the other hand, if the vnode leader detects that its vnode’s size is dangerously bellow the target replication degree, it prevents objects from being lost by integrating its vnode with its successor vnode. The vnode leader obtains its successor vnode’s membership and broadcasts it to its vnode neighbors, which then change their cluster
identifier accordingly. Even though this operation increases the number of objects stored at the successor vnode, the pnodes from the small vnode also join in this effort, attenuating the increase in load. Notice that this would not occur should the small vnode be merged to any other vnode.

4.2.2 The Structured Layer

The structured layer of Rollerchain is a double-linked ring composed of virtual nodes provided by the unstructured layer. As described earlier, the vnode identifiers are selected as to promote the load balancing of the number of objects stored at each vnode. Therefore, vnode identifiers are not uniformly distributed in the identifier space. In the original Chord paper, node identifiers are assumed to be uniformly distributed in the identifier space. Thus, its routing mechanism becomes inefficient if applied to Rollerchain.

In order to provide efficient routing in Rollerchain’s identifier space under such conditions, our routing mechanism achieves \( \log(n) \) routing complexity by relying on routing tables which provide “large jumps” across the identifier space, and on the identifier ring to perform the final hops to the destiny vnode. Each vnode in Rollerchain has a virtual link to its immediate successors and one routing table with \( \log(n) \) rows, with virtual links to distant vnodes in the ring. As it happens in DHTs such as Chord [25] or Viceroy [19], each row in the routing table represents an exponentially larger jump than the previous row. However, whereas in these solutions the larger jumps refer to the identifier space, in Rollerchain each row in the routing table represents an exponentially larger jump in the number of vnodes on the ring. Notice that when vnodes are distributed uniformly in the ring, both schemes create similar routing tables. More details on how the routing tables are populated are given in Sec.4.5.

It is important to notice that even though this routing scheme ensures \( \log(n) \) routing in our system, other routing mechanisms, such as those based on Skipnets [10] or the work presented in [13], could also be used.

4.3 Layer Interoperation

The previous section described the functionality provided by both layers required to accomplish Rollerchain’s goals. This section details how both layers are integrated, particularly how virtual links between vnodes translate into links between physical nodes, how these links are maintained, how the objects are stored in the correct vnodes and how messages are routed between virtual nodes.

4.3.1 Virtual Link Creation

As described before, virtual nodes cooperate to create a Ring-based DHT. To preserve the logical ring, and to support efficient routing, each virtual node maintains a virtual link to other virtual nodes in the ring. More precisely, a virtual link needs to be maintained to each different virtual node in the finger table (including the successor).

Virtual links between two vnodes, say \( A \) and \( B \), need to be materialized by links between individual members of these vnodes (Fig. 1a). For fault-tolerance and load balancing, these links should be distributed evenly among these members. For instance, consider that virtual node \( A \) has \( n \) members and virtual node \( B \) also has \( n \) members.

![Figure 1. Join and virtual links](image-url)
It would be highly undesirable to have a single pnode \( a_k \) to have \( n \) links to each member of \( B \) (Fig. 1b). It would be equally undesirable if all nodes in \( A \) establish a link to the same pnode \( b_k \) in \( B \) (Fig. 1c). The ideal solution would be to have each pnode \( a_i \in A \) to have a different link to a different pnode \( b_i \in B \) (Fig. 1d). In this way, each pnode would be required to maintain a single link and the virtual link would be materialized by \( n \) independent physical links. Additionally, this ensures that \( n - 1 \) physical links between \( A \) and \( B \) would still be alive after a single pnode failure.

In order to quickly achieve an even distribution of links among members of a cluster, the vnode leader (the pnode with the smallest identifier), say \( a_0 \), coordinates a procedure of virtual link creation. This procedure is used when a vnode leader updates the virtual links of its vnode, particularly during vnode division and vnode finger update (which will be described later). We assume that when the virtual link creation procedure is invoked, \( a_0 \) is aware of at least one member \( b_k \in B \). The leader \( a_0 \) starts by requesting \( b_k \) to return the current snapshot of \( B \)'s membership; we recall that there is an anti-entropy procedure running inside each vnode that keeps such information up-to-date.

Knowing the full membership of \( A \) and \( B \), the leader \( a_0 \) establishes links among \( A \) and \( B \) in a round-robin manner, until all pnodes of \( A \) and \( B \) are connected to some pnode in the other vnode (note that Fig. 1 is an over simplification of reality, as \( A \) and \( B \) generally do not have the exact number of nodes). Finally, this mapping is propagated by anti-entropy to all members of \( A \) (in fact, the mapping does not need to be explicitly disseminated; since the algorithm is deterministic, the leader only propagates the membership snapshots that should be used as seed for determining the assignment). Each individual pnode \( a_k \) then initiates the establishment of links to its corresponding pnodes in \( B \).

4.3.2 Virtual Node Maintenance

The membership of a virtual node may change as pnodes join or leave the overlay. As it is crucial for the connectivity of Rollerchain that its ring remains connected despite system dynamics, such changes require adjustments to the link assignments that materialize the virtual links maintained by the vnode to its successor and predecessor nodes.

Consider first the result of a join operation on virtual node \( A \) by pnode \( a_k \). The pnode may help in materializing the virtual links maintained by the virtual node \( A \). To this purpose, and for each virtual link, pnode \( a_k \) will attempt to “alleviate” the load of some other members of \( A \), by serving as endpoints of some of their links. This can be achieved as follows.

As noted before, members of the same vnode execute an anti-entropy protocol among themselves. To help in the maintenance of a virtual link, the information exchanged between pnodes includes the number of forward and backward links maintained by each pnode for a given virtual link. Therefore, \( a_k \) can select a node that has more forward or backward links than the others and takes one of its links. On the other hand, if any other pnode with more than one link learns about \( a_k \), it moves one of its links to \( a_k \) instead. For instance, assume that some other pnode \( a_l \) has two links to \( b_l \) and \( b_k \): \( a_l \) keeps the link to \( b_l \) and closes the links to \( b_k \) and, in turn, \( a_k \) setups a link to \( b_k \). On the other hand, if the number of links is perfectly balanced, the new node just creates a redundant link at random.

Consider now the case where a pnode leaves a vnode. The physical links established by that pnode will break. This means that the pnodes on the other endpoint of those links will perform the link re-distribution described in the previous paragraph.

Finally, consider the case where multiple join and leaves occur in succession. If joins and leaves are perfectly interleaved, a link is lost in each leave but a new link is created in each join, and the number of links that materializes the virtual link between two vnodes is kept constant. However, if bursts of leaves or joins occur, the balance

---

2Note that there is no need to adjust the remaining virtual links in the finger table as those are non-essential to maintain overlay connectivity.
may no longer be preserved.

If as a result of the anti-entropy procedure, the vnode leader detects a scenario of heavy imbalance in the link distribution for some virtual link, it triggers a re-balance procedure. This rebalance procedure consists of sending the current physical link to virtual link mapping to the endpoints of the physical links causing imbalance so that the pnodes involved create links to pnodes with fewer incoming links.

4.3.3 Virtual Node Division

As a result of multiple joins, some vnodes may become very large, which generates unnecessary data redundancy. By splitting the vnode into two smaller vnodes one can reduce the signaling cost of maintaining the groups as the key/value pairs of the original vnode become divided between the two resulting vnodes, reducing the information that needs to be exchanged by anti-entropy.

Vnode division is performed in such a way that the resulting two vnodes have equal size and equal load. For this purpose, one of the vnodes preserves the identifier of the original vnode and the other vnode becomes its predecessor, assuming an identifier that causes the new vnode to become the owner of half of the key/value pairs of the original vnode.

Virtual node division is controlled by the leader of the original vnode, that sends a message to all vnode members with the identifiers of the new vnode, the membership of each vnode, and a new assignment for the virtual links maintained by each vnode. Note that this division procedure ensures that the new vnodes are neighbors in the DHT.

4.3.4 Virtual Node Merge

As we have noted before, if the rate of leaves/failures is the same as the rate of joins, new pnodes will probably replace failed pnodes in a vnode. In a steady-state scenario, vnodes are stable.

However, as a result of multiple leaves and failures, the size of a vnode may become too small, below the desired replication level for the key/value pairs. This happens when a vnode has very few keys while other vnodes remain heavily loaded in the system. Since the join procedure looks for vnodes that are highly loaded, in order to offer a better load balancing, a vnode with few keys (and therefore, not heavily loaded) may not be selected by new pnodes.

When a vnode becomes too small, it is merged with its successor. This ensures that all processes that are part of the two merging vnodes retain their key/value pairs. The anti-entropy in the resulting vnode will ensure that those keys will be later replicated by the remaining members of that group. Similarly to the procedures above, the merge is started by the leader of the merging vnode.

4.4 Key/Value Pairs Replication

The replication of key/value pairs among the members of a vnode is performed using a combination of eager and lazy replication schemes.

When a key/value pair is inserted in a vnode, eager replication is used. The pnode that receives the request uses a best effort multicast primitive to replicate the pair among \( R \) other members (\( R \) is the target replication degree).

Subsequently, replicas are maintained using a lazy replication scheme that leverages on the anti-entropy protocols executed among vnode members. To this end, processes include the keys that they own in the anti-entropy exchange. If (as a result of an anti-entropy exchange) a pnode discovers that it misses some key/value pairs, it simply downloads them from its vnode neighbor.

To maintain the correct keys at each vnode, each pnode periodically checks the keys it stores. When a pnode detects it is storing data for which the keys should be in its successor (the keys are between its vnode identifier and its successor vnode identifier) or its predecessor (the keys are not between its identifier and its predecessor
identifier), it transfers the content to the correct vnode and deletes it locally. Notice that, as the replication of contents in each vnode is handled by the anti-entropy protocol, the data is sent to a single pnode, as it will be subsequently replicated in a lazy manner among the members of its vnode.

As it happens with other replication mechanisms for which there is no master copy of the data [5], this mechanism does not provide strong consistency among the replicas. Writing to only one node and pushing the data in a lazy scheme may cause lookups for data to fail, even though the data is already stored in the system. Although faster writes improve the performance of the system, such behavior might be undesirable. To avoid such inconsistencies, the confirmation of the write operation may be delayed until the data is propagated to all pnodes in the virtual node, which would ensure that after each write all replicas would be able to answer lookups.

4.5 DHT routing

DHT routing is performed in Rollerchain closely following the Chord routing with some twists. Logically, lookups follow virtual nodes, using the virtual links maintained in the virtual node’s finger table. In reality, lookups are implemented by individual pnodes. When the lookup arrives at a pnode, it uses its own finger table to select a pnode to be the next hop for the lookup. If the pnode cannot contact the next hop, the lookup operation does not need to be aborted. Since there is a virtual link connecting its own vnode with the next hop vnode, and this link is materialized by multiple physical links, the lookup is re-routed to another neighbor, which attempts to forward the lookup using one of its own links. This is similar to the re-route mechanisms used by DHTs that have routing tables with alternate paths (such as, for instance, Pastry [23]). Due to the redundancy in the virtual link maintenance, it becomes very hard to undermine routing in Rollerchain.

The finger tables are periodically updated by the leader node, which updates one row at a time, first locating the correct vnode and then broadcasting the virtual links to the other pnodes in its vnode, similarly to what was described in Sec. 4.3.1. As it happens in Chord, each line in a vnode’s routing table represents a (virtual) node at an exponentially increasing distance. However, based on the techniques introduced in Chord# [24], our scheme does not rely on the vnode identifiers to populate routing tables. The objective is to create routing tables in which row \(i\) represents a vnode which is \(b^i\) hops away in the ring (where \(b\) is a configuration parameter). For simplicity of exposition, \(b = 2\) will be used in the following example.

The vnodes for each row in the finger table are found by recursing in other vnode’s finger tables. Each row is populated using information from the routing table of the vnode in the previous row. The intuition behind this scheme is that if row \(i\) of vnode \(A\) represents vnode \(B\), located \(X\) hops away, then row \(i + 1\) represents a vnode at \(X \times 2\) hops, which corresponds to the vnode at row \(i\) of \(B\)’s routing table. Thus, the vnode for row \(i + 1\) of vnode \(A\) can be found by sending a message with \(TTL = 1\) to vnode \(B\), to be routed to the vnode in row \(i\) of \(B\)’s routing table. Row 0 is populated with the successor vnode, creating a basis for recursion. This method can be generalized: row \(i\) of a vnode’s routing table is populated with the vnodes reached by performing \(b\) hops in the ring using the pointer in row \(i - 1\) of each hop’s routing table. Notice that this mechanism involves a constant number of messages to update the vnode’s routing table, whereas Chord’s routing table update procedure has a complexity of \(\log(n)\).

5 Evaluation

In this section we present experimental results that validate the design of Rollerchain. We have performed extensive experimental work using the Peersim simulator [11] event-based engine. All results reported in this paper were taken using a system composed of 10,000 nodes. The system was populated with 500,000 objects. In order to extract comparative measures, we used Chord combined with the neighbor replication scheme described earlier (henceforth referred to only as Chord) as this is the most cost-effective replication scheme [16]. All reported results are an average of, at least, 5 independent runs of each experience.
The goal of our experimental work is to evaluate the performance of Rollerchain in terms of fault-tolerance, load balancing and, due to the inherent costs of combining overlays, communication overhead.

5.1 Experimental Parameters

In all experiments reported in the paper we used the following protocol configurations. Chord used a replication factor of 6, causing each data item to be replicated in the 3 successors and 3 predecessors of the key owner. Rollerchain was configured with a minClusterSize of 3 and a maxClusterSize 8, which results in vnodes to be composed, on average, by 7 pnodes. This means the average replication factor of Chord is the same as Rollerchain’s, and provides the most balanced performance configuration for Chord’s neighbor replication, in terms of fault-tolerance and load balance.

Furthermore, to ensure fairness in the comparison, the parameter $b$ of Rollerchain was set to 2, and its routing table size was set to 11. This allows both protocols to have a similar number of (distinct) fingers, creating routes with the same number of hops (15 on average). In addition, the anti-entropy mechanism of Rollerchain was executed as often as the replication maintenance routine of Chord.

5.2 Fault-Tolerance

To evaluate the resilience of both systems, we have conducted experiments as follows. Overlays were initialized by having nodes join the system one at a time. After a stabilization period, we induced simultaneous failures to a fraction of randomly selected nodes. The goal of the experiment was to assess the percentage of objects that were reachable after the failure. To that end, after the failures we triggered 2 queries per key (1,000,000 queries in total) at random nodes. Fig. 2 summarizes the results.

As expected, as the fraction of concurrent failures increases, the number of reachable objects decreases for both systems. However, Fig. 2 shows that the number of reachable objects after failures is consistently higher in Rollerchain when compared with Chord.

In particular, when 10% of nodes fail simultaneously, 20% of queries performed in Chord were unsuccessful, while in Rollerchain all queries were successful (i.e., 100% of objects were reachable). We have experimentally determined that in this scenario, only 0.05% of the objects were definitely lost in Chord (i.e., were not stored in any correct node); the higher percentage of unreachable objects is due to the fact that, when 10% of nodes fail simultaneously, the Chord ring breaks, which leads to incorrect routing of queries. Rollerchain does not suffer...
from this effect, as the ring remains connected as long as, at least, one of the pnodes in each vnode remains connected to a pnode on the successor vnode.

Furthermore, the replication scheme employed in Rollerchain simplifies the task of recovering from node failures, reducing the probability of data becoming lost in the process.

5.3 Load Balancing

In order to assess the load balancing capacity of Rollerchain, we have conducted experiments where both Rollerchain and Chord operated in steady state (i.e., no join or leave operations occurred). We assigned keys to 500,000 objects following two distributions. We tested both systems using a uniform random and a zipf distribution (the zipf distribution had a 1.25 skew). The uniform random distribution provides a baseline of comparison, while the zipf distribution emulates a particular scenario where a portion of the identifier space becomes overloaded with an increased number of objects.

In these experiments we triggered 1,000,000 queries at random nodes. Two queries for each object were performed to avoid a bias in this load. In these experiments we measured the number of queries routed and processed (i.e., queries to which a node returns an answer) by each individual peer.

Figure 3 presents the distribution of nodes over the number of queries processed and routed for both considered distributions. The first observation that can be made is that Rollerchain presents, for both operations and key distributions, a much more balanced behavior which is indicated by the narrower distribution of load around the
average value. This supports our claim that the existence of vnodes allows to improve the load balancing.

In particular for the results concerning the load imposed on nodes for processing queries (Figs. 3(a) and 3(b)), one can note that in Chord a large fraction of nodes (10% for the uniform random distribution and 80% for the zipf distribution) do not process a single query. Although some nodes in Rollerchain process few queries as well, it is a much smaller fraction. This happens because some nodes in the system store few objects. However, as Rollerchair relies in vnodes, the number of pnodes effectively in this situation is much lower, allowing Rollerchain to better leverage the available resources in the system.

Notice that Rollerchain exhibits a more balanced query routing load distribution when compared with Chord (Figs. 3(c) and 3(d)). This happens because, in steady state, each pnodo of a vnode in Rollerchain has a similar number of inbound links. Therefore, the probability of any pnodo in a vnode to receive a query is the same. This is not true in Chord, where the number of inbound links of a node depends on the distribution of nodes’ identifiers.

<table>
<thead>
<tr>
<th></th>
<th>Queries Processed</th>
<th>Queries Routed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>AVG</td>
</tr>
<tr>
<td>DHT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rollerchain</td>
<td>8</td>
<td>100</td>
</tr>
<tr>
<td>Chord</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rollerchain</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>Chord</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 1. Load average (AVG), minimum (MIN) and maximum (MAX) for Uniform (U) and Zipf (Z) distributions.

Table 1 depicts the average of loads for both queries processed and routed across all nodes, as well as the minimum and maximum values observed in individual nodes. Results are presented for both the uniform random and zipf key distribution. As discussed previously, the minimum number of queries processed by a single node is zero for both systems due to some nodes storing no objects.

Concerning the maximum number of queries routed and processed by a single node, results show that Rollerchair presents values which are closer to the average, indicating a better load balancing. In fact the most loaded node in Rollerchain routes only 5.5 times more queries than the least loaded node, whereas in Chord it is at least 47 times more. This shows that some nodes in Chord are forced to perform (especially for a zipf key distribution) much more work than other peers in the system. Such nodes have a significantly higher probability of becoming overloaded, and failing.

Contrary to Chord, Rollerchain’s load distributions are not significantly affected by the zipf distribution. This happens because Rollerchain’s vnode identifiers are created based on the load of each vnode and its routing relies in a mechanism that is independent of the skew in such identifiers. This makes Rollerchain suitable for supporting applications that use non-uniform distributions of keys to take consideration, for instance, locality in overlay object placement.

5.4 Communication Overhead under Churn

In order to show the impact of combining two overlays to create Rollerchain, this section presents experimental results for the communication overhead of each system in a churn scenario [27]. To this end, we have conducted experiments similar to the ones presented in Section 5.2, but instead of inducing failures we have induced a period of churn where, for 10,000 consecutive simulation steps, we concurrently remove and add $C$ nodes ($C$ is dubbed churn rate). We performed experiments with churn rates of 1, 10, and 100 nodes per step.

Signaling overhead has 3 main causes: i) maintaining the topology; ii) moving objects among nodes (to ensure that objects are stored at the correct key owner and have enough replicas); and, iii) replication maintenance...
protocols that are used to monitor the replication degree. This last one is the same for both Rollerchain and Chord, as both systems employ these protocols at a similar rate, and keep the same number of objects.

<table>
<thead>
<tr>
<th></th>
<th>C = 1</th>
<th>C = 10</th>
<th>C = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rollerchain</td>
<td>20.71</td>
<td>2.229</td>
<td>0.230</td>
</tr>
<tr>
<td>Chord</td>
<td>25.50</td>
<td>2.580</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 2. Maintenance overhead per node under different churn rates (in MB).

Table 2 presents the communication overhead per node due to maintaining the topology for both systems. The results show that Rollerchain has a lower maintenance overhead. This happens due to the fact that pnodes in the same vnode share routing information among them. This enables such nodes to coordinate their efforts in maintaining such information up-to-date, avoiding the routing of redundant messages to, for instance, maintaining their fingers.

Additionally, as the churn rate increases, there is also a decrease in the topology maintenance cost per node. This is due to a larger number of nodes sharing the maintenance costs, despite the increase in membership dynamics. The total maintenance costs of the overlays can be obtained by multiplying the values in Table 2 by \( N \) and \( C \) and then doubling the result to account for the initial nodes. Even though this result’s order of magnitude is preserved for all values of \( C \), there is a slight increase in this cost, since there is an increase in the number of join requests and in the number of vnode merges, for Rollerchain. Notice that for for \( C = 100 \), the results for Chord are omitted since its ring breaks and the overlay loses connectivity, thus the results are meaningless.

<table>
<thead>
<tr>
<th></th>
<th>C = 1</th>
<th>C = 10</th>
<th>C = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rollerchain</td>
<td>377</td>
<td>380</td>
<td>406</td>
</tr>
<tr>
<td>Chord</td>
<td>1205</td>
<td>1070</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 3. Number of objects transferred between pnodes, under different churn scenarios, per node joining the network

Table 3 presents the number of objects that were transferred among peers in the system, per node joining the network, for each tested churn rate. Rollerchain is required to transfer fewer objects than Chord independently of the churn rate. This happens due to the fact that vnodes offer a “protection” against membership dynamics in general and churn scenarios in particular. Notice that in Chord, when a node joins, it takes a random identifier. With high probability, this will require a number of objects to be transferred to this node from his successor, as it becomes key owner for those objects. Furthermore, a node failure forces the creation of replicas for all its stored objects to ensure a constant replication degree. In Rollerchain, the replication degree is flexible, thus, a new pnode will be integrated in a pre-existing vnode, possibly replacing a failed node. This way, the joining pnode may become a replica for the objects previously stored in the failed node, avoiding one transfer per object. This effect is particularly noticeable with low churn rates, as it is more probable that joining nodes replace those that failed: for each object transferred in Rollerchain, Chord transfers one additional object. Notice that if a new node always replaces a failed node, the minimum number of objects transferred per node is \( N_o \times R \) (where \( N_o \) is the average number of objects per node and \( R \) is the replication degree). For the configuration used in the experiments, this value would be 350. Notice that Rollerchain is far closer to this value than Chord, due to the reasons presented previously.

When \( C \) increases, three phenomena are observed: more merge operations occur in Rollerchain (increasing the number of objects transferred for \( C = 10 \) and \( C = 100 \)); more nodes fail before any objects are transferred to them; and more keys are lost, reducing the number of keys in the system. The latter two phenomena lead to the decrease of the number of objects transferred per node in Chord for \( C = 10 \).
6 Conclusions

In this paper we proposed Rollerchain, a novel DHT that offers high availability of stored data in an efficient manner. Rollerchain is based on the combination of unstructured and structured overlays to generate a DHT of virtual nodes, where each virtual node is materialized by a set of physical nodes. Objects stored in the DHT are therefore replicated among physical nodes of each virtual node.

Experimental results presented in the paper show that Rollerchain, when compared with Chord using a common replication scheme, achieves an increased resilience to failure, conserving up to 7 times more objects when subjected to massive failures, leading to an increased data availability. Furthermore, Rollerchain’s communication overhead under churn is lower than Chord’s, transferring less than half the objects, at a lower signaling cost in all scenarios considered. Finally, the unique topology of Rollerchain allows to improve the load balancing of routing and processing queries, achieving a narrower distribution of load around the average value when compared with Chord.

As future work, we plan to explore new data persistency models to apply in Rollerchain. The fundamental idea is to devise mechanisms that enforce distinct persistency properties on stored data, allowing applications to specify requirements of data stored in the DHT for each particular object.

References