GosSkip, an Efficient, Fault-tolerant and Self Organizing Overlay using Gossip-based Construction and Skip-lists Principles

Rachid Guerraoui, Sidath Handurukande, Kévin Huguenin, Anne-Marie Kermarrec, Fabrice Le Fessant, Rivière Étienne

GosSkip, an Efficient, Fault-Tolerant and Self Organizing Overlay Using Gossip-based Construction and Skip-Lists Principles

Rachid Guerraoui, Sidath B. Handurukande*
School of Computer and Communication Sciences,
EPFL, Switzerland

Kévin Huguenin
ENS Cachan/IRISA
Rennes, France

Anne-Marie Kermarrec
INRIA/IRISA,
Rennes, France

Fabrice Le Fessant
INRIA-Futurs/LIX,
Palaiseau, France

Étienne Rivière
Université de Rennes 1/IRISA,
Rennes, France

Abstract

This paper presents GosSkip, a self organizing and fully distributed overlay that provides a scalable support to data storage and retrieval in dynamic environments. The structure of GosSkip, while initially possibly chaotic, eventually matches a perfect set of Skip-list-like structures, where no hash is used on data attributes, thus preserving semantic locality and permitting range queries. The use of epidemic-based protocols is the key to scalability, fairness and good behavior of the protocol under churn, while preserving the simplicity of the approach and maintaining O(log(N)) state per peer and O(log(N)) routing costs. In addition, we propose a simple and efficient mechanism to exploit the presence of multiple data items on a single physical node. GosSkip's behavior in both a static and a dynamic scenario is further conveyed by experiments with an actual implementation and real traces of a peer to peer workload.

Keywords: Gossip-based protocols, self-organization, data structures, skiplist

1 Introduction and Background

Peer to peer networks are distributed networks where no centralization is used, especially for locating and querying data. The desired properties of such systems include, but are not limited to, load balancing among participating nodes, resilience in face of churn and high expressiveness of query mechanisms. Above all, a peer-to-peer network has to be scalable, and this is particularly important for the efficiency of search algorithms and construction costs of overlays.

Generic P2P overlays [15, 16, 17, 21] usually provide the functionality of a distributed hash table (DHT). They can be used to efficiently locate an object specified by a key (*e.g.*, a filename) within a large set of nodes. They ensure load balancing in terms of hosted objects per node and scalability but the search efficiency (usually in $O(\log(N))$, N being the number of nodes in the overlay) is a result of the inherent tradeoff between efficiency and expressiveness in distributed systems. The fact that the only querying interface is exact-match and that hashing is used to determine the placement of data to nodes lose the initial ordering of objects. As a result, the possibility of richer query mechanisms such as range queries or nearest neighbor queries is restrained, or exhibits crippling costs.

Some work has been done to propose structured overlays that keep this ordering on objects names. Mercury [5] stores objects described by a set of attributes. To each attribute corresponds a ring, on which objects ordering is preserved, thus providing a support for range queries. To ensure load balancing, however, distributed node-count histograms gathering of the naming space have to be performed, and the behavior of this scheme in face of a high churn rate remains unclear. Another set of distributed data structures that avoid hashing to ensure load balancing, and that permit both query efficiency and an expressiveness that is higher than DHTs, are based on the Skip-List structure. A Skip-List is a doubly-linked list where objects with subsequent names are linked on the first level (level 0). Some pointers at each object permits a Skip List to resemble a balanced tree, forming increasingly sparse linked lists at each level. In a *perfect Skip list*, a level h pointer at a node traverses exactly 2^h nodes, while for Skip-Nets [11] or probabilistic Skip-lists [14], the structure resembles a randomized balanced tree, with $O(logN)$ insertion and querying costs w.h.p., and a similar top-down querying strategy. However, a Skip List uses only one linked list per level, thus implying a high load on nodes participating on upper levels, and penalizing the whole structure upon deletion of such nodes. While this was not an issue for centralized data structures, these two points are a concern for a peer-to-peer data structure. Skip-graphs [3] are similar to Skip-Lists but use several concurrent linked lists for levels > 0, to balance the

*This work was done when the author was affiliated with EPFL

load of query propagation and ensure fault resilience. Skip-webs [2] are based on the same concepts for multidimensional data. They provide a $O(\log N/\log\log N)$ query cost in single dimension and $O(\log N)$ in multiple dimensions. However, all these distributed data structures rely on a deterministic construction of the overlay (and on leaving peers using a fair leaving procedure) and need additional mechanisms to repair themselves in presence of node failures.

On the other hand, gossip based protocols have proven efficient for the construction and maintenance of distributed systems. Gossip techniques have been successfully used in many settings [12] including database maintenance [8], multicast [6], routing tables management [19] and attribute-based publish/subscribe systems [10]. Their application to the maintenance of overlay networks [7, 18, 20] has shown their ability to gracefully deal with churn, building self-organizing networks able to resist to the loss of a large part of the network without collapsing. Gossip techniques enable to keep overlay construction algorithms simple yet efficient, without the need for explicit overlay reconstruction mechanism in case of node departures.

Contributions: In this paper, we propose the design and implementation of GosSkip, a distributed data structure providing a better tradeoff between query expressiveness and query efficiency than DHTs. As in the above systems, GosSkip has the important property of preserving content locality in the semantic space. GosSkip links *applications objects* (rather than computing entities) in a structure that eventually resembles a set of exact balanced trees, while balancing the load of queries uniformly among peers. GosSkip is built using a gossip-based protocol and is resilient to high churn rates. Overlay construction messages can be piggy-backed on top of *heart beat* messages which are in any way present in most distributed systems. An evaluation of GosSkip behavior both in a static and dynamic scenarios is performed using a deployed implementation of the protocol and a real workload of a file sharing application. The paper is organized as follows: basic principles are given in Section 2 and the details of algorithms in Section 3. Section 4 gives evaluation results of the protocol using real traces and in dynamic settings. Section 5 proposes an approach to leverage the presence of multiple peers on a physical node and improve routing and robustness.

2 GosSkip at a Glance

GosSkip is a structured peer-to-peer overlay linking application objects in a distributed data structure. We will thus discuss indifferently of data elements as being *peers* in the overlay. Each peer has a name that depicts its semantic for the application. The only necessary property is that these names follow a deterministic total ordering. Then, the position of a data element is fully determined by its name. For the remaining of the paper, N will denote the number of peers.

GosSkip organizes peers in such a way that they form a sorted doubly linked list (or ring). Once a peer has located himself in the sorted list, links to level-0 neighbors are straightforward to implement. For each peer, additional longer links l_i, for levels $1 \leq i \leq \log_k(N)$ skip over k^i peers. These links form a set of perfect skip lists, providing the functionality of balanced k-ary tree: $O(\log_k N)$ search and insertion costs. In a single skip list, a peer has a probability of being part of each level i equal to $\frac{1}{2^i}$. This can lead to a high load of query propagation on the top-level nodes. In GosSkip, this is not the case as every peer is present in all $\log_k(N)$ levels and has an equal probability of being part of any query path. The load of query propagation is thus balanced uniformly among all peers.

Peer management In GosSkip, a peer is associated to one data element. Its management is the responsibility of the *physical node* (computing entity) that published this element. For the sake of clarity, we assume for now that there is a one-to-one mapping between peers and physical nodes, and we come back to this issue in Section 5. The mapping of peers to node can be modified to move application objects according to some heuristics (physical proximity, communication patterns) but such techniques do not modify the algorithms and their description is out of the scope of this paper.

3 Overlay Construction

In this section, we describe the mechanisms used to create and maintain the GosSkip overlay.

Joining and leaving the network. When a peer wants to join the overlay, it simply sends a `join` message to a peer already participating in the system as in most p2p algorithms [15, 16]. The join message progresses in the sorted list until the peer location is found and the peer is then inserted still preserving the sorted order. Then, as gossip messages are exchanged in the system, the peer is gradually integrated in the upper level lists. When a peer wishes to leave the system, it just stops gossiping messages and will gradually be discarded from the neighbor lists. Besides, a peer failure is detected by its neighbors which in turn remove it from their list of neighbors triggering the creation of a new level 0 link instead.

Establishing long links. GosSkip relies on gossip messages to construct long links. This message can be piggy-backed over maintenance or applications messages that already exist in the network. To ease description, we will denote as *right hand* and *left hand* neighbors on level i the peer that directly follows (respectively precedes) a peer in a list. Each peer periodically sends gossip messages to the peers on the right hand side first, as shown in Figure 1.

Each message consists of a collection of entries. Each entry is composed of an identifier (*e.g.*, IP address; Id1 in

Figure 1), its associated data item (d1 in Figure 1) and a counter. As in all gossip-based protocols, time at each peer is divided in periods of fixed duration. Each new period, each peer forwards a subset of the entries it receives during the last period. It also adds an entry with its own id, data item and a counter set to zero along with the forwarded entries. Each peer increments the counter of every entry it receives before forwarding it. Once received, if the counter at peer *p* is equal to k-1 (k is a configurable system parameter, gives the number of peers each link skip over, and in Figure 1 we consider k as 2) the entry is not further gossiped (by simply removing it from the message) and peer *p* adds the peer associated with the removed entry together with the information associated with that entry to its neighbor list (*e.g.*, as in Step 3 of Figure 1, peer 3 adds Id1 to its neighbor set together with d1). Peers have neighbors on right and left hand sides, maintained respectively in rightward- and leftward-neighbor lists. These neighbor lists represent the routing state of each peer.

Note that, as a result of removing entries from messages, once the counter reaches parameter k-1, the size of the gossip message (in terms of entries) is limited to k entries irrespective of the network size. At the end of this process, peers know about other peers that are k hops away, on the left hand side. These steps are depicted in Figure 1 where k is set to 2. Immediate neighbors (i.e., one hop away) peers are level-0 neighbors and neighbors that are k hops away are level-k neighbors. For example, as in Figure 1, node 1 and 2 are level-0 neighbors while node 1 and 3 are level-1 neighbors. Likewise, each message is associated with a level representing the level of the neighbors between which the message is sent.

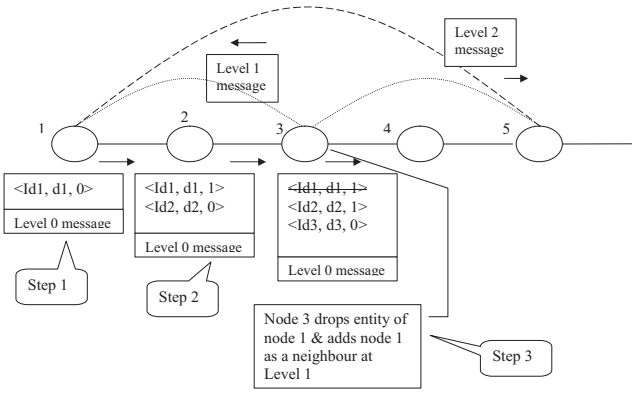


Figure 1. Gossip-based construction

Higher level gossip messages. GosSkip is fully built by iterating on this algorithm at each level. To set additional long links (that skips over more peers), peers gossip similar set of messages but only among level-1 neighbors. Note that level-0 messages are forwarded from left to right: as a result, peers on the right come to know about peers on the left. Hence peers can forward level-1 messages leftward. If the level is an even number, the message is forwarded to the right, else to the left. Once a level-1 message with counter set to 0 is received by a peer, that peer learns about another peer on the right that is k hops away (*e.g.*, in Figure 1 peer 1 learns about peer 3 when peer 1 receives a level-1 message from peer 3).

Peers with their level 0, 1 and 2 neighbors are shown in Figure 1. The lines in Figure 1 depict paths of the messages and as well as long links between peers. For simplicity, only a limited number of links are shown: in reality each peer has level 0, 1 and 2 neighbors. This scheme can be further extended to form links of greater length by gossiping another set of messages among level 2 neighbors and so on. More precisely, additional links are constructed as the system size grows. The number of links maintained by a peer is bounded by $2 \log N$. If a given peer does not *hear* from a neighbor before a *time out* period, it removes the corresponding link.

Algorithm 1 Message reception

```
1:  upon RECEIVE (message msg) with msg.level=l by peer i
2:  out-buffer_l ← null
3:  for all entities e ∈ msg do
4:    if e.counter = 0 AND l ≠ 0 then
5:      if l MOD 2=0 then
6:        leftward-neighbors.add(l,[e.peer-id,e.peer-value])
7:      else
8:        rightward-neighbors.add(l,[e.peer-id,e.peer-value])
9:      end if
10:   end if
11:   if e.counter = k-1 then
12:     if l MOD 2=0 then
13:       leftward-neighbors.add(l+1,[e.peer-id,e.peer-value])
14:     else
15:       rightward-neighbors.add(l+1,[e.peer-id,e.peer-value])
16:     end if
17:   else
18:     e.counter ← e.counter+1
19:     out-buffer_l ← out-buffer_l ∪ {e}
20:   end if
21: end for
```

Algorithm 2 Message emission

```
1:  At peer i
2:  for all neighbors in level l ∈ [0..l_max] do
3:    for each t * (l + 1) sec do
4:      msg ← out-buffer_l
5:      e ← [myID,myValue,0]
6:      msg ← msg ∪ {e}
7:      msg.level ← l
8:      if l MOD 2=0 then
9:        send msg to immediate rightward-neighbor at level l
10:     else
11:       send msg to immediate leftward-neighbor at level l
12:     end if
13:   end for
14: end for
```

3.1 Gossiping Algorithm

Algorithm 1 and Algorithm 2 show the pseudo-code of the gossip construction of links. In the pseudo-code, *peer-value* refers to the data item associated to an external peer and *myValue* refers to the local peer's data item.

Message reception. Algorithm 1 describes the steps carried out by a peer when it receives a message of level l. Once a gossip message is received, one out of two possible link types are created. In the first link type, links skip a number of peers as discussed earlier. These links are constructed if the counter of a given entry is set to k-1: then the associated value is added to the relevant (either left or rightward) neighbor list (lines 11-17) and the entry is no longer gossiped. For example, if the level of the message is an even number, the leftward neighbor list is updated (*e.g.*, for a message at level 0, a neighbor at level 1 is added to leftward list if the counter has reached k-1). If the counter is less than k-1, the counter of the entity is incremented (line 18) and it is added to the out-going buffer (out-buffer$_l$) corresponding to the level l (line 19).

The second link type connects immediate neighbors in a given level. For example, peer 1 and 3 are immediate neighbors at level 1. Whenever the counter of a given entry e is set to 0, e corresponds to an immediate neighbor at level l. Then, depending on whether l is odd or even, the corresponding neighbor list is updated as shown in line 4-10 (level 0 links are managed when nodes join and leave).

Message emission. Algorithm 2 shows the message forwarding algorithm. For all the neighbors at each level messages are forwarded periodically in the relevant direction. The period of forwarding depends on the level: lower level messages are gossiped more frequently than higher level ones. As a result, lower level links are maintained with more accuracy (in terms of link length) in the presence of join/leave of peers than higher-level links.

The *out-buffer*$_l$ data structure contains the entries received during the last period: a message is constructed containing all the entries of this buffer (line 4). An entry corresponding to peer i is also added to the message (line 5-6) with the associated counter set to 0 (line 5). The level of the message is set accordingly (line 7). Each message has a direction according to its level (line 8-12): for example, as in Figure 1, level 0 messages are sent rightward.

3.2 Routing and Spreading

Routing to a peer according to its value (or routing to the peer that has the nearest value, if it does not exist), is similar to querying a value in a balanced tree. The process begins at the higher level linked list, and goes down to lower levels when, at a peer, the current level link skips too many peers. This requires $O(\log N)$ routing steps. If the query has to be further spread over several peers, for a range query or if many peers share the same data element description, we use the spreading protocol that we describe in the next paragraph.

Efficient and fault-tolerant spreading. While routing provides the very same exact-match interface as a DHT, GosSkip preserves the ordering of data items in the list. This permits to define a more general query model, that is a *spreading algorithm*, which is both used as a range query mechanism and to propagate messages among peers with the same value. This algorithm is designed to cope with dynamicity. While some links at each level may be missing, or some peers may fail or some transient routing failures between peers may exist, the spreading still reaches all non failed peers in the given range. The spreading algorithm exploits the multiple balanced trees structure of GosSkip both to speed up the multicast process and to provide resilience to failures. The first peer satisfying the query (joined by routing to any point in the range) is responsible for initiating the multicast process to all the *matching area*.

Each encountered peer follows Algorithm 3. The key idea is, at each peer, to divide the spanning space using high level links up to a level l, and to delegate each neighbor on this level a sub-space of the matching area. When a peer forwards a query to one of its neighbors at level $l-1$, it assigns it the task of spreading it, in the same direction, to every peer between itself and its next immediate neighbor at level l. Each query for spanning a region contains a *level upper bound* and an *offset*. The level indicates how many levels the peer as to deal with, and the offset is used to avoid overlapping when repairing failed links. A *spread message* consists in spreading a query to $\left(2^{level+1}-1\right) - offset$ immediate neighbors in a given direction. Figure 2 depicts an example. Let peers 0 to 8 be a subset of the matching area and peer 0 be the first reached target of the routing. Peer 0 spreads the query on its right: it sends simultaneously spreading messages to his neighbors (peers 1,2,4 and 8). Peer 1 is in charge of itself only, peer 2 is in charge of itself and peer 3, etc. Thus, the maximum load for a query propagation on a peer is $O(\log N)$ (maximum l messages) and a peer receives the query one time only. This algorithm builds efficiently a spanning tree using alive GosSkip links for all peers in the matching area with a high resilience to failures.

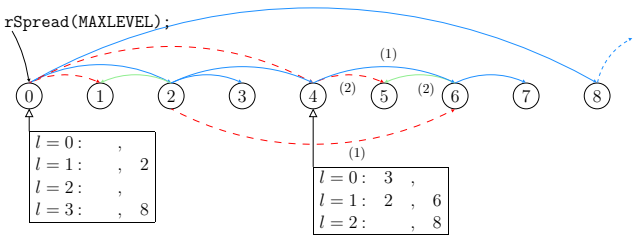


Figure 2. Spreading algorithm principles

3.3 Failure Recovery

Nodes in the overlay can fail or leave without notification: we refer to them simply as failures (we do not ad-

Algorithm 3 GosSkip Spreading – Upon reception of *lspread-msg* at level l with $offset$ on peer n

Require: $level \geq -1$,$offset \geq 0$
Ensure: Spreads the query to every matching $\left(2^{level+1}-1\right) - offset$ left neighbors
1: $l \leftarrow level$
2: **while** $l \geq 0 \wedge 2^l \leq offset$ **do** {*compute the highest destination level according to the offset correction*}
3: $offset \leftarrow offset - 2^l$
4: $l \leftarrow l - 1$
5: **end while**
6: $l_{matching} \leftarrow$ highest-level matching neighbor ($l_{matching} \leq l$)
7: **if** $level = \texttt{Maxlevel} - 1$ **then**
8: send lSpread-msg($level$,0) to immediate leftward-neighbor at level $l_{matching}$
9: **else**
10: **if** $l_{matching} = level$ **then** {*the target level is reachable*}
11: send lSpread-msg($level - 1$,$offset$) to immediate leftward-neighbor at level $l_{matching}$
12: **else**
13: send lSpread-msg($level$,$offset + 2^l$) to immediate leftward-neighbor at level $l_{matching}$
14: **end if**
15: **end if**
16: $l_{last} \leftarrow l_{matching}$
17: **for** $l \leftarrow l_{matching} - 1$ **to** 0 **step** -1 **do**
18: $n \leftarrow$ immediate leftward-neighbor at level l
19: **if** n exists **then**
20: **if** $l_{last} = l + 1$ **then** {*the following neighbor is valid*}
21: send lSpread-msg($l - 1$,0) to n
22: **else** {*some following neighbors are broken*}
23: send lSpread-msg($l_{last} - 1$,2^l) to n
24: **end if**
25: $l_{last} \leftarrow l$
26: **end if**
27: **end for**
28: **if** $n = null$ **then** {*uses the last valid neighbor to spread to the first broken neighbors*}
29: send rSpread-msg($l_{last} - 1$,0) to immediate leftward-neighbor at level l_{last}
30: **end if**

dress malicious peers in this paper). Failures are handled along two directions: (1) establishing new links between new neighbors instead of failed ones, (2) use of alternative links instead of failed links to forward messages before full recovery.

Establishing New Links. Establishing long links consist in repairing level 0 links. Once these links are established between new neighbors, the higher level links will be eventually constructed as a result of the gossip process. Note that even if the higher level links are faulty just after a failure, the forwarding of queries can take place as we will describe shortly. In the presence of multiple simultaneous node failures, two complementary approaches can be taken for failure recovery.

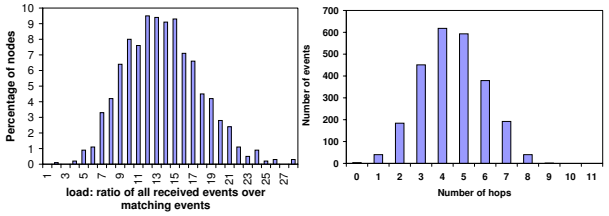
A node keeps information about R distinct neighbor nodes (i.e., nodes having different IP addresses) in the immediate vicinity. The R set of neighbors is similar to the *leaf set* structure in Pastry [16]. The use of distinct IPs helps to resolve issues that can arise when one physical computer executes R or more number of peers that are contiguous in the overlay. The maintenance of R neighbors is done using the level 0 messages itself: these messages are like *heart beat* messages between these nodes. If peer n_{r+1} fails, the peer n_r will replace n_{r+1} with n_{r+2} to form a level 0 link. As a result, if R consecutive peers do not fail simultaneously, the overlay functions properly. To tolerate more than R simultaneous failures around a given point in the overlay, we use the long higher level links. For example, if the node n_r experiences failures among its neighbors, it can use higher level links to multicast recovery messages to other neighbors in the close vicinity. Once the alive nodes in the vicinity responds, peer n_r identifies the closest level 0 neighbor and establishes a new connection. Recovery using the multicast approach is more time consuming and comparatively complex to implement than the first solution based on knowing R set of neighbors.

4 Evaluation

In this section, we describe experimental evaluation of GosSkip. We present our experimental settings and performance evaluations both in static and dynamic scenarios. We also give results on the behavior of GosSkip in presence of nodes failures. For evaluating the performance of GosSkip we implemented the algorithm and carried out a set of experiments on a distributed platform, using computers that are distributed within EPFL campus and Grid'5000 [1]. We used 256 processors, each executing several instances of GosSkip peers to increase the participating peers up to 1000 in the overlay. The parameter k is set to 2 and the gossip period is set to 20 seconds.

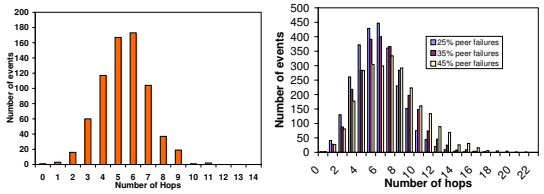
Real workloads. We used a real p2p trace to construct a sample set of data elements and queries. This trace was collected from a modified FastTrack ultra-peer, implemented in the MLDonkey multi-network file-sharing p2p client [9]. FastTrack is a hybrid p2p network where peers can act as ultra-peers, an ultra-peer being responsible for a set of regular peers. Regular peers send their shared files lists and requests to their ultra-peer. Ultra-peers forward queries among themselves and answers queries they receive with the address of the corresponding regular peer. The trace we used was collected during four days (Oct 14 to 18 2004) and contains more than 6 GB of raw data (all applications messages that passed over the ultra-peer). Collected messages include (i) search queries issued by peers, defining constraints on pre-defined attributes and (ii) cache content advertising by peers to the ultra-peer.

We used the trace to generate the workload as follows. Every data element has a multidimensional representation that we map to a single dimension using lexicographic linearization. While FastTrack client applications may specify



(a) Distribution of $load_p$ (b) Route lengths.

Figure 3. Static scenario results



(a) Route lengths (b) Massive failure impact

Figure 4. Performance of GosSkip in a dynamic setting and with failures

several attributes for files they publish in the network, only a very few of them are broadly used, others being used only by a very few client's search requests. Early analysis of the trace has shown that search requests are specifying keywords (99.9% of requests), genre (among predefined genres in id3 tags of mp3 audio files) and language (again in the predefined list of id3 tags). Lexicographic ordering of the three attributes we used is as follows: language then genre and finally keywords set. Not surprisigly for a real workload, both genres and languages follow a zipf distribution.

Lexicographic ordering is an application-based assumption, but one can use any linearization technique [4, 13] to map multidimensional data to the GosSkip model, provided that elements can be ordered without ambiguity.

The following experiments, if not specified explicitly, are based on a 1000 peers overlay and 1000 queries, both based upon the real trace. We first examine performances of GosSkip with static settings and then in a dynamic scenario in presence of peer failures.

Overlay construction and maintenance. We first measure the amount of messages used to construct and maintain the links in the overlay. Each peer gossips with its neighbors to create and maintain its neighborhood and long links. We observe that distribution of number of gossip messages among peers is between 2 to 5 messages per minute.

Load at each peer. We define the load at a peer p ($load_p$) as the ratio between the number of queries it sees for routing purposes and the number of queries that match its data element. The distribution of $load_p$ shows if the load of propagating queries is balanced among peers. Figure 3(a) depicts this distribution in the static scenario. GosSkip differs from a single skip list as it distribute load of propagation of queries among peers: there is a low imbalance in the distribution of this load, as no peer is loaded with more than two times of the mean load ($\overline{load_p} = 13, 5$).

Routing performance. We also count the number of *hops* to deliver queries to a matching peer or to a peer that has sufficient local knowledge to know that the message has to be discarded. Figure 3(b) presents the hop-count distribution for 1000 queries. Most queries reach corresponding peers in our 1000 peers overlay in less than 6 hops, a very few of them using up to 9 hops to deliver the query. This shows the ability of GosSkip to efficiently route messages, confirming the expected $O(\log_k N)$ routing property, while not overloading any peer in the overlay.

Next we show a complementary set of performance results. More specifically, we examine the performance of the GosSkip in a dynamic setting and in presence of failures.

Routing performance in a dynamic scenario. This experiment shows how GosSkip deals with dynamic scenarios concerning routing efficiency. We evaluate how the queries would be routed to a recently joined peer. To this end, we first construct an overlay with 300 peers (less than 1/3 of eventual total of peers). Once the network stabilizes with this initial set of peers, we did the following steps: 1) at each cycle, add a new peer and let it joins the overlay 2) just after this peer establishes level-0 links a query that matches the new peer's data element is injected into the overlay (note that by this time the new peer has no other links to and from it other than level-0: *i.e.*, no long links). 3) We then count the number of hops to deliver this query to the new peer. Above 3 steps are carried out till the total number of peers in the network is equal to 1000.

In this experiment we use a total of 700 queries that match newly joined peers. Figure 4(a) depicts the number of hops taken to deliver queries. The upper bound for hops for the initial set of peers (*i.e.*, 300 peers) is 8.22: the upper bound for hops for the eventual total of peers (*i.e.*, 1000) is 9.96. As seen in Figure 4(a) some queries take more step than this: but in general GosSkip performs well in this kind of dynamic scenario.

Query spreading. We first evaluated the resilience to failures of the spreading algorithm. Our results match the expected $O(\log m)$ (m is the size of the matching area)complexity. We evaluate the failure resilience of the algorithm by measuring the proportion of peers reached by a query when each peer may fail at each round with a probability p. We compare our algorithm to a basic *nearest neighbor* algorithm which spreads queries using level-0 links only and a *next neighbor* algorithm which forwards queries to the lowest level alive neighbor. These algorithms spread queries in linear time. The two latter methods are obviously better since they consider different level links as well. However, we observe that the GosSkip spreading algorithm

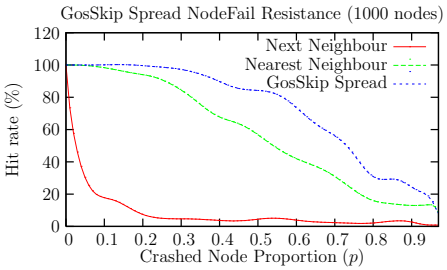


Figure 5. Spreading robustness

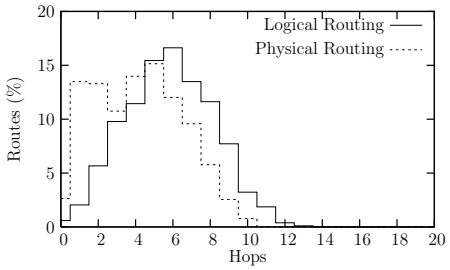


Figure 6. Physical routing impact

outperforms the nearest neighbor algorithm as the proportion of failed nodes increases, thus confirming its ability to deal with more dynamic scenarii.

Massive nodes failures. We performed an experiment to check the performance of our system in presence of massive failures. To that end, we consider the following failure scenario: after a number of nodes join, form the overlay and stabilize, we let K% of nodes crash simultaneously: we consider three different cases where $K = 25\%$, 35%, and 45%. Right after the failure we start injecting queries into the overlay and we measure the performance. This is an adverse failure scenario such as non independent failures (*e.g.*, failure of number of nodes due to a power failure in a given geographic region).

After such a failure, the overlay will reconstruct links replacing the faulty links and neighbors: this would take a certain amount of time. Our goal is to measure the performance of the overlay before the recovery of the overlay takes place. In other terms, we measure the performance of the query forwarding before any recovery action takes place after the failure.

The performance is measured in terms of (1) number of hops taken by an message before being delivered or terminated (2) number of messages that are not delivered to nodes that exist in the overlay and match the query. Note that because of failures the number of hops can be larger than in the case when there are no failures and that there can be queries which cannot be forwarded because of any anomalies that can exist in the overlay just after the simultaneous crash of nodes. To forward queries after the failures peers use alternative links instead of faulty ones. Figure 4(b) shows the number of hops taken to deliver/terminate queries after the failure. As seen, the number of hops are larger than the $O(\log N)$ (which is equal to 9.96 hops) upper bound: but still it is limited to a relatively small values in spite of large percentage of peer failures. In the case of 25%, 35%, and 45% peer failures the number of queries that could not be delivered to the existing interested peers are 6, 18, and 53 respectively. That is, only a very small fraction of queries are not delivered to the peers that exist in the overlay. All other queries are delivered using alternative links in spite of failures. This shows the very robust nature of the GosSkip which is a result of the redundancy of links.

5 Leveraging Physical Locality

So far, we assumed a one-to-one mapping between a peer and a physical node. However, several peers might be hosted on the same physical node. In this section, we propose to leverage this property in order to improve routing efficiency. We assume that the routing information on one physical node is accessible to all logical peers hosted on that node.

Physical routing Communications between peers hosted on a same physical node are instantaneous since they do not require network communication latency. We use this property to improve routing. To this end, we modify the proximity measure on the overlay. When choosing the next peer for a query propagation, we do not only consider that peer's neighbor, but also other peers present on the physical nodes as natural candidates. The distance between two peers (p_1, p_2) that lie on two different physical nodes is estimated on p_1 as $d(p_1, p_2) = \frac{1}{2}(2^{l_{sup}} - 2^{l_{inf}})$, where l_{sup} is the maximum level at which the associate neighbor of p_1 is *before* p_2 in the ordering (respectively the minimum level of a neighbor that is *after* p_2 for l_{inf}). The distance between any two peers that are on the same physical node is 0, since routing between these two nodes incurs no communication.

Efficiency To evaluate the impact of physical routing on routing efficiency, we distributed on Φ physical nodes uniformly at random 10Φ peers. Figure 6 shows that using physical routing permits a shift to a lower value of the mean route length, while keeping the load balanced: most queries are sent within 4 hops, while 6 hops were needed for the regular routing mechanism.

Robustness Physical routing also helps to make GosSkip more resilient to failures, as if a logical peer has no alive candidates among its neighbor to forward the query, it can uses the neighborhood of the other peers on the same physical node. Experiments show the impact on routing if, at the same time, all physical nodes have a probability p of crashing. Each physical node among Φ is given some logical peers uniformly drawn from 10Φ logicals peers in the network. A large number of queries for peers that were originally in the overlay are performed: some will fail due to the absence of the corresponding peer, some other may

fail due to non existent direct routing path. Figure 5 shows that using physical routing raises the hit ratio by 10 to 15 percent by diminishing the number of these non existent routing paths, due to the greater number of alive neighbor choices at each step.

6 Concluding Remarks

Traditional approaches for designing peer-to-peer overlays link physical nodes in a distributed data structure providing a distributed hash table interface. While such systems provide nice properties in term of routing efficiency, their ability to handle complex queries is low, due to the hashing used to map objects to node. On the other hand, some work has been done to propose distributed data structures based upon the Skip List principle that do not present this drawback. However, these approaches were mostly interested in the data structure itself, and did not provide any implementation details or solutions to deal with dynamicity. The explicit construction mechanism may be an issues be if the churn is high. In this paper, we follow an approach that is quite similar to the latter, and we connect application objects in an efficient and load balanced data structure that eventually resemble a set of perfect Skip Lists. We step away from traditional explicit construction mechanism by using gossip-based construction algorithms. This permits the overlay to be highly resilient to nodes failures and arrivals (churn). Moreover, a spreading algorithm that deal with node permanent or transient failures is proposed. Using a real implementation and a trace from a file sharing system workload, experimentations conveyed the good behavior of GosSkip, both in a static and dynamic scenario. Finally, extensions of the routing protocol to leverage the presence of multiple logical peers on a physical node are proposed. Experiments demonstrate the positive impact on the routing performance. We are currently working on the extension of GosSkip to deal with more complex application patterns, and to adapt the approach to multi-dimensional description of application objects.

Acknowledgment. We are very grateful to B. Koldehofe and A. Kupsys for their support.

References

[1] The grid'5000 project. HTTP://WWW.GRID5000.ORG.

[2] L. Arge, D. Eppstein, and M. T. Goodrich. Skip-webs: efficient distributed data structures for multi-dimensional data sets. In *PODC '05: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 69–76, New York, NY, USA, 2005. ACM Press.

[3] J. Aspnes and G. Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, Jan. 2003.

[4] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramidtechnique: Towards breaking the curse of dimensionality. In *Proc. of SIGMOD*, 1998.

[5] A. Bharambe, M. Agrawal, and S. Seshan. Mercury:supporting scalable multi-attribute range queries. In *ACM SIGCOMM'04*, Portland, Oregon, USA, Aug 2004.

[6] K. Birman, M. Hayden, O.Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM TOCS*, 17, 1999.

[7] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. Planetp: Using gossiping to build content addressable peer-to-peer information sharing communities. In *HPDC*, pages 236–249, 2003.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of ACM PODC*, 1987.

[9] F. L. Fessant. Mldonkey, a multi-network file-sharing client. http://www.mldonkey.net/, 2002.

[10] R. Guerraoui, S. B. Handurukande, and A.-M. Kermarrec. Gosskip: a gossip-based structured overlay network for efficient content-based filtering. Technical Report LPD-REPORT-2004-005 200495, EPFL, Switzerland, 2004.

[11] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, , and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. of USITS*, 2003.

[12] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations. In *Middleware*, pages 79–98, Toronto, Canada, 2004.

[13] B. C. Ooi, K.-L. Tan, C. Yu, and S. Bressan. Indexing the edges a simple and yet efficient approach to high-dimensional indexing. In *Proc. of PODS*, 2000.

[14] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *CACM*, June 1990.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM*, 2001.

[16] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware*, 2001.

[17] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM*, 2001.

[18] S. Voulgaris, E. Rivière, A.-M. Kermarrec, and M. van Steen. Sub-2-sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks. In *IPTPS'06: the fifth International Workshop on Peer-to-Peer Systems*, Santa Barbara, USA, FEB 2006.

[19] S. Voulgaris and M. van Steen. An epidemic protocol for managing routing tables in very large peer-to-peer networks. In *Proc. IFIP/IEEE DSOM*, 2003.

[20] S. Voulgaris and M. van Steen. Epidemic-style Management of Semantic Overlays for Content-Based Searching. In *EuroPar*, Lisboa, Portugal, Sept. 2005.

[21] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE JSAC*, 22, 2004.