GosSkip: a Gossip-based Structured Overlay Network for Efficient Content-based Filtering

R. Guerraoui†, S. B. Handurukande†, A.-M. Kermarrec‡,
†Distributed Programming Laboratory, EPFL, Switzerland
‡IRISA/INRIA, Rennes, France

Abstract

It is appealing to implement publish/subscribe systems in a peer to peer (P2P) manner to circumvent the scalability issues of broker-based (semi-decentralized) systems and simplify the deployment. While existing P2P generic infrastructures provide a scalable support for topic-based publish-subscribe systems, they are not well adapted to content-based ones. In this paper, we advocate the need for dedicated overlay networks -where the overlay structure reflects the actual structure of the underlying application properties- to implement efficient content-based publish-subscribe infrastructures. We propose a new scalable P2P event filtering mechanism where filters are arranged according to a dictionary-based semantics. This structured overlay, which we call *GosSkip*, relies on gossip messages to construct a structure eventually similar to a perfect Skip list, preserving the semantic locality of the items stored in the overlay. In GosSkip, events are delivered to matching subscriptions in O(log N) routing hops, N being the total number of subscriptions. The very same approach can be extended to support range queries. Preliminary implementation results based on a real P2P trace convey the scalability and the efficiency of the approach both in static and dynamic scenarios.

1 Introduction

In publish-subscribe systems [7], information consumers, called subscribers, register their interest to specific events in order to be asynchronously notified of any event matching their subscription. A key functionality of event-based systems is to provide subscribers with the ability to receive only events they are interested in. To this end, it is desirable to perform filtering according to the content of the event. Subscribers specify their interests using subscription filters, generally consisting of a number of predicates. Systems that provide this feature are also known as content-based publish-subscribe systems [7], and are considered for many applications such as stock exchange notifications and sensor networks.

Traditional filtering schemes use a set of dedicated server-like intermediate brokers to achieve filtering. To overcome the intrinsic scalability limitations of broker-based approaches [3, 1] as well as simplifying the deployment, P2P overlay networks are natural candidates for the implementation of content-based publish-subscribe systems. Generic P2P overlays [11, 13, 12, 17], provide the functionality of a distributed hash table (DHT) and can be used to efficiently locate an object specified by a key (e.g., a filename) within a large set of nodes. While they provide an efficient support for topic-based publish-subscribe systems [4], it is not straightforward to use them efficiently when multiple attributes are involved in the search. Some attempts have been recently made in that direction [5, 14]. Unfortunately, in the presence of popular attributes, few nodes get highly overloaded and the matching algorithm is computationally expensive. These approaches use general-purpose structured P2P overlays thus relying on a high-level of randomization to achieve load-balancing. Willow [15], on the other hand combines a DHT with "aggregation" and SQL like predicates to implement publish-subscribe systems. In this paper, we advocate the need for a dedicated P2P overlay reflecting the structure of the application itself while preserving its semantics: the overlay links application objects rather than computing entities.

This paper presents an event filtering scheme for content-based publish/subscribe built using a genuine P2P model called *GosSkip*. In GosSkip, subscriptions filters are organized according to a lexicographic order to form a P2P structured overlay so that published events are efficiently routed, in O(log N) hops (N being the total number of subscription filters), to interested subscribers. We build such a structured overlay network using gossip-based

techniques. Gossip techniques have been successfully used in database maintenance [6], multicast [2] and routing tables management [16]. Our structured overlay network eventually forms a "perfect Skip list [10, 9]"[1]. As in Skip Net [9] which is similar to probabilistic-Skip lists [10], GosSkip has the important property of preserving content locality in the semantic space. However GosSkip eventually builds a perfect Skip list in a deterministic fashion.

Section 2 introduces the dictionary-based subscription model. Section 3 presents GosSkip construction and maintenance. Preliminary results of our implementation are provided in Section 4 and Section 5 concludes.

2 Dictionary-based Model

GosSkip relies on ordering subscription filters. We define a dictionary-based subscription model where subscription filters can be arranged to introduce a dictionary like order to these filters. As shown in Example 1, a filter consists of a set of predicates. Each predicate has a 3-tuple, attribute, value and operator. It is possible to order the values of a given attribute whether the values are of type numeric, string etc. As a result, a set of predicates related to a given attribute can be ordered: Example 2 shows the left most predicates of the filters f1, f2, f3, f4, ordered according to the alphabetical order of the values while the right most predicates of filter f2, f3 are ordered according to the numerical order of the values. Based on this, the filters can be ordered by ordering predicates starting from the left most predicate and progressively ordering the filters according to the other predicates till the right most predicate. As a result, we have the following order for the filters: $f3 < f2 < f4 < f1$.

Example 1 *An example set of subscription filters:*

```
f1 = (class, weather, =),(country, Switzerland, =),
        (city, Geneva, =)
f2 = (class, auctions, =),(category, vehicle, =),
        (type, cars, =), (year, 2003, =)
f3 = (class, auctions, =),(category, vehicle, =),
        (type, cars, =), (year, 2002, =)
f4 = (class weather, =),(country, France, =),
        (city, Paris, =)
```

Example 2 *Ordering Individual Predicates:*

```
(class, auctions, =)  <  (class, weather, =)
(year, 2002, =)  <  (year, 2003, =)
```

To perform this ordering, peers also need to agree on the predicate order. This can be specified according to a lexicographic order based on attributes names or at the advertisement phase of the events by the publisher before any event is published. It is very common that applications rely on a well defined set of attributes, known by all peers. For example, requests in P2P file sharing systems contains attributes such as *title*, associated with the operator *contains* and *size*, associated with the operator >. All requests are made tacitly according to this model.

Once subscription filters are ordered, they are mapped to the overlay structure as discussed in Section 3. Subsequently, matching events can be forwarded using GosSkip to interested subscribers.

3 Gossip-based Structured Overlay for Efficient Event Filtering

GosSkip is a structured P2P overlay where peers are related to subscription patterns. A peer, and its location in the overlay structure, is defined by its subscription filter. We now describe how GosSkip is constructed using gossip-based techniques. We then describe peer insertion, deletion and event management.

3.1 Overlay Construction

To simplify the presentation, we assume that there is a one-to-one mapping between subscription filters and nodes, although there might be several subscriptions filters per physical node[2].

GosSkip organizes subscription filters in a structured P2P overlay in such a way that their associated data items form a sorted linked list like structure. Search for data items may be done simply by traversing the list. Obviously, as the system size increases, the search becomes inefficient due to a large number of hops. To circumvent this issue, longer links (such that a link can skip over number of nodes) between peers are established as in Skip lists so that a search requires only O(log N) routing hops [10, 9]. In other words, peers require to be aware of their immediate neighbours in the sorted list and of only few other peers located at increasing distances. Links to

[1] Skip lists have been proposed as an alternative to balance binary trees and is in fact a linked list augmented with additional pointers. It can be generalized into a form of distributed hash table (DHT). As discussed in [9], in a "perfect Skip list", level h pointer traverses exactly 2^h nodes, for example.

[2] In the rest of the paper, we call a participant of the overlay, a peer.

immediate neighbours are straightforward to implement once a peer has located itself in the sorted list. We present below how to create long links.

Establishing Long Links. GosSkip relies on gossip messages to construct long links. To minimize the overhead of these gossip messages, it is possible to piggy-back them over heartbeat messages (or even application messages) as used in the maintenance of most P2P networks. Each node periodically sends messages to the nodes on the right hand side first, as shown in Figure 1. [3]

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 (e.g., a filter; d1 in Figure 1) and a counter. Periodically, each peer forwards a subset of the entries it receives during the last period. A peer also adds an entry with its own id, data item and the 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 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 neighbour list (e.g., as in Step 3 of Figure 1, peer 3 adds Id1 to its neighbour set together with d1). Peers have neighbours on right and left hand sides, maintained respectively in rightward- and leftward-neighbour lists.

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. This process is depicted in Figure 1 where k is set to 2. Following the terminology used in [10], we call the immediate neighbours (i.e., one hop away), level-0 neighbours and the neighbours that are k hops away level-1 neighbours: for example, as in Figure 1, node 1 and 2 are level-0 neighbours while node 1 and 3 are level 1 neighbours. Likewise, each message is associated with a level representing the level of the neighbours between which the message is sent.

[3]For simplicity of presentation, we show nodes forming a linear structure but nodes can form a circular: in other terms, if there are only 5 nodes in the network, node 1 and 5 are neighbours forming a circle.

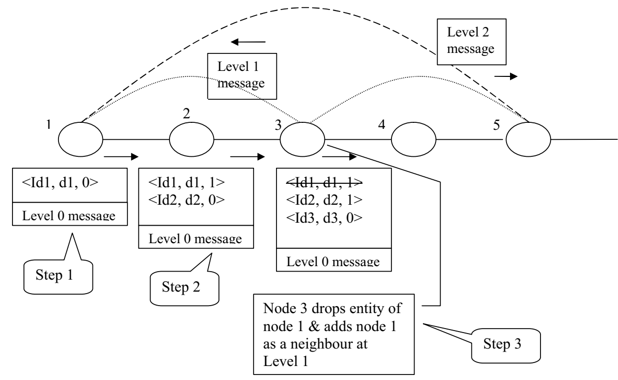


Figure 1: Gossip-based construction of the GosSkip overlay network.

Higher Level Gossip Messages. To add more longer links (that skips over even more number of peers), peers gossip similar set of messages but only among level-1 neighbours. 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. In short, 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 are 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).

Following a similar protocol, the entries in these specific messages are stopped for further forwarding when the counter exceeds k. Whenever an entry is stopped from being further forwarded by a given peer, it adds the associated peer of the entry as a level 2 neighbour.

Peers with their level 0, 1 and 2 neighbours are shown in Figure 1. The lines in Figure 1 depict paths of the messages and also the long links between peers. For simplicity, only a limited number of links are shown: in reality each peer can have level 0, 1 and 2 neighbours. This scheme can be further extended to form links of greater length by gossiping another set of messages among level 2 neighbours and so on: in other words, additional links will be automatically constructed as the system size grows. The number of links maintained by a peer is bounded by 2log N.

3.2 Algorithm

The pseudo-code for the gossiping algorithm is presented on Figure 2. To present the algorithm in a general form, the data item associated with the peer

```
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-neighbours.add(l,[e.peer-id,e.peer-
         value])
7:     else
8:       rightward-neighbours.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-neighbours.add(l+1,[e.peer-
         id,e.peer-value])
14:    else
15:      rightward-neighbours.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
```

(a) Message reception

```
1: At peer i
2: for all neighbours 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-neighbour
         at level l
10:    else
11:      send msg to immediate leftward-neighbour at
         level l
12:    end if
13:  end for
14: end for
```

(b) Message emission

Figure 2: GosSkip Overlay Construction

is referred as the "value" (e.g., peer-value, myValue): in our case it refers to the subscription filter.

Message Reception. Figure 2(a) shows 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 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) neighbour list (Figure 2(a) line 11-17) and the entry is no longer gossiped. For example, if the level of the message is an even, leftward neighbour list is updated (e.g., for a message at level 0, a neighbour 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 neighbours in a given level. For example, peer 1 and 2 are immediate neighbours at level 0; on the other hand peer 1 and 3 are immediate neighbours at level 1. Whenever the counter of a given entry e is set to 0, e corresponds to an immediate neighbour at level l. Then depending on whether the l is an odd or even, the corresponding neighbour list is updated as shown in line 4-10 (level 0 links are managed when nodes join and leave which is not shown here). For example in Figure 1, if peer 1 receives a level 1 message from peer 3, peer 1 adds id and value of peer 3 as the right hand neighbour at level 1.

Message Emission. Figure 2(b) shows the message forwarding algorithm. For all the neighbours 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. As a result lower level links are maintained with more accurately (in terms of link length) in the presence of join/leave of peers.

The buffer *out-buffer$_l$* 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.

3.3 Joining and Leaving

When a peer wants to join GosSkip, it simply sends a `join` message to a peer already participating in the system as in most insertion algorithms [11, 12]. 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. As the gossip messages are exchanged in the systems, the peer is gradually integrated in the neighbour lists. When a peer wishes to leave the system, it just stops gossiping messages. Besides, its neighbours detect their neighbour failure and therefore removes it from their list of neighbours: a new level 0 link is created by

peers besides the failed/departed peer.

If a physical node hosts a large number of subscription filters, there might be some scalability issues with regards to the state needing to be maintained. However, this issue remains manageable since the physical node imposes itself the load.

3.4 Event Routing

GosSkip is used to determine how to route events to matching subscriptions. Note that subscription filters are treated like any other values stored in the sorted linked list in the overlay. Hence event routing is similar to finding a value in the linked list. In other terms, in GosSkip, events are forwarded along the overlay network until the peers that are interested in the event are reached. We consider that peers publishing events are either part of the overlay or are able to contact a peer participating in the overlay. The event will then be automatically and efficiently be routed to either one of the matching peer if any or to a close peer. Forwarding of events should be terminated when there are no more matching subscriptions. Thanks to the content locality of the structured overlay network, it is feasible for a given peer to decide when to forward and when to terminate the forwarding process using only its local knowledge. The absence of any interested peer can be detected since each peer knows the subscription pattern of its neighbours. Likewise, a matching peer receiving an event will check whether they are other interested peers (in the case they are contiguous in the overlay) and forward them the event.

There might be some pathological cases where a large number of events are published with no interested subscribers. In that case, the nodes closest to the matching value may be overloaded (similar to the case when there are popular files in DHT based file sharing networks). This can be circumvented by forwarding a "proxy filter" to wards the publisher.

3.5 Range Queries

A subscriber s_1, with identity id_1, can subscribe to a range of values by using a range subscription filter such as f=(v_1, v_2) where v_1 and v_2 are values such that $v_2 > v_1$, representing a contiguous range in the overlay. Range queries represent an important challenge in P2P content-based publish-subscribe systems. GosSkip can be extended to handle such subscriptions. Due to the lack of space, we do not present here the detailed algorithm but we describe the approach. In simple terms, a range subscription can be inserted by storing a special routing entry in a subset of peers, S_R, including the peers having values between v_1 and v_2. This routing entry indicates that when such a peer receives a matching event, it should also forward the event to the peer as indicated in this routing entry.

As a first step, the subscriber s_1 joins the overlay as a normal subscriber that has a filter f=v_1: then it will forward a special gossip message attached to the level 0 gossip messages that are forwarded to the right hand side neighbours by all peers in S_R. This special message consists of f and id_1 and indicates that s1 is interested in all events between v_1 and v_2. When this message is received, every peer in S_R performs 2 tasks: 1) forwards the message to its right hand side neighbours along with level 0 gossips; 2) stores f=(v_1,v_2), with id_1 as a special routing entry. In addition, if peer p is the peer having a filter f_p=v_p such that v_p is the smallest value in the overlay but still $v_p > v_2$, then p also performs task 2.

When an event v, having a value between v_1 and v_2 is published, it will be forwarded to the corresponding peer. This peer will check the routing entry that is stored in task 2 and forward the event to peer with id_1 for example. Filter merging and filter covering relationships can be used to limit the number of range subscriptions stored at a given peer.

Predicates can have "wildcard" characters (as $*$ in shells) indicating that the subscriber "does not care" about the values of the given attribute and likes to receive events irrespective of values of the attribute set to the wildcard: but still events delivered to subscriber should satisfy other predicates. Assume a filter consists of three predicates: f=$p_1 \wedge p_2 \wedge p_3$. If in a subscription, p_3 is specified as a wildcard, this filter can be handled like a normal range query. Here the range consists of all the possible values of the attribute of p_3. On the other hand, if p_1 and p_3 are specified with values and p_2 is specified with a wildcard character, the algorithm needs another set of extensions: to that end, number of additional routing entries need to be kept at various peers in the overlay. This comes at the expense of a heavier join phase for the given peer and with the additional cost of maintaining routing entries. We are in the process of exploring these extensions.

4 Evaluation

For evaluating the performance of GosSkip we implemented the algorithm and carried out a set of experiments using computers that are distributed within

EPFL campus [4]. Each computer executes several dozens of GosSkip instances to increase the participating processes up to 1000 in the overlay. We used a real P2P trace [8] to construct a sample set of subscription filters and events: we also used a synthetic trace where events and subscriptions are randomly distributed in the corresponding event and subscription space.

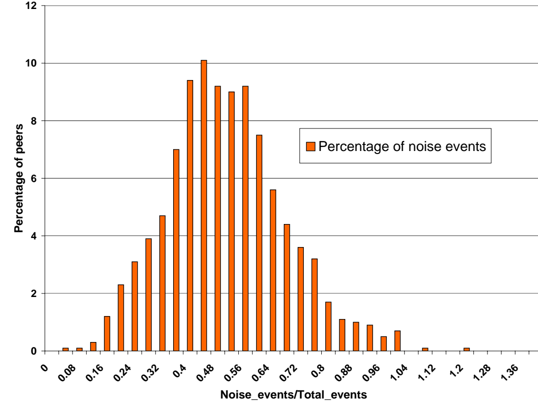
Considering peers' interests are represented by their shared files, a filter is constructed so that the filter matches one shared file by the peer. Events are also constructed from such information: one event represents one file shared by a peer. These events and filters consist of 3 attribute, namely file name, type and size. In our experiment, the parameter k was set to 2 and 2500 events were injected into the overlay with 1000 subscription filters both in the case of real P2P trace and synthetic trace.

Once the overlay network is constructed, we select an event e and forward it to a randomly selected peer in the overlay. The event is then routed to matching subscriptions if any (or garbage collected otherwise).We evaluated GosSkip along two metrics: *(i)* the load on each peer and *(ii)* the number of hops involved in event forwarding.

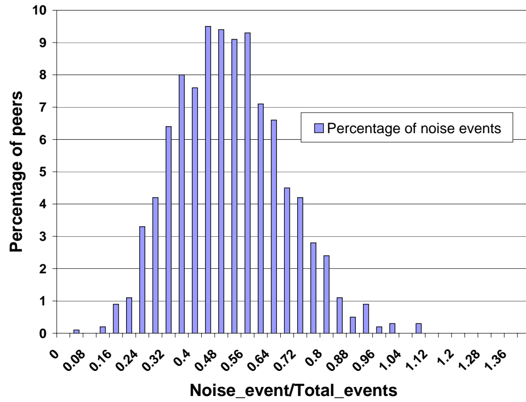
In this preliminary evaluation, to evaluate the load, for each peer, we count the number of matching and non matching events: we call the unmatching events, "noise" events. The number of noise events received mainly depends on the number of events injected into the overlay. Therefore, we "normalized the noise" by considering the fraction [noise events at peer/total events injected to overlay]. The distributions of noise level among peers are depicted in Figure 3(a) (with real P2P trace) and Figure 3(b) (with synthetic trace): the x axis represents the normalized noise events and y axis represents the percentage of peers receiving a given noise level. For example, in the case of real P2P trace around 10% of peers receive 0.48% of the total number of events sent to the system. The most heavily loaded peers (0.1% of peers) received only about 1.2% of the total events sent to the system (note that we do not count in this load the events that peers were indeed interested in). A similar distribution is observed the synthetic trace.

For each event, we also count the number of hops it has taken before being delivered or garbage collected by a peer. Figure 4(a) (with real P2P trace) and Figure 4(b) (with synthetic trace) show the number

[4]we are in the process of deploying the experiments in the Planet Lab.



(a) Real Trace



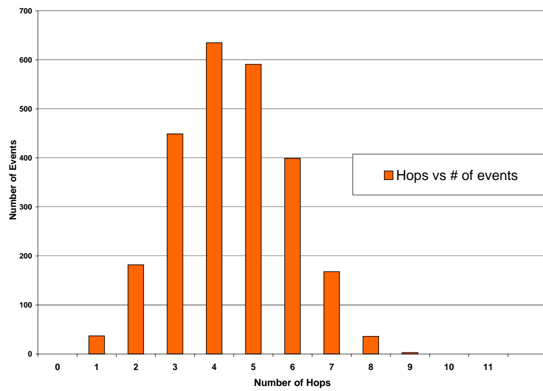
(b) Synthatic Trace

Figure 3: Fraction of(noise events/total events in system) received at nodes

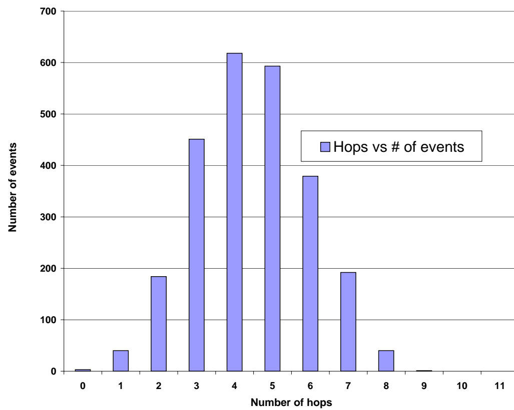
of events against the hop count: most of the events (e.g., with real P2P trace 635 out of 2500 events) are delivered or garbage collected by using only 4 hops. Again results are similar in both the traces.

As seen in these experiments, events can be delivered efficiently (with limited amount of hops bounded by the logarithmic value of subscriptions) while not overloading peers in the overlay.

Routing in a Dynamic Setting. We also carried out an experiment to see how GosSkip performs in a dynamic scenario: more precisely we wanted to check how the events would be routed to a newly joined peer. For this we first constructed 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) add a new peer with a randomly selected subscription and let it joins



(a) Real Trace



(b) Synthatic Trace

Figure 4: Number of hops taken to deliver events

the overlay 2) just after this peer establishes level 0 links (with 1 hop length) an event that matches the new peer's subscription 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 taken to deliver this event 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 used the real P2P trace and a total of 700 events that match subscriptions of newly joined peers are injected. Figure 5 depicts the number of hops taken to deliver events. The upper bound for hops for the initial set of peers (i.e., 300 of 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 5 some events take more step than this: but in general GosSkip performs well in this kind of dynamic scenario.

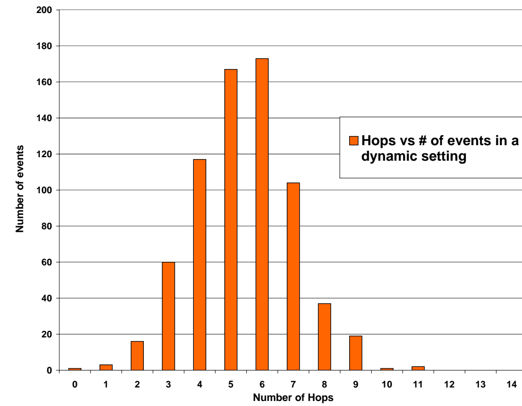


Figure 5: Number of hops taken to deliver events in a dynamic setting

5 Conclusions

GosSkip represents two major contributions over previous work on P2P event-filtering mechanisms. First, GosSkip relies on a gossip-based approach to implement an eventual perfect skip-list like overlay network. Second, GosSkip does not rely on the uniformity and the randomness used as the basis to evenly balance the load between peers in a DHT-like overlay network and connects data items rather than computing peers. The subscription filters are disseminated in the overlay in a lexicographic order and events are efficiently routed to matching subscription filters without overloading non interested peers. To the best of our knowledge, this represents one of the first attempt to step away from generic P2P structured overlays and connect application items together rather than computing entities. There are few issues that still need to be addressed and experimented with in our approach. GosSkip is currently implemented in the context of a content-based publish-subscribe system. However, we believe that the philosophy of GosSkip and the data structure it provides could be used in various contexts such as for efficient search. We are currently investigating the use of multiple dimensions search using GosSkip.

Acknowledgement. We are very grateful to F. Le Fessant and B. Koldehofe for their support.

References

[1] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. of ICDCS)*, 1999.

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

[3] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM TOCS*, Aug. 2001.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized publish-subscribe infrastructure. *IEEE JSAC*, 20(8), Oct 2002.

[5] Y. Choi, K. Park, and D. Park. Homed: A peer-to-peer overlay architecture for large-scale content-based publish/subscribe systems. In *Proc. of DEBS*, 2004.

[6] 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.

[7] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM computing Surveys*, 35(2), June 2003.

[8] F. L. Fessant, S. Handurukande, A.-M. Kermarrec, and L. Massoulié. Clustering in peer-to-peer file sharing workloads. In *Proc. of IPTPS*, 2004.

[9] 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.

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

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

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

[13] 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.

[14] P. Triantafillou and I. Aekaterinidis. Content-based publish-subscribe over structured p2p networks. In *Proc. of DEBS*, 2004.

[15] R. van Renesse and A. Bozdog. Willow: Dht, aggregation, and publish/subscribe in one protocol. In *Proc. of IPTPS*, 2004.

[16] 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.

[17] 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.