Publish-Subscribe Tree Maintenance over a DHT

Paolo Costa and Davide Frey
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
{costa,frey}@elet.polimi.it

Abstract

Content-based publish-subscribe middleware is emerging as a promising answer to the demands of modern highly dynamic distributed computing by providing the necessary decoupling and flexibility. The majority of currently available systems implement event dispatching on top of an overlay network with a tree topology. However, they fail to provide any mechanism to maintain it in the presence of failures, thus hampering their applicability in dynamic scenarios.

In this paper, we present a novel approach to reconfiguring the overlay topology by exploiting a Distributed Hash Table. Our algorithm supports arbitrary tree topologies and deals very well with the dynamicity of network scenarios by limiting the impact of reconfigurations induced by topology changes. These results are confirmed by simulations which validate the applicability of our approach in reconfigurable publish-subscribe middleware. Beyond publish-subscribe, the algorithm is applicable in a wide range of contexts and provides a general way to maintain an overlay network with a controlled topology in dynamic environments.

1. Introduction

The widespread availability of computing devices and network connectivity is fostering a renewed interested in distributed computing applications, ranging from simple data sharing facilities to complex systems able to take decisions and perform computations in a decentralized fashion. More and more often, it is necessary to coordinate devices belonging to several networks and administrative domains, thereby making issues like unreliable communication and host disconnections more and more difficult to mask. Within this context, middleware platforms play a major role in allowing application programmers to abstract from the details of distributed communication and coordination, and indeed a number of middleware solutions have been studied and implemented in recent years. A kind of middleware that appears well suited to this kind of scenario is that based on the publish-subscribe paradigm. In publish-subscribe middleware, applications communicate by exchanging messages which are delivered to all the components that have expressed an interest in receiving them.

Recent work in publish-subscribe has focused on developing scalable large-scale middleware platforms [1, 4]. Nevertheless, currently available publish-subscribe middleware fails to address issues like the intermittent connectivity that characterizes the hosts taking part in large-scale distributed applications. Our research group has recently focused on these issues and has developed solutions to enable the deployment of publish-subscribe middleware in highly dynamic network scenarios [9, 3, 5].

In this paper we complement these efforts by presenting an algorithm to maintain an overlay tree topology. Altough this work is motivated by our research on content-based publish-subscribe middleware, the algorithm is far more general and can be applied in several forms of group communication including multicast, topic-based publish-subscribe and peer-to-peer applications.

The algorithm exploits a Distributed Hash Table to map hosts onto a reference tree topology. Simulation results validate the effectiveness of the approach both in controlling the number of neighbors of the hosts in the network and in minimizing the impact of the algorithm on the protocol responsible for the reconfiguration of the event routing.

The paper is structured as follows. Section 2 introduces reconfigurable publish-subscribe systems and distributed hash tables. Section 3 describes our algorithm. Section 4 complements this description with an evaluation through simulations. Finally, Section 5 presents some related approaches and Section 6 concludes the paper.

2. Background and Requirements

Before we delve into the details of our protocol, it is important to understand both its application domain and the elements it relies on for its operation.

2.1. Reconfigurable Publish-Subscribe Middleware

The publish-subscribe communication paradigm has emerged in recent years as a good means for the development of large scale distributed applications. Applications exploiting this paradigm are organized as networks of autonomous components, the publish-subscribe clients, which communicate by exchanging asynchronous messages. Three primitives lie at the basis of this communication infrastructure. A client may feed events into the system by means of a *publish* operation and it may express or revoke its interest in receiving events using the *subscribe* and *unsubscribe* primitives. Content-based systems allow clients to specify subscriptions using regular-expressions while subject-based event classes play a role which is similar to that of groups in multicast communication.

In large-scale systems, routing is often addressed with a subscription forwarding strategy [1]. A single tree is used to exchange subscriptions and events between dispatchers. The former are delivered to all dispatchers, establishing the paths which will be followed by the latter along the tree.

In our previous work [9, 3, 5], we tackled the problem of reconfiguring the distributed event dispatcher as a way to address highly dynamic network scenarios. The reconfiguration problem was decomposed into three subproblems. The first is the modification of the topology of the dispatching infrastructure as a result of the appearance of new hosts, the disappearance of others or the failure of communication links. The second consists in the reconfiguration of the routes laid out by subscriptions to match the newly determined topology [9, 5]. The third consists in minimizing the events which are lost during this reconfiguration process [3].

Our tree management protocol addresses the first of these subproblems and maintains a tree topology while attempting to minimize the impact of its changes on the protocols responsible for rearranging subscription information.

2.2. Distributed Hash Tables

In recent years, research has devoted much effort to the optimization of information lookup and retrieval in peer-to-peer infrastructures. A number of systems have appeared which implement the distributed version of the well know hash-table data structure [13, 16, 12, 11]. Distributed Hash Tables (DHT) enable efficient information lookup by means of a simple operation: mapping keys onto network hosts. Available DHTs employ several strategies to maintain the interconnections between hosts and to distribute keys (and objects) among them. This allows them to provide very efficient lookup operations, with a communication complexity on the order of the logarithm of the number of hosts, in dynamic scenarios characterized by host failures and disconnections. The algorithm we propose in this paper exploits these characteristics to build and maintain a tree-shaped overlay network.

3. Algorithm Description

The flexibility of distributed hash tables, combined with their ability to withstand topological changes such as node failures, makes them an appropriate basis for the development of an overlay maintenance algorithm for reconfigurable publish-subscribe systems.

In the rest of this section, we present an algorithm that exploits a DHT to build and maintain a tree-shaped overlay network satisfying the requirements of publish-subscribe middleware.

This algorithm is not tied to any specific DHT implementation, but it requires that the DHT satisfy the following properties.

1. *If k_1 and k_2 are mapped onto the same node n, then either all the keys between k_1 and k_2 or all the keys between k_2 and k_1 are also mapped onto n.*
2. *Each key is assigned to one and only one host.*
3. *Each host is aware of the keys it has been assigned to.*

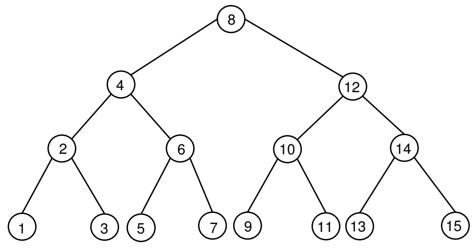
Chord [13] is probably the DHT which most naturally satisfies these requirements, but most other DHTs can be integrated to support them

3.1. Overview

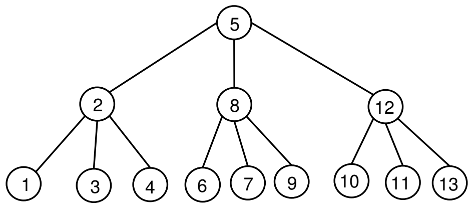
Our approach is based on the following simple idea. We take a predefined tree structure and map each host participating in the overlay onto its nodes, exploiting the distributed hash table. In the following, we will use the terms *node* and *key* to refer to the elements of the common tree structure, and the term *host* to refer to the actual members of the overlay.

Each network host refers to a rooted tree structure like the one depicted in Figure 1(a), in which each tree node is tagged with a distinct value from the set of all possible keys. The host uses the DHT to determine where it should be placed on this tree. More precisely, it retrieves its set of keys and explores the tree structure in breadth-first order until one of them is encountered. This process associates each host with a uniquely determined node of the tree structure, but it does not necessarily associate each tree node to a specific network host. The number of nodes in the tree structure is therefore an upper bound on the number of nodes in the overlay.

Once a host has determined its position, it can discover its neighbors in the overlay by trying to connect to the hosts



(a) A binary tree structure.



(b) A tree structure with a node degree of four.

Figure 1.

associated to neighboring positions in the tree structure. If any of these is not claimed by any host, it simply continues by traversing neighboring positions either ascending towards the root of the tree or scanning the subtrees rooted at the neighbors farther from the root.

3.2. Properties of the Common Tree Structure

In our approach, hosts determine the identity of their neighbors independently of each other and reach an agreement thanks to the use of a common tree structure. In other words, if a node n determines that another node m should be its neighbor, then m, in turn, determines that it should connect to n. Among the various possible alternatives, we will concentrate on tree structures which satisfy the following property.

Let n be a node in the tree structure and p its parent and let $k(n)$ and $k(p)$ denote their keys. Let $k_m = min(k(n), k(p))$ and $k_M = max(k(n), k(p))$, then node n has at most one child node c whose key k_c does not lie in the interval between k_m and k_M.

Additionally, $k_c < k(n)$ if $k(n) < k(p)$ and $k_c > k(n)$ otherwise.

In the case of binary trees (with node degree equal to three), the above property is satisfied by a binary search tree like the one shown in Figure 1(a). For larger values of node degree, it is possible to define tree structures similar to the one in Figure 1(b). The rationale behind this choice is that we aim to build a tree in which the number of neighbors of each host is bounded by the maximum degree of the nodes in the tree. The above property guarantees this fact in the binary case and makes it highly probable in general.

3.3. Mapping hosts onto keys

In order to give a precise definition of the behaviour of our algorithm, it is convenient to describe the mapping between hosts and the nodes of the reference tree structure. This can be done with the following definitions.

Definition 1 *Let h be a host, we define the* keyset *of h, K_h, as the set of keys assigned to h by the DHT.*

Definition 2 *Let K be a set of keys, the* topmost *key in K, $\overline{K}$, is the first key in K which is encountered in a breadth first traversal of the reference tree structure, starting from the root key.*
We also define the topmost *key $\overline{h}$ of a host h to be the topmost key of its keyset K_h, that is $\overline{h} = \overline{K_h}$.*

The concept of topmost key is what enables the mapping between hosts and the reference tree structure. Each host is in fact associated to the node representing its topmost key.

3.4. Algorithm operation

The core of the algorithm is constituted by the operations that are carried out whenever a node joins or leaves the network. A host joining the network determines which other nodes should become its neighbors and establishes a connection with them. These hosts, in turn, recompute their own sets of neighbors and modify their connections. All the hosts that are added to or removed from some other host's neighbor set are forced to recompute their own neighbor sets and operate accordingly. Intuitively, this means that the joining or leaving of a single node may modify significantly the interconnections between hosts. However, our empirical evaluation shows, as discussed in Section 4.2.2, that these changes remain confined to a restricted area.

Each host determines its neighbor set based on the position of its topmost key in the reference tree structure. For the sake of clarity, we first describe the algorithm a host uses to determine its parent and then describe the one it uses to determine its children.

Determining the parent host The parent of a host is defined according to the relationship between keys and their parent keys in the reference tree structure. However, since multiple neighboring keys may be assigned to the same host, it is necessary to define a new way to associate a host's key with the key corresponding to the host's parent.

Definition 3 *Let k be a topmost key, we define $par(k)$ as the closest key to k along the path from k to the root key, such that $par(k)$ is the topmost key of some host and $par(k) \neq k$.*

The foregoing definition enables us to define the parent of a host with topmost key k as the owner of $par(k)$.

A host can easily determine $par(k)$ and, hence, its parent, by means of a simple iterative procedure. It first determines the parent key k_p of k in the reference tree structure. Then it queries the DHT to determine the host h_p which owns k_p, and its keyset. If $k_p = \overline{h_p}$, then $k_p = par(k)$ and h_p is chosen as the parent, otherwise the process is iterated starting from the parent key of k_p.

Determining child hosts Children are determined in a similar way. Let us consider a host h. The host carries out a depth first traversal of the subtree rooted at $\overline{h}$, and locates each topmost key, t, such that the path from t to $\overline{h}$ contains no topmost keys other than t and $\overline{h}$.

Managing neighborhood changes As soon as a host has computed its new set of neighbors, it compares it with its old set. This allows it to notify a neighborhood change to the hosts which are in the symmetric difference of the two sets, triggering the execution of the algorithm at each of these hosts.

3.5. Example

With reference to the binary tree in Figure 1(a) let us consider as an example the set of hosts $\{h_5, h_9, h_{11}, h_{13}, h_{15}\}$[1].

Exploiting the service offered by the DHT, each host is able to determine its own position in the overlay network autonomously without further communication with other hosts. In Figure 2(a) the final network configuration is sketched together with the keyset of each host and its topmost key, shown with a bar on top. Host h_9 discovers it is designated to act as the root since its topmost key $\overline{h_9}$ is equal to 8, which is the root key in the reference tree structure. It then determines its children by querying the DHT for the owners of keys 4 and 12 (hosts h_5 and h_{13}) and creates a link with them.

Analogously, h_5 and h_{13} realize their topmost keys are, respectively, 4 and 12 and that their parent h_p is h_9, since $par(k) = 8$ for both. Besides, h_5 tries to detect its children, starting from its left child. It does not find any topmost key exploring its left subtree (keys 1, 2 and 3 belong to h_5's keyset) and continues by checking whether a right child is available. The first key it encounters is 6, which falls inside K_{h_9}, but is not h_9's topmost key. Therefore, h_5 considers the next keys occurring in depth-first order, namely 5 and 7. Again, these keys are not topmost keys (key 5 is owned by h_5 itself and key 7 is in K_{h_9} but, as described above, $\overline{h_9} = 8$) and so they are discarded. As no other key is available in this sub-tree, h_5 terminates its procedure by connecting only to its parent h_9.

[1]For the sake of clarity we adopt the key distribution scheme of Chord[13] according to which each host is responsible for all the keys ranging from its predecessor (excluded) to itself (included). Nevertheless, as mentioned above, all the schemes satisfying the aforementioned properties can also be used.

Exploring its sub-tree, h_{13} discovers that keys 10 and 14 are the topmost keys of h_{11} and h_{15} and establishes a connection with them. Similarly, h_{11} and h_{15} connect to h_{13}, the owner of their $par(k)$. None of them connects to other nodes as no topmost keys are present in their sub-trees.

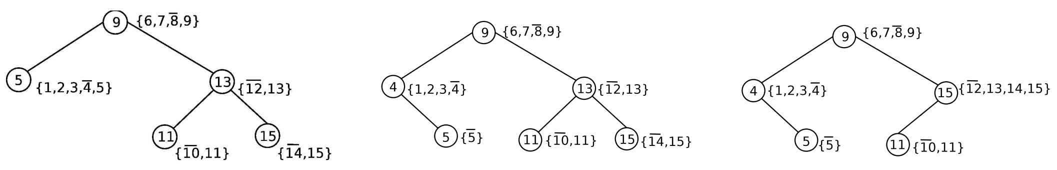
Now suppose that h_4 decides to join the network: it notifies the DHT of its presence and retrieves its keyset $K_{h_4} = \{1, 2, 3, 4\}$ (note that K_{h_5} changes accordingly and now contains only key 5). Its topmost key is 4 and its parent is the owner of key 8, i.e. h_9. As for its children, it cannot find any left child since keys 1, 2, 3 belong to K_{h_4}, while it uses h_5 as right child, since h_5's topmost key is now 6. When contacted, h_9 and h_5 realize that the topology has changed and consequently check whether they have to modify their neighbor sets. In this case, no further modifications are needed (other than the connection to h_4), so no action is performed. The final result is depicted in Figure 2(b). Note that hosts h_{11}, h_{13} and h_{15} are unaffected by the reconfiguration and no computation or message exchange is required by them.

As a last example, let us see what happens if h_{13} fails. Host h_9 detects a disconnection and updates its neighborhood adopting the usual strategy, finally connecting to h_{15} (whose topmost key is now 12). In turn, h_{15} is notified of the topology change and initiates an update procedure that will eventually connect it to h_9 as its parent and to h_{11} as its child. Host h_{11} is also made aware of the topology change and updates its neighbors analogously (see Figure 2(c)).

3.6. Algorithm correctness and properties

The requirements we pose on the DHT, together with the properties of the common tree structure, guarantee that, in a stable network, the algorithm terminates and the procedures which determine parent and child hosts agree on their results.

In addition, the proposed approach provides the ability to control the number of connections that each host establishes in the overlay network. This property helps guarantee the scalability of the middleware by preventing resource-constrained hosts from having too many neighbors. In particular, it can be shown that in the case of a binary tree the number of neighbors of each host is at most as large as the degree of the nodes in the reference tree structure. Con-


9 $\{6,7,\overline{8},9\}$
5 $\{1,2,3,\overline{4},5\}$
13 $\{\overline{12},13\}$
11 $\{\overline{10},11\}$
15 $\{\overline{14},15\}$
9 $\{6,7,\overline{8},9\}$
4 $\{1,2,3,\overline{4}\}$
13 $\{\overline{12},13\}$
5 $\{\overline{5}\}$
11 $\{\overline{10},11\}$
15 $\{\overline{14},15\}$
9 $\{6,7,\overline{8},9\}$
4 $\{1,2,3,\overline{4}\}$
15 $\{\overline{12},13,14,15\}$
5 $\{\overline{5}\}$
11 $\{\overline{10},11\}$

(a) Initial state. (b) Host 4 joins the network. (c) Host 13 disappears.

Figure 2. Sample reconfigurations with nodes being added and removed.

sidering larger values of node degree, we argue that this property still holds with high probability if a uniform key distribution is adopted.

Finally, the evaluation presented in Section 4 highlights the limited impact of our algorithm on the reconfiguration of routing information. Obviously, the actual costs for this reconfiguration and for the delivery of event and subscription messages depend on the protocols used on top of our overlay manager.

4. Evaluation

We implemented the algorithm described in Section 3 using OMNet++[15], a discrete event simulation system. The purpose of our simulations is twofold. Firstly, they serve as a test to verify that the algorithm is indeed able to maintain the desired overlay in a network with hosts that join and/or leave at arbitrary times. Secondly, they provide a measure of its impact on the reconfiguration of routing information in a distributed publish-subscribe system.

4.1. Setting

The scenario we adopted in our simulations consists of a network of hosts which connect and disconnect at random intervals. The number of hosts in the network, N, is kept approximately constant by equating the rates at which connections and disconnections occur. This setting is built incrementally, starting from an empty network and having new hosts join the existing overlay using the protocol described in Section 3. As soon as the network reaches a size of N hosts, the system evolves to a state in which each connection is followed by a disconnection after a specified interval $T_{add,remove}$. Measurements are taken for a time interval T_{meas} from the moment in which the network reaches its size.

The DHT is simulated as an abstract component with the ability to map keys onto hosts and conversely. In particular, we assume that the DHT offers the same consistent view to all the hosts in our algorithm. This corresponds to our algorithm being invoked only after the DHT has stabilized subsequently to the connection or disconnection of a host. Simulations were carried out with a key-space of 256 identifiers, which also determines the size of the reference tree structure, and an actual number of hosts ranging from 10 to 120. In the following, we present results obtained with a binary reference tree. We conjecture that similar conclusions hold for trees with larger values of node degree.

4.2. Results

Our experiments suggest that our approach is a valuable candidate when choosing a topology maintenance algorithm for a distributed publish-subscribe system. Our results are presented as follows. In Section 4.2.1 we evaluate the characteristics of the topology maintained by the algorithm, while in Section 4.2.2 we analyze the impact of our approach on a publish-subscribe system built on top of it.

4.2.1 Correctness and overlay properties

In order to asses the algorithm's ability to maintain the desired overlay network, we let the algorithm run for a time T_{meas} and evaluated the characteristics of the resulting overlay after a period of reconfigurations. More precisely, we first checked that the desired topology was indeed a tree, that is an acyclic and connected graph, and then evaluated the number of neighbors of each host to verify that the resulting node degree was bounded by that of the reference tree structure.

Figure 3 shows the distribution of node-degree obtained after the T_{meas} period. The graph clearly shows that the resulting tree is very close to an ideal balanced tree, with the majority of the hosts either being leaves or having a degree of three, and with none having a larger degree.

4.2.2 Impact on publish-subscribe routing

While controlling the number of neighbors may help in building scalable publish-subscribe middleware on top of an overlay management algorithm, controlling the impact

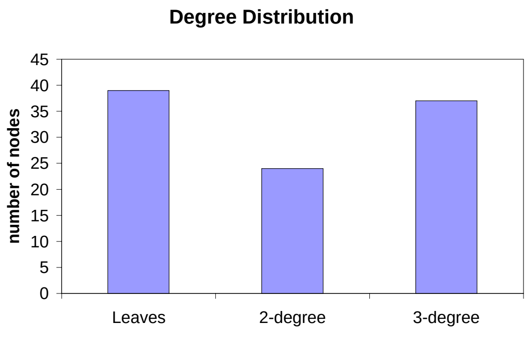


Figure 3. Degree distribution in the case of a binary reference tree

of failures, and more in general of topology changes, is of paramount importance when building middleware for highly dynamic environments.

Characterizing the impact of topology changes An ideal overlay manager should handle the disconnection or the connection of a host by minimizing the number of hosts affected by the event. Previous work [5] has precisely characterized the hosts in the overlay network that may be affected by a topology change in cases when a link is replaced by another. The characterization may be extended to the case of host connections and disconnections in the following way. We consider a *reconfiguration area* consisting of the subtree interconnecting the endpoints of the links that are removed or created as a result of a topology change, once the algorithm has stabilized. This characterization, albeit independent of the specific reconfiguration protocol, provides us with an estimate of the overhead associated with the reconfiguration of subscription information.

Impact evaluation The definition of reconfiguration area allows us to assess the goodness of our algorithm with respect to its application in reconfigurable publish-subscribe middleware. The results obtained in our simulations, depicted in Figure 4, show that the size of the reconfiguration area remains very low even when the number of hosts becomes large.

The size of the reconfiguration area can be compared with the number of hosts which are directly involved in the topology change. In the ideal case, the reconfiguration area should only consists of the appearing or disappearing host and its neighbors. Our algorithm is very close to this ideal case with the average number of hosts contacted during a reconfiguration being between two and three.

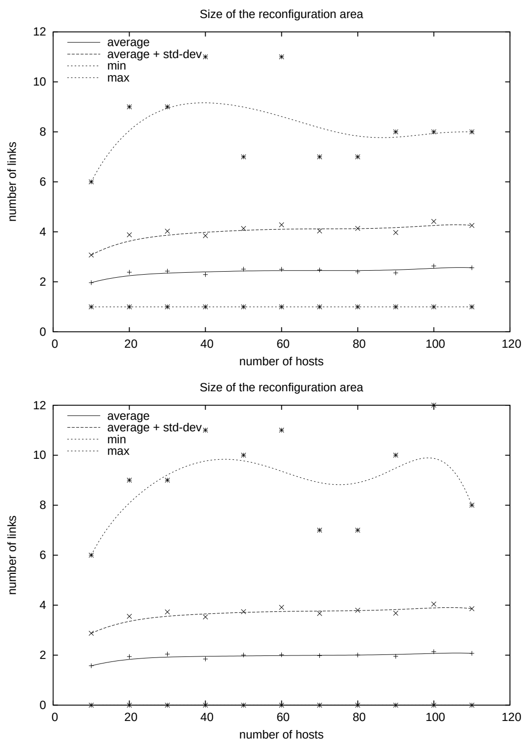


Figure 4. Size of the reconfiguration area in the case of host connections (top) and disconnections (bottom).

5. Related Work

The vast majority of systems targeted to multipoint communication rely on a tree-shaped overlay network to deliver events but only very few of them provide details on how the tree is rearranged when a topological reconfiguration occurs [7, 6].

Some systems [17, 2, 10] leverage a DHT service to maintain the tree in presence of faults. By and large all of them adopt a *rendezvous-based* scheme: a specific key identifies the root and the host responsible for that key is universally recognized as the root. To join the network each host asks the DHT for a valid route to the root and then forwards a join request to the next host towards the root. When a host receives a join request, it records its originator as its child and checks whether it is already member of the tree. In this case, it simply drops the request, otherwise it forwards the request towards the root, acting as the originator.

When a host detects a failure in the path leading to the root, it initiates a new join procedure and eventually, after DHT reconfigures itself, it is provided with a new host to attach to. If the root fails, a new host becomes responsible for the root key and all the other hosts have to start a new join procedure.

Though this approach exploits the reconfigurability of DHTs, both the techniques they use and the results they achieve are quite different from ours. First, such approaches do not enable the definition of the desired topology as we do: this implies that the resulting topology is predefined, dependending on the specific DHT they rely on. Second, they give no hints about the impact that a disappearing host has on the rest of the network. However, it seems likely that the closer the host is to the root, the more hosts need to modify their routing tables. The extreme case in which the root disconnects forces all the hosts in the network to initiate a join procedure, thus flooding the network with requests.

Another related approach is Willow [14]. This system adopts a strategy similar to that of Kademlia [8]: it treats hosts as leaves in a binary tree, with each hosts's position being determined by the sequence of bits in its ID.

6. Conclusions

The maintenance of a tree of interconnected dispatchers is a key issue in reconfigurable publish-subscribe middleware that has not been thoroughly investigated by current research. In this work we presented an algorithm which maintains the topology at the basis of a distributed publish-subscribe system, exploiting the potentialities of distributed hash tables. The algorithm exhibits some interesting properties, including the ability to control the number of neighbors of the hosts in the resulting overlay network, and a very low impact on the reconfiguration process dictated by topology changes. These properties were validated through simulations which confirm the validity and the potentiality of the proposed approach. Finally, although we described the protocol in the context of publish-subscribe middleware, our work can be applied to other instances of group communication as well as to any system which requires the maintenance of an application-level tree-based overlay network.

Our plan is to investigate to a further extent the characteristics of this algorithm both from an empirical and from a theoretical point of view, and to test it in a prototype implementation of a publish-subscribe platform that is currently under development.

References

[1] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, Aug. 2001.

[2] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002.

[3] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation. In *Proc. of the 24th Int. Conf. on Distributed Computing Systems*, 2004.

[4] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. on Software Engineering*, 27(9):827–850, Sept. 2001.

[5] G. Cugola, D. Frey, A. Murphy, and G. Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *19th Annual ACM Symp. on Applied Computing (SAC'04)*, 2004.

[6] P. Francis. *Yoid Tree Management Protocol (YTMP) Specification*. ACIRI, April 2000. `http://www.icir.org/yoid/docs/ytmp.pdf`.

[7] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, pages 197–212, October 2000.

[8] P. Maymounkov and D. Mazires. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, pages 53–65, 2002.

[9] G. Picco, G. Cugola, and A. Murphy. Efficient Content-Based Event Dispatching in Presence of Topological Reconfiguration. In *Proc. of the 23rd Int. Conf. on Distributed Computing Systems*, May 2003.

[10] P. Pietzuch and J. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proc. of the 1st Int. Wkshp on Distributed Event-Based Systems*, July 2002.

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

[12] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 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 the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01)*, 2001.

[14] R. van Renesse and A. Bozdog. Willow: Dht, aggregation, and publish/subscribe in one protocol. In *Proceedings of the Third International Workshop on Peer-to-Peer Systems*, pages 173–183, 2004.

[15] A. Varga. OMNeT++ Web page. `www.omnetpp.org`.

[16] B. Y. Zhao, L. Huang, S. C. Rhea, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[17] S. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Network and Operating System Support for Digital Audio and Video, 11th International Workshop, NOSSDAV 2001, Port Jefferson, NY, USA, June 25-26, 2001, Proceeding*, pages 11–20.