Content-based Publish-Subscribe Over Structured P2P Networks

Peter Triantafillou and Ioannis Aekaterinidis
Research Academic Computer Technology Institute and
Department of Computer Engineering and Informatics, University of Patras, Greece
{peter,aikater}@ceid.upatras.gr

Abstract

In this work we leverage the advantages of the Chord DHT to build a content-based publish-subscribe system that is scalable, self-organizing, and well-performing. However, DHTs provide very good support only for exact-match, equality predicates and range predicates are expected to be very popular when specifying subscriptions in pub/sub systems We will thus also provide solutions supporting efficiently subscriptions with range predicates in Chord-based pub/sub systems.

1. Introduction

Publish/subscribe systems are becoming very popular for building large scale distributed systems/applications. The main functionality of pub/sub systems is the delivery of published notifications from every producer (publisher) to all interested consumers (subscribers). Publishers, who are completely unaware of the existence of the consumers, publish events (information) through the system by specifying the values of a set of well defined attributes. The consumers are expressing their interest through appropriate subscriptions and wait until they are informed about a matching event. A publish/subscribe infrastructure is responsible for matching events to related subscriptions and delivering the matching events to interested consumers.

Building a centralized publish/subscribe system has the advantage of having a global image of the system and thus making the matching algorithm much easier to implement. This approach suffers from scalability problems as the number of publications and subscriptions increases. Thus, the decentralized approach is more appropriate. The main challenge in a distributed environment is the development of an efficient distributed matching algorithm.

The peer-to-peer (p2p) paradigm is appropriate for building large-scale distributed systems/applications. P2p systems are completely decentralized, scalable, and self-organizing. A popular class of them is the "structured" p2p systems. The most prominent of these systems are built using a Distributed Hash Table (DHT [7],[8],[9]), which is a mechanism that provides scalable resource look-up/routing.

2. Related work

2.1 Publish/Subscribe systems

There are two types of publish/subscribe systems: i) *topic based* and ii) *content based*. Topic based systems are much like newsgroups. Users express their interest by joining a group (a topic). Then all messages related to that topic are broadcasted to all users participating to the specific group.

Content-based systems are preferable as they give users the ability to express their interest by specifying predicates over the values of a number of well defined attributes. The matching of publications (events) to subscriptions (interest) is done based on the content (values of attributes).

Publish/subscribe systems can be built in a distributed manner, avoiding the lack of scalability and fault-tolerance of centralized approaches. Distributed solutions are mainly focused on topic-based publish/subscribe systems [1], [2], [3]. Some attempts on distributed content-based publish/subscribe systems use routing trees to disseminate the events to interested users based on multicast techniques [4], [5], [15], [16]. Some other attempts use the notion of rendezvous nodes which ensure that events and subscriptions meet in the system [14].

Some approaches have also considered the coupling of topic-based and content-based systems. The authors in [6] used a topic-based system (Scribe [1]) that is implemented in a decentralized manner using a DHT (Pastry [7]). In their approach the publications and the subscriptions are automatically classified in topics, using an appropriate application-specific schema. A potential drawback of this approach is the design of the domain schema as it plays fundamental role in the system's performance. Moreover, it is likely that false positives may occur.

2.2 Chord and DHTs

Distributed Hash Tables (DHTs [7], [8], [9], [11]) have been adopted to create peer-to-peer data networks. In a DHT each node has a unique identifier (nodeID) selected from a very large address space. Each message can be associated with a key which is a unique identifier of the same type as nodeID. The main functionality of a DHT is the following: giving a (message, key) pair the system locates (routes the message) to the node whose nodeID is numerically closest to that key. DHTs ensure that routing requires $O(\log(N))$ hops to locate/store a message (where N is the maximum number of nodes in the network).

Chord [9] is a fairly simple structured peer-to-peer network based on a DHT. Compared to unstructured peer-to-peer networks like Gnutella [12] and MojoNation [13] where neighbors of peers are defined in rather ad hoc ways, Chord is structured because of the way peers define their neighbors. Chord provides an exact mapping between node identifiers (nodeID) and keys associated with messages using consistent hashing **Error! Reference source not found.**. NodeIDs and keys are mapped to a large circular identifier space, e.g. $0\ldots2^{160}$ for 160-bit IDs. Values in this space can be viewed as positions in the ring defining the name/identifier space. Thus, given a key, Chord maps it to the (ring position) node whose nodeID is equal to the key. If this node does not exist, the key is mapped to the first successor of this node on the ring. This node is called the *successor* of the key.

Chord efficiently determines the successor of an identifier (key) in $\frac{1}{2}\log(N)$ hops on average (and in $O(\log(N)$ hops in the worst case). In the steady state each node maintains routing information of up to $O(\log(N))$ other nodes. Adding or removing a node from the network can be achieved at a cost of $O(\log^2(N))$ messages. Chord has become very popular and has been used as a building block for several large-scale distributed systems.

2.3 Contribution: Publish/Subscribe systems over DHTs

We choose to use Chord because of its simplicity and popularity within the peer-to-peer community. We leverage the advantages of Chord to build a content-based publish-subscribe system that is scalable, self-organizing, and well performing.

Furthermore, although DHTs provide very good support for exact-match equality predicates (i.e. find the node storing the item with id=itemID) they do not provide good support for range predicates (which are typically very popular when specifying subscriptions in pub/sub systems). We will show how to build Chord-based pub/sub systems which can support range predicates. We will first provide a startup solution and will then extend the Chord substrate to further improve the performance of matching events against subscriptions with range predicates. As far as we know, this is the first work that: (i) leverages DHT research to build large scale content-based pub/sub systems and (ii) while supporting subscriptions with range predicates efficiently.

Event 1					
Type	*Min* $v_{min}(\cdot)$	*Max* $v_{max}(\cdot)$	*Precision* $v_{pr}(\cdot)$	*Name*	*Value* $v(\cdot)$
string	-	-	-	Exchange	=NYSE
string	-	-	-	Symbol	=OTE
float	0.0	20.0	0.01	Price	=8.40
float	0.0	20.0	0.01	High	=8.80
float	0.0	20.0	0.01	Low	=8.22

Figure 1. An event example.

Subscription 1	
Name	*Value*
Exchange	=NYSE
Symbol	=OTE
Price	<8.70
Price	>8.30

Subscription 1	
Name	*Value*
Symbol	=OTE
Price	=8.20
Low	<8.05

Figure 2. Example with two subscriptions.

3. Publish/Subscribe over Chord

The Event/Subscription Schema.
The event schema of this model (Figure 1) is a set of typed attributes. Each attribute a_i consists of a type, a name and a value $v(a_i)$. The type of an attribute belongs to a predefined set of primitive data types commonly found in most programming languages. The attribute's name is a simple string, while the value can be in any range defined by the minimum and maximum ($v_{min}(a_i)$, $v_{max}(a_i)$) values along with the attribute's precision $v_{pr}(a_i)$.

The subscription schema is more general (Figure 2), allowing to express a rich set of subscriptions which contain all interesting subscription-attribute data types (such as integers, strings, etc) and all common operators (=, ≠,<, >, etc.). An event matches a subscription if and only if all the subscription's attribute predicates/constraints are satisfied. A subscription can have two or more constraints for the same attribute which can be thought as if we had two or more different subscriptions with unique constraints over their attributes. Finally, an event can have more attributes than those mentioned in the subscription attributes.

The Subscription Identifier.
A subscription id is the concatenation of three parts:

1. c1: The id of the node receiving the subscription (i.e., where the subscription "belongs"). The size of this field is m bits in a Chord ring with an m-bit identifier address space.
2. c2: The id of the subscription itself. The size of this field in bits is equal to the rounded-up base-2 logarithm of the maximum number of outstanding subscriptions a node can have (e.g. if each node needs to manage 1,000,000 of subscriptions, c2 will be 20-bits long).
3. c3: The number of attributes on which constraints are declared. The maximum value of this field is equal to the total number of attributes supported by the system.

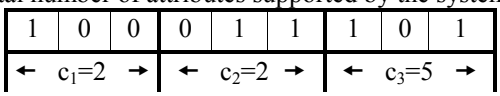

Figure 3. An example subscription id (subID).

Assume an example Chord ring with a 3-bit identifier address space. Each node can support 8 outstanding subscriptions with an attribute schema including 7 attributes. The subscription id depicted in Figure 3 identifies subscription 3 (c_2=3), belonging to node 4 (c_1=4), comprised of constraints on 5 attributes (c_3=5).

We should note that for every subscription there is a node in the network storing metadata information for it. That node is identified by the c_1 field of the subscription id and it keeps metadata information about the subscription (for example the IP address of the user that generated the subscription etc.).

3.1 Processing subscriptions

Consider, for simplicity, that there is an example pub/sub system supporting only one attribute (a_1). In general, subscriptions specify a single value or a range of values for the attribute a_1. The main idea behind our approach is to store the subscription ids into those nodes of the Chord ring that were selected by appropriately *hashing the values of the attributes* in the subscriptions. The matching of an incoming event can be performed simply by asking those nodes for stored subscription ids.

3.1.1 Storing subscriptions. Storing subscription is done using the hash function provided by Chord (later we will change this to improve performance). Consider that this hash function h() (e.g., SHA-1) returns an identifier uniformly distributed in the address space used for node identifiers. Thus, the result of this hash function h() for the value v(a_i) of the attribute a_i is k (k=h(v(a_i))).

In the case where a subscription contains an equality operator on the single attribute of the example schema, we place the subscription id at the node whose id is the least id which is equal or greater to k (that is *successor*(k) from the Chord API). Therefore, the subID will be placed at the following node: *successor*($h(v(a_i))$).

Things are slightly more complicated with ranges of values. In this case, we map the range on the Chord network storing the subID to all the mapped nodes. Suppose that there is a subscription declaring a range of values over the attribute, a_i which is defined to be between $v_{low}(a_i)$ and $v_{high}(a_i)$. Since all values between $v_{low}(a_i)$ and $v_{high}(a_i)$ are finite, e.g. n (remember that the attribute a_i was declared with a specific precision $v_{pr}(a_i)$), we follow n steps and at each step we store at a Chord node, which is chosen by hashing the previous value incremented by the precision *step*, the subID of the given subscription (the algorithm can be seen in Figure 4).

subID : subscription id
$\boldsymbol{a_i}$: attribute i , $\boldsymbol{L_{ai}}$: List of subIDs for attribute a_i
$\boldsymbol{v_{pr}(a_i)}$: precission of attribute a_i
$\boldsymbol{v(a_i)}$: value of attribute a_i, ***h()*** : Chord hash function

1. For every attribute a_i in subscription
2. if a_i has equality constraint
3. store *subID* in node = *successor*($h(v(a_i))$) in the L_{ai} list.
4. else if a_i has a constraint in $[v_{low}(a_i), v_{high}(a_i)]$
5. $v = v_{low}(a_i)$
6. while $v \leq v_{high}(a_i)$
7. store *subID* in node = *successor*($h(v)$) in the L_{ai} list.
8. $v = v + v_{pr}(a_i)$

Figure 4. The procedure of storing subscriptions

If our schema consists of many attributes, we follow the above procedure for each attribute in every subscription. The only difference is that we keep a different list of subscription ids at each Chord node for every attribute in our schema. For example, consider the case of subscription 1 of Figure 2. The attributes are being processed one at a time starting with *Exchange*. The subscription id of subscription 1, say $subID_1$ will be placed at *successor*(h("NYSE")) node in the list dedicated for attribute *Exchange* and at *successor*(h("OTE")) node in the list dedicated for attribute *Symbol*. As you can see, there is a range constraint over the *Price* attribute, 8.30<*Price*<8.70. Since the precision of the attribute *Price* is defined to be 0.01, the $subID_1$ will be placed in 39 Chord nodes defined by *successor*($h(v_j(Price))$) for the following sequence of values 8.31, 8.32, …, $v_j(Price)$, …, 8.68, 8.69.

3.1.2 Updating subscriptions. Updating a subscription involves a procedure during which the values of all attributes contained in the subscription are updated using the standard API of the Chord system. In the case of

equality only two nodes are affected. First, the node that is mapped to the old, stale, value is forced to delete the subID for the attribute that belongs to the subscription with identifier subID. Second, a new node is going to store the subID, depending on the id returned from the Chord's hash function passing the new updated value. In other words, we delete the subID from the node with nodeID=successor(h($v_{stale\_value}(a_i)$)) and then we add it to the node with nodeID = successor(h($v_{updated\_value}(a_i)$)).

***subID**$_i$* : subscription id of subscription **i**
***N**$_{a\text{-}event\text{-}i}$* : number of attributes defined in the event **i**
***N**$_{a\text{-}sub\text{-}i}$* : number of attributes defined in a subscription **i**
***N**$_{list\text{-}sub\text{-}i}$* : number of collected lists that subID$_i$ is stored
***a**$_i$* : attribute **i**
***L**$_{ai}$* : List of subIDs for attribute a_i
v(a**$_i$**) : value of attribute a_i
h() : Chord hash function
Delivery List: the list of subIDs that the event is going to be delivered.

1. for every event$_j$ arriving at the system
   subIDs lists collection phase
2. for every attribute a_i in event$_j$ with value v(a_i)
3. go to node *successor*(*h*(v(a_i)))
4. retrieve the list L_{ai} of subIDs found there
   Matching phase
5. for every list L_{ai} of subIDs
6. for every subID$_k$
7. retrieve the number of attributes defined from the c_3 field: $N_{a\text{-}sub\text{-}k}$
8. count the number of collected lists that subID$_k$ is stored: $N_{list\text{-}sub\text{-}k}$
9. if $N_{a\text{-}sub\text{-}k}$ equals $N_{list\text{-}sub\text{-}k}$
10. we have a match
11. remove subID$_k$ from all lists
12. store subID$_k$ in the *Delivery List*
    Delivery phase
13. for every subID$_i$ in the *Delivery List*
14. contact the node that keeps the subscription. Its nodeID is the c_1 field of the subID$_i$
15. retrieve the IP of the client that generated the subscription with id : subID$_i$
16. deliver the event to the client

Figure 5. The matching algorithm.

As we said before, ranges are spread all over the Chord ring. Thus, updating a range (in other words updating the $v_{low}(a_i)$ and $v_{high}(a_i)$ values) results in following the above procedure for all Chord nodes that store the subID for the given range of values. The procedure we follow depends on the new values of the range bounds ($v_{low\_NEW}(a_i)$ and $v_{high\_NEW}(a_i)$) compared to the old ones. If $v_{low\_NEW}(a_i) < v_{low}(a_i)$ we store the subID to a number of nodes that are going to cover the $[v_{low\_NEW}(a_i), v_{low}(a_i))$ range. The same procedure holds when $v_{higb\_NEW}(a_i) > v_{high}(a_i)$ resulting in covering the range $(v_{high}(a_i), v_{higb\_NEW}(a_i)]$. In the case where $v_{low\_NEW}(a_i) > v_{low}(a_i)$ or $v_{higb\_NEW}(a_i) < v_{high}(a_i)$ we delete the subID form the nodes covering the range $[v_{low}(a_i), v_{low_NEW}(a_i))$and $(v_{high\_NEW}(a_i), v_{higb}(a_i)]$ respectively.

Deleting subscriptions is done as explained above since the updating includes the deleting procedure.

3.2 Processing events: The matching algorithm

The distributed matching algorithm should be able to cope with expected very high loads, determined by high event arrival rates.

Suppose that an event arrives at the system with $N_{a\text{-}event}$ attributes defined. The matching algorithm starts by processing each attribute separately. It first tries to find the node which stores subIDs for the value v(a_i) of the attribute a_i. This node is the n=*successor*(*h*(v(a_i))). The algorithm, then, stores the list of unique subIDs found to be stored in node n in the list L_{ai} designated for a_i. After processing all attributes, $N_{a\text{-}event}$ lists of subIDs will be stored. Suppose, now, that a subID$_k$ presented in at least one of those lists consists of $N_{k\text{-}sub}$ attributes ($N_{k\text{-}sub}$ can be easily derived from the field c_3 of the subID defined in section 3). Then, the subID$_k$ matches the event if it appears in exactly $N_{k\text{-}sub}$ lists collected from the Chord ring. The matching algorithm can be seen in Figure 5.

Example: Matching events with subscriptions.

Suppose that we have the subscriptions of Figure 2 generated by two clients connected to a Chord node and the event of Figure 1. First, the algorithm will collect all the subIDs lists in which the values of the event attributes, satisfy the corresponding constraints of the subscriptions.

For this to be done, the algorithm starts with attribute *Exchange* and retrieves the subID list ($L_{Exchange}$) from node *successor*(*h*("NYSE")). This list contains only the subID$_1$. Hence, we have $L_{Exchange}$**→subID$_1$**. For the attribute *Symbol* the corresponding list is L_{Symbol}**→subID$_1$, subID$_2$**, since both subscriptions are satisfied for the specific event. For the attribute *Price* only subscription 1 is satisfied and, thus, the list is L_{Price}**→subID$_1$**. Finally, for the attribute *Low* only subscription 2 is satisfied and the list is L_{Low}**→subID$_2$**.

After this phase of the matching process the collected subscription ID lists are:

$L_{Exchange}$……. **→subID$_1$**
L_{Symbol}…… **→subID$_1$, subID$_2$**
L_{Price}…….. **→subID$_1$**
L_{Low}…….. **→subID$_2$**

Subscription 1 was found in three lists while subscription 2 was found in two lists. By processing appropriately the subIDs of subscriptions 1 and 2 (the c_3 part) we can find out that both subscriptions have constraints over three attributes. Since subscription 1 was found in three lists, a match is implied and so we keep the $subID_1$ in order to inform the node which generated the subscription about the matched event. This is done by consulting the node storing the subscription (with nodeID equal to the c_1 field of the $subID_1$) and holding metadata information for $subID_1$, in order to locate the IP address of the client that generated the subscription. Then, the event is delivered to the interested client. Subscription 2 on the other hand is dropped since the number of lists that the $subID_2$ was found in is 2 while the number of attributes defined in it is 3.

3.3 Expected performance

In a Chord network with N nodes and 2^m-bit address space the average number of nodes that must be contacted to find a *successor* is ½log(*N*) hops.

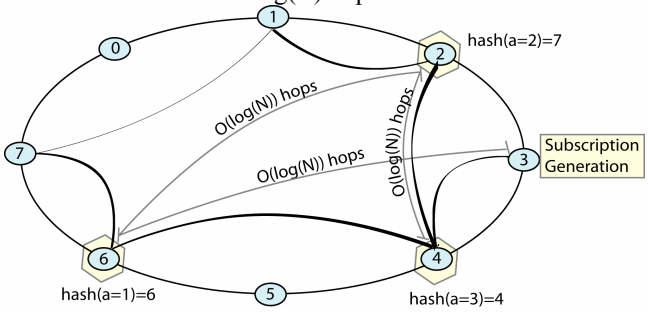

Figure 6. Storing range values with Chord.

During the subscription storage procedure, the average number of hops needed to store a subID depends on the type of constraints over the attributes. In equality constraints, the average number is ½log(*N*), since the subID is stored in a single node, i.e. the *successor*(*h*(v(a_i))). When the constraint is a range of values (e.g. [$v_{low}(a_i)$, $v_{high}(a_i)$]) over the attribute a_i with precision $v_{pr}(a_i)$ (in Figure 1 the v_{pr}(*Price*) of attribute *Price* is 0.01) then $r = \frac{v_{high}(a_i) - v_{low}(a_i)}{v_{pr}(a_i)}$ nodes are affected leading to r·½log(*N*) hops on average in order to store the subID.

The update/deletion of subscription again depends on the type of constraints over the attributes. For an equality constraint, an update can be performed by contacting 2·½log(*N*)=log(N) nodes. For ranges the number of nodes is k·log(*N*) on average, where k depends on whether the new range is smaller or wider compared to the previous one.

The event-processing (matching) process involves contacting $N_{a\text{-}event}$·½log(*N*) nodes to collect the subscription ids lists. Thus, we see that, by design, our proposal leads to fast and scaleable event matching.

4. Improving performance

When trying to store a subscription over the Chord ring with attributes defined by a range of values, we need perform r·½·log(*N*) hops on average for every attribute (note that r depends on the precision of the value as well as the $v_{low}(a_i)$, and $v_{high}(a_i)$, values of the range interval). In this section we extend the Chord's functionality so that range attributes will require r+½·log(*N*) hops.

4.1 OPChord : Order Preserving Chord

We use a 2^m-order preserving hash function (OPHF) in order to store the sequential values of a range interval in sequential nodes over the Chord ring.

Expected performance.

We need to perform ½·log(*N*) hops on average to locate the node which will store the minimum value of the range (that is $v_{low}(a_i)$ for the attribute a_i). Then, we have to perform r hops to store the remaining values in the range. This approach leads to r+½·log(*N*) hops in total.

The Order Preserving Hash Function.

Suppose, now, that every attribute a_i is characterized by $v_{min}(a_i)$: the minimum value that a_i can take, $v_{max}(a_i)$: the maximum value that a_i can take, and $v_{pr}(a_i)$: the precision of a_i . If $v_j(a_i)$ is defined to be any value in the interval [$v_{low}(a_i)$,$v_{high}(a_i)$], the OPHF is:

$$h(v_j(a_i)) = \left(s_o(a_i) + \frac{v_j(a_i) - v_{\min}(a_i)}{v_{\max}(a_i) - v_{\min}(a_i)} \cdot 2^m \right) \bmod 2^m$$

The $s_o(a_i)$ is defined to be:

$$s_o(a_i) = hash(attribute\_name(a_i))$$

and is used to randomize the node on the Chord ring where the minimum values of different attributes will be stored, leveraging thus different subsets of the Chord network. *Hash*() is the base hash function used by Chord (e.g. SHA-1). Note that there is a different OPHF for every attribute.

4.2 Subscription and event processing with OPHF

The algorithms are generally the same as the ones presented earlier. The only main difference is the use of OPHF instead of the base hash function of Chord.

Example: Storing subscription.

Consider a Chord ring with 3-bit identifiers and 8 nodes and a subscription of a single integer attribute *a* arriving at node 3 with constraint: 0< v(*a*) <4. Using Chord (Figure 6) would require *O*(r·log(*N*)) hops to store the subID at three

nodes (in this example r equals 3, as there are 3 distinct values in the interval (0,4)), Hashing the first value (a=1) returns node 6 requiring to access $O(\log(N))$ nodes to reach node 6 (½·log(N) on average). Repeating the previous step, the other nodes that will store the subID are 2 and 4, requiring overall $O(r \cdot \log(N))$ hops at most (in our example, 6 hops).

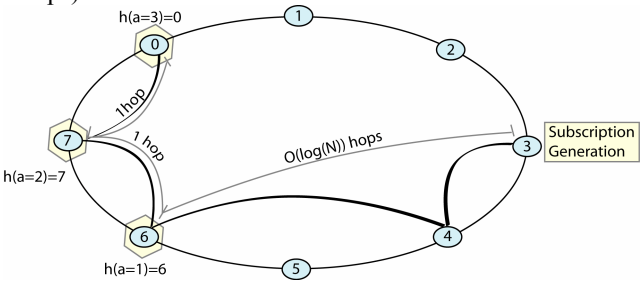

Figure 7. Storing range values with OPHF/Chord.

Suppose, now, that we use OPHF/Chord (Figure 7). We need to perform $O(\log(N))$ hops only once at the very first time when trying to reach the first node (node 6). Then, storing the subID at nodes 7 and 0 requires two more hops.

4.3 Discussion

Load balancing in the Chord system is based on the randomness guarantees of the consistent hashing function. We have investigated load balancing within the OPHF architecture, but it is beyond the scope of this paper.

We should also briefly note the "small domain problem": when the number of nodes in the network is much greater than the domain of attribute values, this could lead to have k "useless" nodes between two consecutive (ring positions) values in the range. In this case we need to pay an overhead of extra hops in order to store subIDs for a range of values, in the OPHF design. We have developed solutions that alleviate the extra-hop problems; however, they are beyond the scope of this paper.

5. Concluding remarks

In this work we have shown how to leverage a popular DHT, Chord, towards building scalable, self-organizing, well-performing, content-based pub/sub systems. We have shown how to support subscriptions that involve equality and range predicates and the associated performance benefits. Our proposal favors fast and scaleable event processing. This is achieved by essentially turning the task of event processing into a DHT lookup operation. To our knowledge this is the first work that meets these goals.

6. References

[1] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron., *Scribe: A large-scale and decentralized application-level multicast infrastructure.* Journal on Selected Areas in Communication, vol. 20, Oct. 2002.

[2] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. *Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination.* 11th ACM NOSSDAV, pp. 11-20, '01.

[3] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. *Application-level multicast using content-addressable networks*. In Proc. 3rd International Workshop of NGC, vol. 2233, pages 14–29. LNCS, Springer, 2001.

[4] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. *An efficient multicast protocol for content-based publish-subscribe systems*. In Proceedings of the 19th ICDCS, pp. 262–272, 1999.

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

[6] D. Tam, R. Azimi, H. Jacobsen, *Building Content-Based Publish/Subscribe Systems with Distributed Hash Tables*, Int. Workshop on Databases, Information Systems and Peer-to-Peer Computing, September 2003.

[7] A. Rowstron and P. Druschel. *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*. In Proc. 18th IFIP/ACM Int. Conference on Distributed Systems Platforms (Middleware 2001), pages 329-350, November 2001.

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

[9] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. *Chord: A scalable peer-to-peer lookup service for internet applications*. In proceedings of ACM SIGCOMM 2001, pages 149-160, 2001.

[10] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. *Achieving scalability and expressiveness in an Internet-scale event notification service*. Proc. ACM PODC, pp 219–227, 2000.

[11] Zhao, Y. B., Kubiatowitcz, J., Joseph, A.: *Tapestry: An infrastructure for fault-tolerant wide-area location and routing*. Tech. Rep. UCB/CSD-01-1141, Univ. of California at Berkley, Computer Science Dept. (2001)

[12] Gnutella: http://gnutella.wego.com

[13] Wilcox, B., Hearn, O.: *Experiences Deploying a Large-Scale Emergent Network*. In 1st International Workshop on Peer-to-Peer Systems, IPTPS'02 (2002)

[14] P. R. Pietzuch and J. Bacon, *Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware*, in proceedings of the DEBS'03 conference, 2003.

[15] P. Triantafillou, A. Economides, *Subscription Summarization: A New Paradigm for Efficient Publish/Subscribe Systems*, In IEEE, ICDCS 2004.

[16] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann, *A Peer-to-Peer Approach to Content-Based Publish/Subscribe,* in DEBS 2003.