A Self-Organizing Crash-Resilient Topology Management System for Content-Based Publish/Subscribe

R. Baldoni, R. Beraldi, L. Querzoni and A. Virgillito
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”
Via Salaria 113, 00198, Roma, Italy
email: {baldoni, beraldi, querzoni, virgi}@dis.uniroma1.it

Abstract

Content-based routing realized through static networks of brokers usually exhibit nice performance when subscribers with similar interest are physically close to each other (e.g. in the same LAN, domain or in the same geographical area) and connected to a broker which is also nearby. If these subscribers are dispersed on the Internet, benefits of such routing strategy significantly decrease. In this paper we present a Topology Management System (TMS), a component of a content-based pub/sub broker. The aim of a TMS is to mask dynamic changes of the brokers' topology to the content-based routing engine. TMS relies on a self-organizing algorithm whose goal is to place close (in terms of TCP hops) brokers that manage subscribers with similar interests keeping acyclic the topology and without compromising network-level performance. TMS is also resilient to broker failures and allows brokers join and leave.

1. Introduction

Content-based publish/subscribe (pub/sub) communication systems are a basic technology for many-to-many event diffusion over large scale networks with a potentially huge number of clients. Scalable solutions are obtainable only deploying a distributed event system made up of a large number of cooperating processes (namely brokers) forming a large scale application level infrastructure.

Current state-of-the-art content-based systems ([3], [4], [6] and [10]) work over networks of brokers with acyclic and static topologies. These systems exploit the content-based routing algorithm (CBR) introduced in SIENA [3] for efficient event diffusion. By making a mapping between the brokers' topology and the subscriptions present at each broker, CBR avoids an event to be diffused in parts of the network where there are no subscribers interested in, thus reducing the number of TCP hops experienced during the event diffusion toward all the intended subscribers. However, such a filtering capability performs at its best only when each broker presents a high physical similarity among its subscriptions (i.e., all subscribers sharing similar interest are physically connected to the same broker). On the contrary, when subscribers sharing similar interests are connected to a dispersed subset of distinct brokers, which are far from each other in terms of hops, the performance gain obtained using the CBR algorithm, with respect to an algorithm that simply floods the network with events, becomes negligible. Therefore such performance gain depends from the distribution of the subscriptions in the brokers' network, which is usually not under control.

In this paper we present a self-organizing topology management system (TMS) for content-based publish/subscribe. TMS encapsulates all the logic associated with the management of a dynamic network of brokers. The main aim of a TMS system is to put a CBR algorithm working at its best condition of efficiency. Therefore, TMS dynamically rearranges TCP connections between pair of brokers in order to put subscribers sharing similar interests as close as possible, even though they are connected to distinct brokers, thus creating the conditions for a reduction of TCP hops per event diffusion. Unfortunately, on TCP/IP networks reducing an application level metric (like TCP hops) can lead to poor performance at the network level. TMS exploits network-awareness capabilities to ensure that a reconfiguration is made permanent only if it does not harm network-level performance.

TMS also includes a module to heal the network after joining/leaving/crashing of brokers in order to maintain an acyclic topology. In general, such a reconfiguration is confined in the neighborhood of the failed node. In this sense, the system provides local self-healing capabilities. Let us finally remark that content-based systems like ([3], [4] and

[6]) require human intervention whenever the brokers' network topology has to be changed in order to add or remove brokers.

In the rest of this paper we give a general overview of the TMS, explaining its features. The paper is structured as follows. Section 2 presents the general architecture of a content-based publish/subscribe system. Section 3 describes the internal architecture of our new Topology Management System. Section 4 describes the details of the algorithms used to handle topology maintenance and self-organization. Section 5 compares our work with the related works in this research area. Finally, Section 6 concludes the paper.

2. Content-Based Pub/Sub Background

This Section defines the key concepts related to a content-based publish/subscribe system and the architecture of a generic broker. Such a system manages subscriptions and events defined over a predetermined event schema, constituted by a set of n *attributes*. An event is a set of n values, one for each attribute, while a subscription is defined as a set of *constraints* over attributes.

A content-based pub/sub system is structured as a set of processes, namely *event brokers*, $\{B_1, .., B_N\}$ interconnected through transport-level links which form an acyclic topology. In particular each broker maintains a set of open connections (*links*) with other brokers (its neighbors). The link connecting broker B_i with broker B_j is denoted as $l_{i,j}$ or $l_{j,i}$. Each broker may have only a bounded number of neighbors (its maximum fan-out F) to ensure that the number of concurrent open connections does not harm process performance.

The internal architecture of a generic broker is depicted in figure 1; each broker is composed by five components:

Subscription table. This table contains all the subscriptions issued by subscribers to the broker. We define as the *zone of interest* of a broker B_i, denoted Z_i, the union of all local subscriptions at B_i.

Pub/Sub manager. A broker can be contacted by clients, that, depending on their role, can be divided in *subscribers* or *publishers*. Publishers produce information by firing events to one of the brokers, while subscribers express their interests for specific events by issuing subscriptions to one of the brokers. The Pub/Sub manager has the responsibility of managing message exchanges with clients (i.e. publishers and subscribers). Subscriptions received by subscribers are inserted in the subscription table, while events received by publishers are passed to the CBR engine. The Pub/Sub manager is also responsible of matching events received by the network (and passed to it by the CBR engine) with

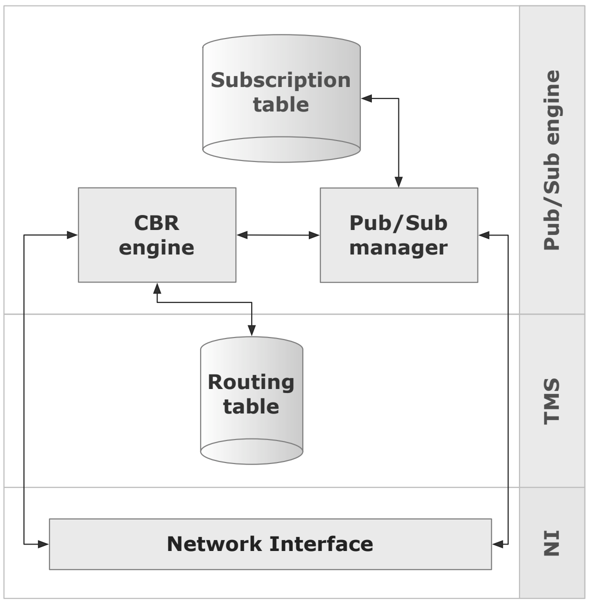

Figure 1. A generic broker architecture

local subscriptions contained in the subscription table, in order to notify the corresponding subscribers.

Routing table. This table stores a local view of both the the broker's network topology and the global subscription distribution. Specifically, the number of entries of the routing table at broker B_i is equal to the number of B_i's neighbor brokers. Each entry provides the zone of interest advertised by the corresponding neighbor node. For example let B_j be a neighbor of B_i. The zone of interest stored into the row associated with B_j in B_i's routing table represents the union of all zones of interests behind the link $l_{i,j}$.

CBR engine. This component implements the content-based routing algorithm (CBR) for acyclic peer-to-peer topologies introduced in SIENA [3]. It has two main functions that read and update the routing table: forwarding of events and updating of the zone of interests of each broker.

Each time an event e is received by the broker either from its local publishers (via the Pub/Sub manager) or from a link, this component forwards e *only* through those links which can lead to potential e's subscribers. In this case we say that B_i acts as a *forwarder* for e. Moreover we define *pure forwarder*, a forwarder broker which hosts no subscriptions matching the event e. Forwarding links are determined by the CBR engine from the routing table, by matching the event against the entries contained in it.

When a new subscription S arrives at the pub/sub manager broker of B_i, if S is not a subset of Z_i, it is first added to the Subscription table and then spreaded to B_i's neighbors via the CBR engine. When a CBR component of a broker B_j becomes aware of S through a message received from one of its incoming TCP links, namely l, it updates the row of its routing table by adding S to the l's entry in the routing table. Note that only the zone of interest can be changed by the routing protocol.

Network Interface. This component manages all the low level network-related issues. Different implementations can be used to let the broker work on different types of network.

While the first three components constitute the core of the pub/sub engine, the routing table can be considered the distributed representation of the brokers' network. In this sense we define it as the Topology Management System (TMS) of the broker. For a simple content-based pub/sub broker based on a static and acyclic network topology, like the one depicted in figure 1, the TMS is reduced to a single data structure, i.e. the routing table. Due to the absence of "active components" inside the TMS, this type of systems require human intervention in order to modify the network topology (adding or removing entries inside the routing table).

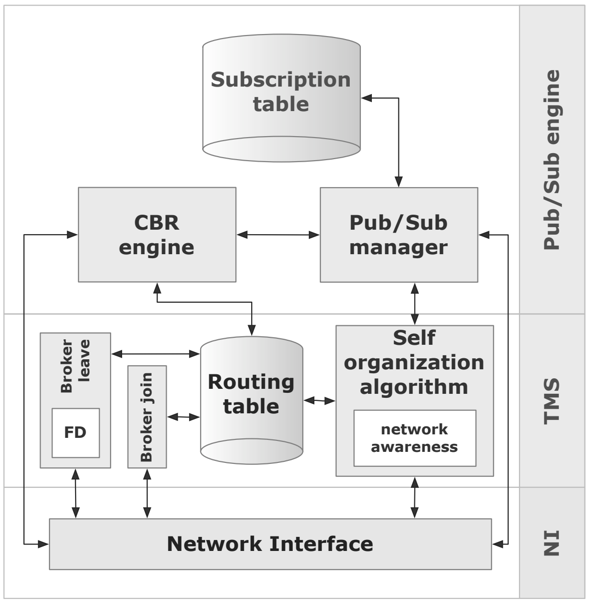

Figure 2. A crash-resilient self-organizing broker architecture

3. A TMS Architecture

This Section describes the internal architecture of a novel Topology Management System for Content-based Publish/Subscribe. Our TMS is capable of adding automatic network management, failure resiliency and self-organization to a plain CBR-routing algorithm. This is done by a set of components that update transparently the set of neighbors of a broker B_i by modifying the number of rows and/or the content of a row in the routing table.

The architecture of the TMS at broker B_i is depicted in figure 2; we added three main active elements to the simpler TMS shown in the previous figure (a detailed description of each of these components can be found in Section 4.1):

Broker join manager. This component is used by B_i to manage a new broker B_j trying to join the network. The algorithm used by this component (i) tries to obtain a fair distribution of the brokers over the whole network and (ii) updates B_i's routing table as soon as B_i accepts B_j as a new neighbor.

Broker leave manager. This component is used by B_i to manage brokers that want to leave the network and maintain the network healthiness through algorithms that handle broker crash failures. In order to reach this last goal, the component incorporates an eventually perfect failure detector based on heartbeats. The algorithm ensures that the topology remains acyclic without partitioning despite broker crash and/or leave. Each neighbor of the failed broker, say B_j, has to cooperate with the rest of B_j's neighbors to consistently modify its local routing table.

Self-organization engine. This component implements a self-organization algorithm capable of increasing CBR performance through topology reconfiguration of the brokers' network. Each reconfiguration tends to place close to each other, in terms of TCP hops, brokers whose zone of interests are very similar in order to reduce as much as possible the number of pure forwarder. However, a reduction of TCP hops does not always leads to better network level performance (for example establishing a TCP connections between two brokers with a narrow bandwidth can decrease the performance of the overall system even though these brokers share a large zone of interest). Therefore this component embeds a network-awareness module that allows topology reconfigurations only if network level performance are not severely harmed.

It is important to remark that TMS varies the number of entries in the routing table transparently wrt to the CBR engine; a cooperation between TMS and the CBR can however increase the performance of the pub/sub system.

4. A Detailed View of the TMS

4.1. Broker Join

A broker joining the system is added as a new neighbor to one already in the system. The join algorithm ensures a fair distribution of the brokers inside the network (i.e, new brokers are added with higher probability to brokers having a small number of neighbors). Moreover, it ensures that the number of connections handled by each broker is always bounded. When a new broker B_j wants to join the system, it starts contacting a broker randomly chosen among the ones already in the system. A broker receiving a request can choose to accept or forward it to its neighbors (one at a time, with the eventual exclusion of the broker it received the request from). A join request is accepted by a broker B_i with probability $p = \frac{F-|R_i|}{F}$, where $|R_i|$ is the number of rows in B_i's routing table. This ensures that the number of neighbors for a broker does not exceeds the maximum allowed. When a broker B_i accepts a join request, it adds a new entry for B_j to its routing table and sends to B_j the union of its zone of interest Z_i with all the zones covered by its links. This zone is added by B_j to its routing table, associated to the newly created link.

4.2. Broker Leave and Failures

A broker may leave the network voluntarily or if a failure occurs. We assume that brokers can fail by crashing, that they can fail only one at a time, and that the failure of a broker is eventually detected by all its neighbor brokers (using well-know failure detection techniques, like heartbeats). Moreover, a non-faulty broker can be erroneously detected as faulty by its neighbors only for a finite amount of time.

The leave-handling protocol must ensure that after a broker leaves, the global properties of the brokers' network are maintained, i.e. new connections have to be established, such that the network topology remains acyclic. Moreover, the state of the routing tables must be kept consistent removing the subscriptions that were hosted by the leaving broker and adding the newly created paths. We will present the protocol in the following by referring to Figure 3.

The leave-handling protocol requires an additional data structure at each broker. A broker B_i maintains a *neighbors list* NL_j for any of its neighbors B_j. This is an ordered list containing all the brokers that are neighbors for B_j. Figure 3(a) shows the neighbors list NL_3 of Broker B_3, contained in each of its neighbors. Neighbor lists are updated at each new brokers join or leave.

If a broker B_j leaves the network voluntarily, it sends a disconnection message to all its neighbors, while if it is faulty, it simply stops sending heartbeats. In both cases each broker in NL_j will eventually become aware of B_j leave and will request a new connection to the broker following itself in NL_j. Each broker $B_i \in NL_j$ receiving a connection request from the preceding broker in NL_j (with the exception of the first broker in NL_j) will accept the connection and test if B_j has actually left the network; if this is the case B_i will issue a connection request to the next broker in NL_j. When a connection request reaches the last broker in NL_j, a new path connecting all brokers previously connected by B_j has been created. Consider the Figure 3(b). If the broker B_3 leaves the network, eventually the connections shown as dashed lines will be set up. It may happen that a broker is erroneously suspected as faulty by some of its neighbors. A broker B_i suspecting a fault in one of its neighbors B_j, sends it a disconnect message before starting the protocol, forcing the link tear down.

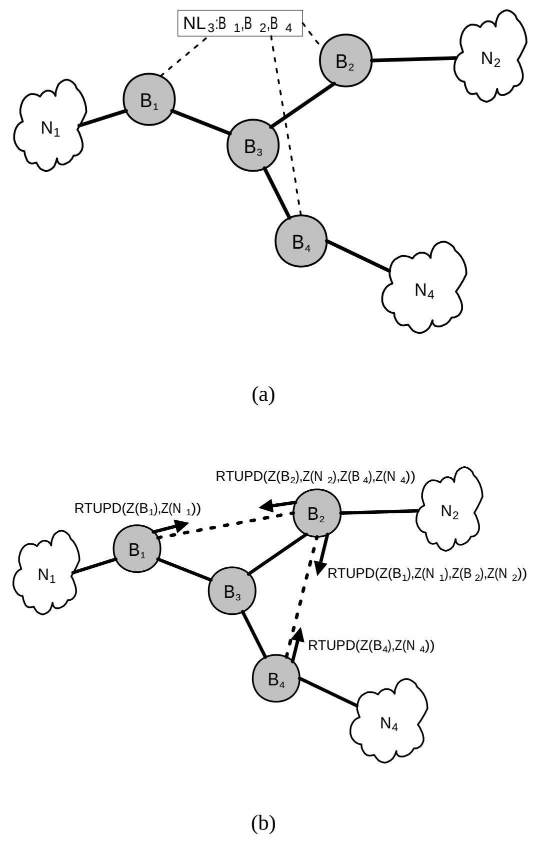

Figure 3. Managing Fault of a Broker

At this point, routing tables have to be repaired. This means (i) deleting all the subscriptions that previously belonged to B_j on all B_i's outgoing links, except the one connecting to B_i and (ii) updating the routing tables on all the newly created links.

As far as the latter point is concerned, two route update messages are generated. The first one is generated by the broker that started the reconstruction procedure, say node B', and the other by the one that ended the procedure, say node B''. B' sends its message toward B'' and vice versa. Each message is forwarded by all brokers in NL_j, before reaching the destination. Before forwarding the update message to the destination, each broker adds its local subscriptions and the zones of interest of its other links to it. Figure 3(b) shows the details of the route update messages (RTUPD) sent by each broker. The routing table update involves only the neighbors of the faulty brokers without impacting on other parts of the network. In order to avoid message losses all events and subscriptions in transit while repairing the topology are buffered until the repairing process is completed. It is important to point out that the fan-out constraints could be altered after a broker leave. A neighbor of a leaving node (with the exception of first and the last in the neighbors list) opens two new connections, indeed, while dropping only one, thus possibly reaching a number of open connections which is higher than its maximum fanout.

4.3. Subscription-Aware Self-Organization

Our self-organization component works in synergy with the content-based routing algorithm (i.e., it can read the CBR table and modify it after the self-organization) but its actions are totally transparent to it. In the following we describe the various aspects involved in self-organization.

Cost Metrics. The main target of the self-organization is to do its best to avoid the presence of pure forwarders during event diffusion due to the fact that a pure forwarder adds a useless TCP hop to the event diffusion path. This is done through a reorganization of the brokers' network with the aim of minimizing the TPC-hops experienced by an event to reach all its intended destinations. Lowering the number of TCP hops has a positive impact on the scalability of the system: each broker has to process a lower number of messages, and this leads to lower costs for events matching and forwarding. However, it is important to remark that reducing the TCP hops per notification diffusion through a topology rearrangement not necessarily leads to an equal improvement on network level metrics such as IP hops, latency and bandwidth. In other words, an efficient topology configuration wrt TCP hops could exhibit very poor performance wrt, for example, IP hops. Then, our self-organization component takes into account network-level metrics in order to decide if a new connection can lead to a non-efficient network path.

Measuring Subscription Similarity: the Associativity Metrics. Defining subscription similarity in a content based system can be very difficult due to the very generic language used for subscriptions. We introduce an associativity metric which measures the similarity between the interests of two brokers. In a former version of the self-organization algorithm, we exploited the geometric interpretation of subscriptions to compute associativity, starting from the geometrical intersection of the hyper-rectangles representing brokers' subscriptions. However, the method used to measure these intersections introduces a significant computational overhead and it is not straightforwardly extensible to operators such as prefixes and wildcards.

In this paper we introduce a novel solution for computing associativity, which is based on statistics over events received by each broker. A broker B_i maintains the history H_i of the last n received events. Note that, due to the features of the CBR algorithm, histories of distinct brokers usually are completely different. Any event in H_i that matches a subscription of B_i is included in another list $M(H_i)$. Associativity of a broker B_i with respect to another broker B_j, denoted as $AS(j)_i$ is computed as the percentage of events matched by B_i that are also matched by B_j:

$$AS(j)_i = |M(H_i) \cap M(H_j)|$$

We can also define the total associativity at a broker B ($AS(B)$), as the sum of the associativity among the broker and all its neighbors and, finally, the associativity of the whole system (AS) as the sum of associativity of all the brokers.

Self-Organization Algorithm. The algorithm follows this basic simple heuristic: each broker B tries to rearrange the network in order to obtain an increment of its associativity $AS(B)$ while not decreasing AS. In the following we give a quick sketch of the algorithm; a complete specification, along with implementation details and experimental results, can be found in [1].

A self-organization is triggered by a broker B when it detects (through the reading of the CBR tables) that there can be a broker B', not directly connected to B, that can increase its associativity. The aim of the self-organization is: (i) to connect B to B' and (ii) to tear down a link in the path between B and B' in order to keep the topology of the network acyclic (a requirement of the CBR algorithm). The algorithm does its best to select the link in the path between B and B' such that the two brokers it connects have the lowest associativity. Self-organization takes place only if it leads to an increase of AS and this, as confirmed by experiments, means increasing the probability that two brokers with common interests are placed close in the network. Network-level performance is accounted in the algorithm following this simple principle: self-organization can occur only with those brokers whose network-level distance with the source broker is less than a reference value d. In other words, a broker B starting the reconfiguration will choose as its new neighbor the most similar one having a network distance from it in the range delimited by d. Network distance can be measured indifferently with any metric, either IP hops or latency or bandwidth, depending on the specific requirements of the application.

5. Related Work

Several content-based pub/sub systems have been presented in the literature. Early content-based systems, such as SIENA [3] or Gryphon [2] did not provide any form of self-organization. The creation and the maintenance of the topology is left to the user. Peer-to-peer overlay network infrastructures, such as Pastry [9] and Tapestry [12] exploits self-reconfiguration capability for fault-tolerant routing and for adjusting routing paths with respect to metrics of the underlying network. Overlay network infrastructures represent a general connectivity framework for many classes of peer-to-peer applications, including publish/subscribe. For example, Scribe [8] and Bayeux [11] are two topic-based publish/subscribe systems built on top of two overlay network infrastructures (respectively Pastry and Tapestry). Only recently in ([7] and [10]) the same idea is being applied to two content-based systems.

None of the aforementioned systems include a subscription-driven self-organization mechanism. A reconfiguration algorithm similar to the one included in our TMS is described in [5]. Authors assume that some link may disappear and others appear elsewhere, because of changes in the underlying connectivity (this is the case, for example, of mobile ad-hoc networks). "Reconfiguration" in this case means fixing routing tables entries no longer valid after the topology change. In contrast, in our approach we *induce* a reconfiguration to create more favorable conditions for event routing.

6. Conclusions and Future Work

In this paper we presented a crash-resilient, self-organizing Topology Management System for content-based publish/subscribe. TMS masks topology changes to the content-based routing algorithm in order to keep an acyclic topology. TMS is self-organizing in three senses: (i) it handles autonomically brokers joining/leaving the system, (ii) it can self-heal the application-level network after a broker crash and (iii) it can dynamically reconfigure the connections among brokers in order to create paths that increase the performance of content-based routing.

While the implementation of the TMS is currently under development, we already did a simulation study of all the algorithms sketched in this paper. The simulations show how the self organization algorithm achieves significant improvements wrt the TCP hop metric compared to a simple CBR and without affecting the network latency of notifications (more details can be found in [1]). Moreover the topology maintenance algorithm allows the system to offer self-healing properties in presence of single broker crash.

References

[1] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. Subscription-driven self-organization in content-based publish/subscribe. Technical report, downloadable from http://www.dis.uniroma1.it/ midlab/docs/bbqv04techrep.pdf, DIS, Mar 2004.

[2] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In *Proceedings of International Conference on Distributed Computing Systems*, 1999.

[3] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Notification Service. *ACM Transactions on Computer Systems*, 3(19):332–383, Aug 2001.

[4] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI Event-Based Infrastructure and its Applications to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, September 1998.

[5] A. L. M. G. P. Picco, G. Cugola. Efficient content-based event dispatching in the presence of topological reconfiguration. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*, pages 234–243, 2003.

[6] P. Pietzuch and J. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, 2002.

[7] P. Pietzuch and J. Bacon. Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003.

[8] A. Rowston, A. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-Scale Notification Infrastructure. In *3rd International Workshop on Networked Group Communication (NGC2001)*, 2001.

[9] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[10] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A Peer-to-Peer Approach to Content-Based Publish/Subscribe. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003.

[11] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination . In *11th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2001.

[12] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Technical Report UCB/CSD-01-1141, University of California at Berkeley, Computer Science Division, April 2001.