INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1665

DPS: SELF-* DYNAMIC RELIABLE CONTENT-BASED PUBLISH/SUBSCRIBE SYSTEM

E. ANCEAUME A. K. DATTA[2] M. GRADINARIU
G. SIMON A. VIRGILLITO

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

DPS: Self-∗ Dynamic Reliable Content-based Publish/Subscribe System

E. Anceaume A. K. Datta[2]* M. Gradinariu
G. Simon** A. Virgillito***

Systèmes communicants
Projet Adept

Publication interne n ° 1665 — décembre 2004 — 28 pages

Abstract:

Publish/Subscribe systems provide a useful platform for delivering data (events) from publishers to subscribers in a decoupled fashion in distributed networks. These systems have many applications, including web services, stock quotes, free riding monitoring, and Internet games. Developing reliable publish/subscribe schemes for dynamic distributed systems is challenging due to the requirements for scalability for large groups of unpredictable subscribers and coping with arbitrary and frequent failures. In this report, we present a new content-based publish/subscribe system, called DPS (*Dynamic Publish/Subscribe*). DPS combines classical content-based filtering with self-∗ (self-organizing, self-configuring, and self-healing) subscription-driven clustering of subscribers. DPS gracefully adapts to failures and changes in the system while achieving scalable events delivery. DPS is a scalable, totally distributed, fully localized, and self-∗ publish/subscribe implementation. We present numerous protocols to show the versatility of DPS in deploying in wide range of applications. We demonstrate the reliability and scalability of our system through both simulations and analytical studies.

Key-words: dynamic network, content based publish/subscribe, reliability, self*

(Résumé : tsvp)

* School of Computer Science, University of Nevada Las Vegas, USA, datta@cs.unlv.edu. Phone: 702-895-0870. Fax: 702-895-2639.
** France Telecom R&D, Issy les Moulineaux, France
*** Antonino Virgillito is partially supported by project MAIS, funded by Italian MIUR

Centre National de la Recherche Scientifique (UMR 6074) Université de Rennes 1 – Insa de Rennes
Institut National de Recherche en Informatique et en Automatique – unité de recherche de Rennes

DPS: Un système d'abonné/editeur auto-$*$ fiable

Résumé : Les systèmes "abonné/editeur" emergent en tant que nouveau paradigme de communication. Ces systèmes ont de nombreuses applications telles que les services web, les marchés boursiers et les jeux utilisant l'Internet. Développer les schémas abonné/editeur fiables dans des systèmes distribués dynamiques se révèle être un défit étant donné les contraintes spécifiques à ces systèmes (ex. passage à l'échelle, évolutivité). Dans ce rapport nous présentons un nouveau système d'abonné/editeur — DPS (*Dynamic Publish/Subscribe*)—. Notre système construit des groupes sémantiques basés sur le filtrage d'événements tout en garantissant des propriétés auto* (auto-configuration, auto-organisation et auto-réparation). Nous prouvons analytiquement et par simmulation que DPS passe à l'échelle DPS dans des environnements dynamiques en utilisant uniquement des calculs locaux.

Mots clés : réseaux dynamiques, abonné/editeur semantique, auto*

1 Introduction

The publish/subscribe paradigm has emerged in the recent years as an effective technique for building distributed applications in which information has to be disseminated from *publishers* (event producers) to *subscribers* (event consumers).

The decoupled nature of publish/subscribe interaction makes these architectures particularly suitable for building large-scale applications where scalability plays a key role. In publish/subscribe systems, users express their interests in receiving certain types of events by submitting a predicate defined on the event contents. The predicate is called the user *subscription*. When a new event is generated and *published* to the system, the publish/subscribe infrastructure is responsible for checking the event against all current subscriptions and delivering it efficiently and reliably to all users whose subscriptions match the event. Publish/Subscribe systems can be classified into two broad types based on the subscription expressiveness. Topic-based systems assume a predefined set of subscribers. Content-based systems allow to precisely express interests through a complex query language.

Combining expressiveness of subscription language and scalability of the infrastructure poses an interesting challenge that has inspired many researchers of late to explore this topic further. Despite these promising features, actual deployment of such architectures in real, large-scale systems is currently limited by their lack of self-∗ capabilities.

Contributions of the Paper In this paper, we present a content-based publish/subscribe architecture with self-∗ properties, called Dynamic Publish Publish/Subscribe System (DPS) in which nodes coordinate without any human intervention, apart from bootstrapping. Informally, the content-based publish/subscribe problem can be stated as follows: A node p is eventually notified of a published event e if p previously issued a subscription that matches e. Nodes *self-organize* into groups according to *similarity* relationships among their subscriptions. In particular, all subscribers interested in the same attributes are logically connected to one another by clustering themselves into the same group. Then, if some subscriber storing a given subscription is located, other subscribers storing similar subscriptions are quickly located as well. The meaning of *similarity* of subscriptions is not obvious when using a content-based language, where subscriptions cannot be clustered in a straightforward way. We provide formal definitions of similarity, and use them to group nodes.

Groups of subscribers *self-configure* to form tree structures such that one tree is built per attribute. When an event is propagated along the tree, it is transferred only to groups having at least one subscriber interested in the event. The resulting logical structure is such that nodes subscribing to similar sets of events are placed in close proximity from one another, i.e., forming a cluster of a relatively small radius. The goal is to reduce the communication overhead of reaching all the subscribers for a given event by avoiding as much as possible the nodes not interested in that event. We use both masking and non-masking approaches to achieve the stability of DPS. The masking is implemented by increasing the connectivity (using up to $K_s + 1$ additional links, where K_s is a parameter related with the length of the tree) between a group and its neighboring groups (i.e., its successor and predecessor groups). This tolerates the failure/unsubscription of up to K_s neighboring groups without partitioning the logical tree. Moreover, the failures/unsubscriptions within a group are either masked or detected and corrected according to the communication policies we use (epidemic-based or leader-based). The non-masking tolerance is implemented by *self-healing* DPS in the following manner: If the number of failures/unsubscriptions exceeds the expected number

(i.e., more than K_s), DPS can adjust itself in a completely autonomous and local manner. Every root of a subtree that became partitioned from the rest of the tree proceeds like a new subscription.

Our system is *scalable* in four respects: 1) *Local knowledge* enables each node to keep track of a limited number of its neighbors regardless of the size of the system. 2) Neither *broadcasting* nor *manual intervention* is used when new subscriptions enter the system. 3) *Local fault-tolerant mechanisms* ensure that the effect of node failures is confined within a bounded number of neighboring groups (up to K_s groups). 4) *Local self-healing* guarantees that even in case of an unpredictable number of unsubscriptions or failures within different logical trees, the disjoint trees eventually form groups again independent of each other making no impact on other trees. We propose and analyze different methods to traverse the logical trees of the system. Different characteristics of the proposed algorithms allow DPS to cater to needs of different deployment contexts. First, we study the *root-based* approach — the publishing and subscribing processes always start from the root of each attribute tree. This approach provides almost optimal publication/subscription turnaround time . (The publication turnaround time is the elapsed time between an event submission and its notification. The subscription turnaround time is the time required for a subscription to find its similarity group.). However, this induces a very high stress on the root nodes that may fail prematurely. In order to overcome this problem, we propose a generic approach where publications and subscriptions may start from any contact point in the logical trees. The generic approach balances the node and link stress across the logical trees when publication/subscription frequencies are very high.

For both approaches above, we propose and analyze two different communication schemes: leader-based and epidemic. The leader-based approach relies on privileged nodes to handle all the communications within the groups. This reduces the number of messages and the amount of information that needs to be exchanged among the group members. The epidemic approach is based on gossiping of events. This method overcomes message losses and gives probabilistic guarantees of delivery when messages can be lost and nodes can crash frequently. Both solutions are targeted towards scalability, and minimize both the volume of information and number of messages exchanged. A formal analysis and a simulation study confirms the self-$\star$ properties of DPS, showing its scalability and dependability.

The rest of the paper is organized as follows. In Section 2, we present the system model and the problem statement. In Section 3, we first define the logical backbone of DSP. Then we formally introduce a new concept, called *similarity relation*, the logical structure of the DPS is based on. Section 3 also includes the design principles of DSP. Several algorithms for traversing the logical structures for subscriptions and publications, and for communicating within the groups are presented in Section 4. Due to the lack of space, the pseudo-code of the algorithm, the proof of correctness of DSP, proof of its self-$*$ properties, and the complexity analysis are included in the Appendix, that was provided as complement for the reviewing process. An analytical study is provided in Section 5. Detail experimental results are provided in Section 6. We discuss related work in Section 7. We make some closing statements on the proposed and future work in Section 8.

2 Framework

2.1 The Publish/Subscribe System

We assume a large finite, yet unbounded dynamic set of *nodes*, also referred to as *processes* or *processors*. The set is dynamic in the sense that nodes can join or leave at an arbitrary time. Each

node is associated with a unique identifier. Nodes can communicate with each other over a *network* providing a best-effort datagram service, similar to the Internet. In other words, most messages are delivered unless either the sender or receiver fails beforehand. Yet, neither the message delivery nor ordered delivery of messages is guaranteed. The system communication can be modeled by a weakly connected graph such that vertices represent the nodes of the system and edges represent established communication links between processes. We assume that the topology and the size of the communication graph are subject to frequent and unpredictable changes: processes can leave or join the system arbitrarily often, and they can fail temporarily (transient faults) or permanently (crash failures). Communication links can commit transient failures (message loss).

In a publish/subscribe system, nodes cooperate to send (publish), relay, and receive (notify) special messages, namely *events* (or *publications*). The interest of a node in a set of events is referred to as *subscription*, and is expressed as a predicate defined on the content of the events. In order to inform the system about their subscriptions, nodes invoke a `Subscribe` primitive. The invoking node is called a *subscriber*. Additionally, nodes can *publish* events by invoking a `Publish` primitive. We say that a node is *notified* of an event when the system invokes a `Notify` primitive on it. A publish/subscribe system distributes events to all interested parties anonymously, asynchronously, and in a loosely coupled manner. The recipients of the events (i.e., the interested parties) are determined by matching of their subscriptions with the content of the event. Informally, a publish/subscribe system is in charge of: 1) storing all subscriptions associated with the respective subscribers. 2) Receiving all relevant events from publishers. 3) Dispatching all published events to the correct subscribers.

The design of publish/subscribe systems can be classified in two main categories: topic-based and content-based. In topic-based systems, the events are assigned to a predefined set of topics (attributes). The subscription of a topic-based subscriber is a set of topics. The subscriber is notified of all the events that are associated with its subscription. Content-based publish/subscribe systems enable restrictions (filters) on the event content. That is, a content-based subscription specifies equality or range predicates over one or more attributes, and only those events whose contents satisfy all the predicates are notified to the subscriber. Content-based publish/subscribe system is significantly more complex than topic-based since the recipients change for each event and they cannot be easily grouped in advance. Our system follows the content-based approach, as detailed below.

2.2 Specification of the Content-based Publish/Subscribe Problem

In our model, when a node issues a `Subscribe` method, it may take some time for the publish/subscribe system to be aware of this operation, especially because of the delay encountered by all the entities composing the system until they receive the subscription request. After this delay period, the subscription is considered to be *stable*. We denote by T_{diff} the time it takes for the system to compute the set of interested subscribers of an event (by checking the possible matches as discussed earlier) and notify them. Due to asynchrony, this time may only be known by "external observers" of the system, but not by the nodes of the system. A system is considered to be a content-based publish/subscribe system if the following properties are satisfied [1, 3]:

Legality If some node p_i is notified of some event e, then p_i previously subscribed a subscription with a filter f such that e matches f.

Validity If some node p_i is notified with e, then there exists some node that previously published e.

Fairness Every node may publish infinitely often.

Event Liveness Node p_i is eventually notified with e if p_i subscribed a filter f such that e matches f and f was stable within T_{diff} time units after e has been published.

2.3 Publish/Subscribe Data Model

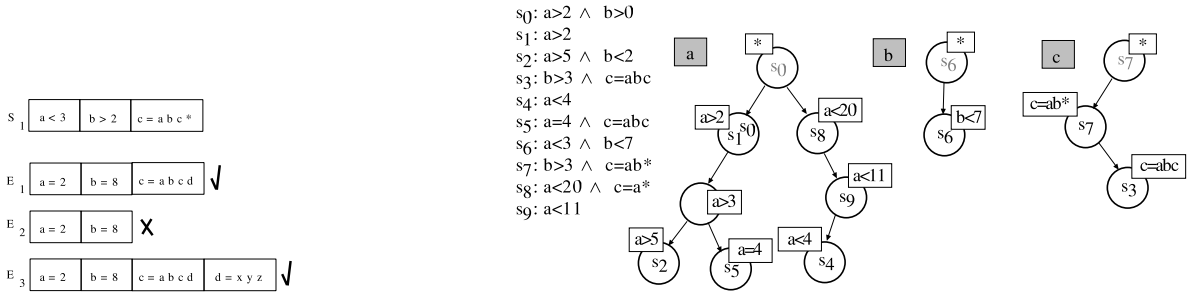
We consider a content-based publish/subscribe data model [10] where both subscriptions and events use as building blocks a finite, yet unbounded universe of attributes. Each attribute is defined by a unique name, a type (e.g.,. integer, string, or enumerated), and a domain (the range of the attribute values). A content-based *subscription* (filter) is a conjunction of predicates, i.e., $F = AF_1 \wedge \ldots \wedge AF_j$, where AF_i defined as a tuple $AF_i = (name_i\ Op_i\ c_i)$ where $name_i$ is the name of the attribute, Op_i an operator, and c_i is a constant value. The operator Op_i can be chosen from a set of basic operators that depends on the attribute type. For example, possible operators for numerical attributes are $\{=, <, >\}$. Complex filters can be expressed as the conjunction of two or more basic operators. For example, a range filter for an attribute a of the form $c_1 < a < c_2$ can be obtained as the conjunction of the two predicates $AF_1 = (a > c_1)$ and $AF_2 = (a < c_2)$. Thus, our subscription language can support a broad variety of content-based constraints, such as equality, comparisons, range, set containment for numerical types, and prefix and suffix wild cards for string types.

An *event* is a conjunction of equalities over the attributes' universe. More precisely, an event is denoted as $E = AV_1 \wedge \ldots \wedge AV_k$, where $AV_i = (name_i = v_i)$, where v_i is the value of the attribute. An event predicate AV matches a subscription predicate AF (denoted as $AV \in AF$) if the attribute names are the same in AV and AF and the attribute value in AV is in the range defined by AF. An event matches a subscription iff for all the predicates in the subscription, a corresponding matching value appears in the event. Consider Figure 2.1.a where S_1 is a subscription, and E_1, E_2, and E_3 are three events. Event E_1 obviously matches subscription S_1. Event E_2 does not match with S_1: Though the values of attributes a and b match the corresponding constraints in S_1, there is no value of attribute c in E_2. Hence the third constraint of the subscription S_1 is considered to be not satisfied. On the contrary, event E_3 matches S_1 since the common attributes are matched, and the fact that attribute d is not present in S is considered as a wild card constraint.

It is important to note that the number of attributes for events and subscriptions is not fixed. That is, each single subscription or event can include an arbitrary number of predicates and no prior agreement among participants is necessary.

3 Semantic Logical Layer

DPS uses as underlying logical layer a virtual forest of tree-like connected semantic clusters. A cluster groups subscribers that share similar interests with respect to their subscription filters. Two clusters are in the same logical tree if all contained subscriptions share a common attribute. Each logical tree is associated with a single attribute. A node may host several subscriptions. So, it may belong to multiple clusters in different logical trees. Moreover, as each subscription may be composed of more than one predicates, a node may belong to a different cluster for each of the attributes of each



(a) Matching Example

(b) Logical Trees

Figure 2.1:

of its subscriptions. In order to reduce the number of clusters a node belongs to (and subsequently, reduce the information to be stored at a node and simplify the cluster joining process), a node is restricted to belong to only one cluster per subscription and thus to only one tree per subscription. It is sufficient for each node to join only one cluster in only one logical tree corresponding to one attribute of its subscription since each event is published in all the logical trees that matches the attributes of the event. This guarantees that an event will be propagated in all the trees for possible subscribers, but maintaining every subscriber only in one tree avoids an event to be received more than once by the same node.

Each node in a tree maintains only a view of the subscriptions of its direct neighbors (successors and predecessors). This localizing property gives our scheme a real edge over other solutions based on a network of brokers (as SIENA) in which the global knowledge of the subscriptions in the system is required. Furthermore, the insertion of a new subscription only affects one node in a tree, and possibly the link with the successors and predecessor of this node; no broadcast is needed. At the same time, the structure supports efficient event dispatching because events can travel down the trees without being forwarded through groups containing non-interested subscribers. In the following, we define a set of relations characterizing "similarity" of interests. A formal definition of these relations can be found in the Appendix. Based on these relations, we further describe the design of the semantic logical structure on which DPS relies.

3.1 Similarity Relationship Definitions

Two nodes are *similar* when they are related through *sibling* relations (defined below). The sibling relation clusters nodes into semantic groups. Two nodes are *siblings* when they share at least one common predicate in at least one of their subscriptions.

Definition 3.1 (Sibling Relation $\bowtie$) *Let p and s be two nodes representing two subscription filters F_p and F_s, respectively. Assume that $F_p = \wedge_{i \in I} AF_i^{F_p}$ and $F_s = \wedge_{j \in J} AF_j^{F_s}$, where I and J are the sets of indices for predicates in F_p and F_s, respectively, and $AF_m^{F_q}$ is the predicate AF_m in the subscription filter F_q. p and s are* siblings *with respect to a predicate AF, denoted as $p \bowtie_{AF} s$* iff $\exists k \in I, k' \in J :: \; AF_k^{F_1} = AF_{k'}^{F_2} (= AF)$.

A semantic cluster (group) is identified through a *group predicate* which is the common predicate on which members of the groups are siblings. For example, the group labeled $A > 3$ refers to the

group of all the subscribers that include $A > 3$ in their subscription predicate. Based on the sibling relation defined above, we define a *labeled group* (semantic cluster) as the set of nodes which are siblings of each other.

Definition 3.2 (Labeled Group) *Let G be a set of nodes and AF a predicate. G is a AF labeled group iff $\forall p, s \in G :: p \bowtie_{AF} s$.*

The *group predecessor* relation imposes a hierarchical ordering among the clusters that is based on the *predicate inclusion* relation. A predicate AF_2 is included in a predicate AF_1 if all the events matching AF_2 also match AF_1.

Definition 3.3 (Predicate Inclusion) *Let AF_1 and AF_2 be two predicates and AV an event. $AF_2 \subset AF_1$*iff *$\forall AV \in AF_2$, $AV \in AF_1$.*

Two semantic clusters (labeled groups) are related through the *group predecessor* relation when their respective labels are related by the above defined predicate inclusion relation.

Definition 3.4 (Group Predecessor Relation $\stackrel{pred}{\rightarrow}$) *Let G_1 and G_2 be two labeled groups with respect to the predicates (or labels) AF_1 and AF_2, respectively. Then $G_1 \stackrel{pred}{\rightarrow} G_2$* iff*$AF_2 \subset AF_1$ and $\nexists G_3$ such that $G_1 \stackrel{pred}{\rightarrow} G_3 \stackrel{pred}{\rightarrow} G_2$.*

Other approaches [14, 15, 6] group similar subscribers by applying a partitioning criteria over the event space. All nodes having subscriptions that fall into a common partition are grouped. We chose the similarity relation over the partitioning method for the following practical reasons: It is simpler to implement, does not require prior agreement among the nodes, does not depend on the number of nodes, and reduces the number of non-matching messages received by nodes in a group. Moreover, as pointed out in [6], the choice of an effective partitioning criterion strictly depends on the distribution of subscriptions and events, the knowledge of which is not required in our approach. However, if partitioning is found more suitable than the similarity relation in some applications, the partitioning can be integrated into our system without significant changes in the algorithms.

3.2 Design Principles of the Semantic Logical Layer

The logical structure on which DPS relies is subscription-driven. Subscribers locally self-organize in semantically connected clusters using the sibling relations (as defined in Section 3.1 and Appendix). A cluster is connected iff there is a path between any two cluster nodes in the underlying communication graph. There is a logical link between two clusters iff they are related through the predecessor relation and there is at least one path in the underlying communication graph that links two nodes, one in each cluster. The clusters self-organize into a tree hierarchy following the group predecessor relations (as defined in Section 3.1 and Appendix). Section 4 further details the implementation of the self-organization algorithms. The logical structure obtained after self-organization is a forest of logical trees, where each tree is associated with an attribute and only one tree is maintained per attribute. Each node of the tree represents a semantic cluster (or group) and it is labeled with a predicate (filter on the tree attribute). The root of the tree is labeled by a special wild card predicate matching all the possible values of the attribute. Groups in the trees are related among each other through the group predecessor relation. In the following, we assume that each attribute is "owned" by an unique node. Figure 2.1.b depicts a set of subscriptions and a possible set of corresponding logical trees.

The owners of the trees labeled "a", "b", and "c" are subscribers s_0, s_6, and s_7, respectively. We have imposed only two constraints on the construction of logical trees — the sibling and predecessor relations. However, this may lead to more than one possible trees. In particular, equality predicates can be predecessors of several different predicates at the same time. The corresponding groups may then appear in different positions within a tree. For example in Figure 2.1.b, in the tree for attribute "a", the group for predicate $a = 4$ is placed below the group for predicate $a > 3$. Such group could also have been considered as a successor of the group $a < 11$ or even the root group $a = *$. Although all the above possibilities are allowed by the sibling and predecessor relations, allowing multiple ways of constructing trees may cause inefficient group locating strategies. To overcome this problem, we impose two additional constraints on tree construction:
Constraint C1: All groups corresponding to equality predicates must be placed following a single consistent convention — as successors of either the greater-than or the less-than clusters.
Constraint C2: All groups corresponding to equality predicates must be placed as successors of the smallest possible cluster. If no such cluster exists, an empty (dummy) group will be created to maintain the consistency.

For example, the group for predicate $a = 4$ is placed as a successor of the group $a > 3$. Note that the latter is a dummy and empty group created only to keep the constraint consistent and presentation clear. Dummy groups do not have any practical consequences.

In order to improve the complexity of both publish and subscribe algorithms in the next section, we introduce a special link, called *virtual bridge links* between branches of the same tree. Two nodes are candidates for the two end points of a virtual bridge link iff they are two clusters of two different branches of the same tree.

Note that the proposed semantic logical layer is completely independent of the underlying communication layer. It can be constructed on top of any structured DHT-based networks (e.g., CAN, Chord, Pastry, JXTA v.2, etc.) or on top of any unstructured networks.

4 The DPS System

In this section, we describe how to build the DPS publish/subscribe communication system based on the notion of logical tree introduced in Section 3.2. Our design goal is to achieve the following three self-∗ properties that are critical to any dynamic and fault-tolerant system: *(1) Self-organization:* Nodes in the system organize themselves into groups according to the similarity relationships. *(2) Self-configuration:* Groups organize themselves into logical trees. *(3) Self-healing:* Nodes preserve the relationships even in presence of dynamic behavior, such as joins, departures, or failures.

In the next two subsections, we will describe how the logical trees are contacted and built, and techniques to propagate a subscription or a publication across the groups belonging to forest of logical trees. In Section 4.1, we focus on the construction and maintenance of logical trees and how they are exploited for diffusing publication/subscription events. We propose two approaches to traverse a logical tree. In the *root-based approach*, a node issuing a subscription or publication contacts the root of the tree. Although this approach is a natural way to start the publication/subscription process, it may overload the root node. This problem is avoided in the *generic approach* where a node randomly chooses a node in the logical tree and contacts that node first. This node is called the *contact point*. In Section 4.2, we propose two different policies for diffusing messages, both subscriptions and publications, inside a group as well as across the groups. The *leader-based* solution relies on a predefined node (the leader) to disseminate the information. In the *epidemic* approach, all nodes are involved in the communication. Note that the approaches for tree construction and

traversal are orthogonal to that for the communication. So, they can be combined to design four possible implementations of DPS.

4.1 Tree Construction and Traversal

As previously mentioned (in Section 3), a node belongs to only one group (and thus, to only one tree) per subscription. The tree is one that matches one of the attributes of the subscription. The attribute can be arbitrarily chosen without affecting the correctness of the solution. However, a more efficient solution would be obtained if the attribute is chosen according to a well-defined criteria. For example, it can be the one corresponding to the most selective predicate inside the subscription. Intuitively, the more "selective" a predicate is, the fewer the number of publications that match the predicate (assuming uniform distribution of publications). On the other hand, if a subscriber is included in the group associated with a non-selective predicate, it will receive many events most of which will be filtered out due to the most selective predicate. By choosing the most-selective predicate, such events will not be sent to the group avoiding unnecessary load on the node and saving network resources. In choosing a tree to join, the network distance among nodes may be considered. The criterion could be to obtain the lowest network distance or to keep the network distance within a bound. Adding network-awareness in the choice of the tree to join could overcome possible inefficiencies if the links were created following only the semantic constraints. In general, a mixture of the above listed criteria, as well as others not listed here can be applied based on the application properties and requirements.

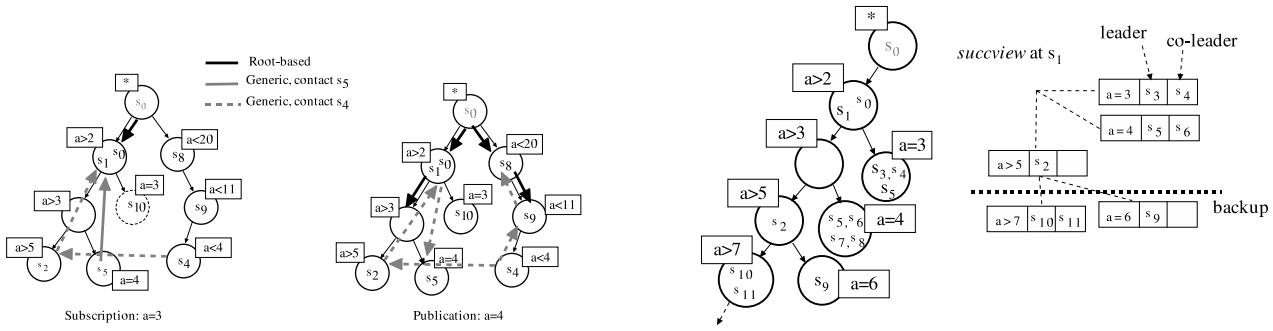
Prior to subscribing or publishing an event, a node enters the system by contacting one or more trees. When a node wants to issue a subscription in the system, it needs to locate and contact the contact point of the chosen logical tree. When a publisher wishes to publish an event, it contacts the contact points of all the logical trees corresponding to the attributes contained in its event. This guarantees that an event will be propagated in all the trees having possible subscribers. Several solutions exist to know the contact point address (if not already known). A widespread one is for the subscriber or publisher to ask any node it knows in the system to propagate a request message containing the attribute(s) it is interested in until locating the contact point(s) [11, 16]. Another one is to use a DHT overlay containing for each known attribute the address of a node sharing the attribute. Note that contact points are also contacted when the perturbations (failures/departures) in DPS cause partitioning of trees. Additional links maintained by each group may prevent the partitioning of a tree, but they may not be enough. In this situations, our system self-heals as follows: The root of each disconnected subtree searches for a contact point in that tree and proceeds with the insertion of new subscriptions.

Figure 4.2.a shows the paths followed by a subscription ($a = 3$) and a publication ($a = 4$) in the tree considered in Figure 2.1.b for both approaches.

Each node maintains a variable, *group* that contains all relevant information associated with the group the node belongs to. The information includes the group predicate, the operator of the predicate, the leader of the group if it exists, the view of the group, the addresses of the successor and predecessor groups if any, the addresses of the bridge connections if they exist.

4.1.1 Schema of the Subscription Process

Construction of the logical tree is done based on subscriptions. Once a subscriber has located the contact point in the logical tree it chose, it needs to locate the group of its sibling. If no such group



(a) Tree Construction and Traversal

(b) Example of Succview Data Structure

Figure 4.2:

exists, the subscriber creates a new one. Note that any activity within the group (such as event and subscription propagation) is blocked while the successor group is changing until the data structures are not updated in all the involved nodes. Allowing publication and subscription events during group construction may result in events not delivered to the new successor group, or more seriously, creating incorrect groups in the trees in case of concurrent group creations. The process of finding a group within a logical tree slightly differs between the root-based and generic approaches (described below). The pseudo-code for tree traversal followed by the subscription process is presented in Figure 4.3.

Generic Approach. 1) *The subscriber is on the right branch.* The subscriber will be eventually inserted in one of the groups of this branch. If an appropriate group for the subscriber is found, a subscription message is sent back to the new subscriber to join the group. Otherwise, it is redirected to either the predecessor or the successor group. The target of the redirection is decided according to the predecessor and successor group relationships. If at some point during the traversal (up or down), no group matches the subscriber's predicate, then a new group is created. Each time a new successor group is created, event propagation is blocked by setting the proper variable. The variable is reset when data structures related to the new successor are updated.
2) *The subscriber is not on the right branch.* The subscriber is redirected to one of the bridge connections, if any. Otherwise, it is redirected to the predecessor groups until either a bridge connection is found or the owner of the tree is reached. If the subscriber operator is not present in any of the tree branches, then a new group is created containing only the subscriber.

Root-based Approach. The contact point redirects the subscriber to the tree branch that matches the subscriber operator, if it exists. Otherwise, it creates a new group (hence, a new branch) containing only the subscriber. If a branch exists, the generic-based approach is applied to find the right group.

Note that for efficiency reasons, both C1 and C2 constraints must be applied for the predicates defined with the equality operation during the tree construction (see Section 3.2). Consider the example on the left side of Figure 4.2.a The black line shows the path followed by the subscription $a = 3$ issued by subscriber s_{10} using the root-based approach. The contact point first redirects it to the (left) branch containing equality predicates. The subscription is received by group $a > 2$.

```
upon receive FIND_GROUP(predicate,node) at group x
initialization:
  FindOperator(op,group) → returns the group whose operator matches op;
  Root(x) → returns true if x is the root of the logical tree;
  begin
  1rst phase: finding the right branch
  if (generic-based approach) ∧ (x.Op ≠ predicate.Op) then
    if (x.Bridge) then
      send(FIND_GROUP(predicate,node)) to x.Bridge;
    else
      if (x.Pred) ≠ ∅ then
        send(FIND_GROUP(predicate,node)) to x.Pred;
      end if
    end if
  if Root(x) then
    if ((succ=FindOperator(predicate.op,x) ≠ ∅) then
      send(FIND_GROUP(predicate,node)) to succ;
    else
      x.blockPublish ← true;
      send(CREATE_GROUP(predicate,x)) to node;
    end if
  2nd phase: finding the right group
  if (predicate = x.Predicate) then
    send(SUBSCRIBE_TO(predicate,x)) to node;
  else
    if (generic-based approach) ∧ (x.Pred.Predicate ⊂ predicate) then
      send(FIND_GROUP(predicate,node) to x.Pred;
    if (x.Succ.Predicate ⊂ predicate) then
      send(FIND_GROUP(predicate,node) to x.Succ;
    else
      x.blockPublish ← true;
      send(CREATE_GROUP(predicate,x)) to node;
    end if
  end
```

Figure 4.3: Tree Construction: Subscription Process

Since this group is the smallest possible predecessor of group $a = 3$, it is considered the designated predecessor for the subscription (Constraint C2). As this group does not exist, it is created below $a > 2$ and s_{10} is added to it. The paths followed by the generic approach are represented by grey lines — solid and broken lines for the contact points s_5 and s_4, respectively. In the latter case, the bridge connection is exploited for traversing from one branch to another.

Removal of subscriptions (unsubscribe) is handled similar to the subscription approaches. It is required to update the data structures of the successor (or predecessor) group and remove the group if the unsubscribing node is the last one.

4.1.2 Schema of the Publication Process

Event dispatching involves a visit of all the logical trees (matching the attributes of the event) built by the subscription propagation process described in Section 4.1.1. In other words, propagation of an event along the tree exploits the similarity relationships in order to visit all (and possibly only) the groups hosting potential subscribers. The pseudo-code is presented in Figure 4.4.

```
upon receive PUBLISH(event,group) at group x
  begin
    if (event matches x.Predicate) then
      PUBLISH_GROUP(event) in x;
      send(PUBLISH(event,x)) to all the successor groups in x.Succ;
    end if
    wait(x.blockPublish)
    if (generic-based approach) then
      if (group ∈ x.Succ) ∧ (x.Pred) then
        send(PUBLISH(event,x)) to x.Pred;
      else
        if (group ∈ x.Pred) ∧ (event matches x.Predicate) then
          send(PUBLISH(event,x)) to x.Succ;
        end if
      end if
      if (x.Bridge) then
        send(PUBLISH(event,x)) to x.Bridge;
      end if
    end if
  end
```

Figure 4.4: Tree Traversal: Publication Process

Root-based Approach. The visit of the trees starts from the root of each logical tree matching the attributes of the published event. An event received by a group is matched against the group predicate. If the event matches the group predicate, it is propagated inside the group. Moreover, it is forwarded to the successor groups provided a group is not being created. Otherwise, propagation is blocked until all data structures are updated. Downstream propagation along the tree continues as long as the event matches the group predicate; it stops otherwise. The successor relation between groups ensures that no matching subscribers are present in any successor groups. So, the entire branch of the tree can be safely excluded from the event propagation. Consider the right side of Figure 4.2.a. In the root-based approach (the black line), the publication $a = 4$ is forwarded downstream from the contact point to all the groups with predicates matching the publication.

Generic Approach. Similar to the corresponding subscription scheme, a node publishing an event e can choose any node in the tree as a contact point instead of choosing the root of the attribute. Similar to the root-based approach, matching events are propagated inside the group and to the successor groups. On the other hand, if the event does not match the group predicate, it still has to be forwarded upstream to the predecessor as follows: If the event has been received by j from its successor, it is forwarded to j's predecessor. If the node is received by the owner of an attribute, it stops forwarding. Otherwise, if the event has been received by j from its predecessor, it is forwarded to j's successor only if it matches j group predicate. Finally, if j holds a bridge reference, it forwards the event along the bridge.

When the publication starts from the group $a < 4$, it is propagated up through both branches (dashed grey lines) through all the matching groups. Note that group $a > 2$ also needs to forward the publication to its successor in order to reach group $a = 4$.

4.2 Communication within the DPS system

4.2.1 Leader-Based Communication

Each group in each logical tree contains a special node which behaves as the *leader* of the group. K_c additional leaders are maintained to deal with the leader failure. A node becomes the leader of a group as soon as it creates its own group or becomes the only member of a group. A node joins an existing group either as a co-leader or a regular member (i.e., neither a leader nor a co-leader). At most K_c co-leaders are maintained in a group. Communication between different groups is realized via their respective leaders. Note that the group leader must not be confused with the attribute owner — they are different nodes.

In order to efficiently handle multiple partitions (either due to volunteer departures or failures), the leader and its co-leaders (if any) maintain $K_s + 1$ additional links. These links, called *back-up links*, enable the leader to tolerate removal of its successor and/or predecessor groups (due to multiple departures within these groups). Note that K_s is proportional to the size of the tree. These links are maintained in the *succview* data structure. This structure is a tree-like structure. It maintains up to K_s back-up links to successor and predecessor groups the leader is aware of. That is, the leader and its co-leaders maintain for each of these predecessor and successor groups, say G, a link on both G'leader and G's co-leaders (if any). Furthermore, as some of these groups G may be dummy groups (do not contain any nodes), at least K_r back-up links (with $K_r \leq K_s$) must be toward leader and co-leaders not belonging to equality groups. This is required for maintaining connectivity of the tree in case the direct successor/predecessor groups of the leader are dummy or contain only a few processes. In summary, at least one node maintains the *succview* data structure per group (if the leader is the only node in the group) and at most $1 + K_c$ ones (if the leader has K_c co-leaders). The maximal size of this tree-like structure is equal to K_s (which is the case when the leader is the only node in the group). Figure 4.2.b depicts an example tree and the corresponding *succview* at a node (s_1). Note that if an equality predicate does not have a corresponding smallest predecessor group in *succview*, this is created as a dummy group. Finally, in addition to variable *group* (see Section 4.1), each node in a group maintains a list, called *sublist* which contains all the subscriptions issued by the node within the group.

Leader-based Subscription. Subscription process is realized by implementing two primitives, namely CREATE_GROUP and SUBSCRIBE_TO (Figure 4.3). Both are invoked on the new subscriber by

the leader of the predecessor group. Upon receipt of such primitives, the new subscriber updates its variable *groups*. If it becomes the leader of a new group, it updates *succview*. This is described in Figure 4.5.

Leader-based Publishing. An event received by a group through a node is always redirected to the group leader. The leader propagates all the events it receives to all the group members. Each member upon receipt of an event notifies its application only if the event matches one of its subscriptions. For a given group, a subscriber may have several subscriptions. The pseudo-code is described in Figure 4.6

```
upon receive CREATE_GROUP(predicate,current_group) at node x
  begin
    group.View := ∅;
    group.Leader := x;
    group.Predicate:=predicate;
    if predicate.op ≠ "="
    Merge_sv(current_group.Succ);
    send(NEW_MEMBER(predicate)) to current_group.Leader;
  end
upon receive SUBSCRIBE_TO(predicate,current_group) at node x from node y
  begin
    group.View := current_group.Leader;
    send NEW_MEMBER(predicate) to current_group.Leader;
  end
upon receive NEW_MEMBER(predicate) at node y from node x
  begin
    if (predicate = group.Predicate)
    group.View:= group.View ∪ x;
    else
      Merge_sv(x,predicate);
      group.blockPublish ← false;
    end if
  end
```

Figure 4.5: Subscription Primitives for Leader-based Approach

4.2.2 Epidemic Communication

The main disadvantage of leader-based communication is due to the special role played by the leader of the group. This is a weak point both for reliability and load balancing. We propose an alternative approach that overcomes these issues compromising the simplicity and efficiency of the leader-based scheme. In epidemic communication [5, 11, 2, 12], each member of a group communicates with a subset of members of other groups. In contrast with the leader-based approach, several copies of a message may traverse the group but failure of a member does not compromise the

```
upon receive PUBLISH_GROUP(e) at node x from node y
initialization:
  Match(e,sublist) → returns true if event e matches one of the subscriptions in sublist
  Notify(event) → deliver event to the application
  IsLeader(x) → returns true if x is the leader of its group
  Group(x) → returns the members of the group whose leader is x
  begin
  if (Match(e,sublist)) then
    Notify(e);
    if (IsLeader(x)) then
      send(PUBLISH_GROUP(e)) to all the members in Group(e);
  end
```

Figure 4.6: Publication Primitives for Leader-based Approach

communication inside the group. Regarding the data structures, each node maintains the *group*, *sublist*, and *succview* variables. *succview* is maintained by each node of a group unlike in the leader-based approach. The size of the group maintained by each node is bounded by size K_g. Data structures are updated for every change of groups and maintained by periodic gossiping.

Epidemic Subscriptions. Similar to the leader-based approach, epidemic propagation of subscriptions is realized through the CREATE_GROUP and SUBSCRIBE_TO primitives. An additional primitive, GOSSIP_SUB, is required to update the views and propagate the update within the group. All these primitives are described in Figure 4.7. Upon receipt of such primitives, the new subscriber updates its variables *group* and *succview*. View update messages are gossiped by each node to F_s other nodes in the group. F_s is called the *subscription fanout*. When a gossip message is received by a node, it is forwarded with probability p, a parameter of the algorithm. To stop the propagation, probability p is reduced by the number of times the message is forwarded. Note that a node issuing a new subscription can receive more than one CREATE_GROUP or SUBSCRIBE_TO messages if the diffusion started from more than one contact points. This does not require any specific check in the algorithms.

Epidemic Publishing. Publications are diffused within a group with a simple gossiping (see Figure 4.8). As for subscriptions, the probability of forwarding an event in the group decreases by the number of times the event is forwarded.

Epidemic approach is prone to undesired behavior when two similar subscriptions are issued concurrently. In particular, if two different nodes in a particular group receive the subscription requests concurrently, two groups corresponding to the same predicate are created. We point out that this behavior is very infrequent and does not harm the correctness of the system. The system continues behaving according to its specification, only suffering from a non-optimal use of resources.

Extending to Generic Approach. Using the above algorithms in the generic approach requires some modifications. A list *predview* containing pointers from the predecessors of a node is required. *predview* must contain processes in non-dummy groups. Upon group creation, the first process in

```
upon receive CREATE_GROUP(predicate,current_group) at node x from node y
begin
   group.View := ∅;
   group.Pred:=predicate;
   if predicate.op ≠ "="
      Merge_sv(current_group.Succ);
      send(GOSSIP_SUB(predicate,group)) to y;
end
upon receive SUBSCRIBE_TO(predicate,current_group) at node x from node y
begin
   group.View := current_group.View;
   send(GOSSIP_SUB(predicate,x)) to F_s nodes in group.View;
   send(GOSSIP_SUB(predicate,x)) to y;
end
upon receive GOSSIP_SUB(predicate) at node y from node x
initialization:
   Size(group) → returns the size of group
begin
   if (predicate = group.Predicate)
      if (Size(group.View) ¡ K_g) then
         group.View:= group.View ∪ x;
      endif
   else
      Merge_sv(x,predicate);
      group.blockPublish leftarrow false;
   endif
   with probability p send(GOSSIP_SUB(predicate)) to F_s nodes in group.View;
end
```

Figure 4.7: Subscription Primitives for Epidemic Approach

```
upon receive PUBLISH_GROUP(e) at node y from node x;
Match(e,sublist) → returns true if event e matches one of the subscriptions in sublist
Notify(event) → delivers event to the application
   begin
      if (Match(e,sublist))
         Notify(e);
      with probability p send(PUBLISH_GROUP(e)) to
         F_s nodes in the group.View;
   end
```

Figure 4.8: Publication Primitives for Epidemic Approach

the group receives both *predview* and *succview* and uses then to initialize its views. It needs to start

a gossiping round in both the predecessor and successor groups to inform them of its presence. The same approach can be followed for the subscription, where a gossip is started within the group. It is necessary to infer if the gossip request arrives from a predecessor or a successor in order to update the right view.

5 Analysis

5.1 Correctness of DPS

In the following we show that DPS verifies the four properties of a publish/subscribe system as per the specification in Section 2.2.

Lemma 5.1 (Legality) *If some node p_i is notified about some event e, then p_i previously subscribed a subscription with a filter f such that e matches f.*

Proof. By construction, a node is notified only with events that match their filters. □

Lemma 5.2 (Validity) *If some node p_i is notified with e, then there exists some node that previously published e.*

Proof. Publishers spread events in DPS only via contact points. That is, any publisher that wants to publish an event should contact a node in the network. If an event arrives at a subscriber then it was received directly from a contact point, or the message was forwarded from the contact point to the node via intermediate nodes. □

Lemma 5.3 (Fairness) *Every node may publish infinitely often.*

Proof. Follows from the publication algorithms. No conditions are imposed on the publishers to publish. □

Lemma 5.4 (Event Liveness) *Node p_i is eventually notified with e if p_i subscribed a filter f such that e matches f and f is stable T_{diff} after e has been published.*

Proof. Assume that event e needs $T_e^{p_i}$ units of time to reach p_i's similarity group. Let T_{diff} be $min_{\forall e,p_i}(T_e^{p_i})$. Each event is propagated in all the trees that match the attributes of the event until finding the groups that do not match the predicates of the event. Since $T_{diff} < T_e^{p_i}$, all the subscriptions that are stable at T_{diff} eventually receive the event in the root-based approach. Each time an event is received by a node in a group, the node checks if the event matches its subscription. On the other hand, in the epidemic approach, nodes are notified only with a high probability since events are gossiped. □

5.2 Self-$*$ properties of DPS

We prove in the following the self-$*$ properties of our system. First, we prove that our algorithms allow nodes to semantically group with respect to the sibling relation $\bowtie$ (formally defined below). Then, we show that the groups self-configure into tree-like structures. Moreover all such structures self-heal whenever they become partitioned.

Lemma 5.5 (Self-organization) *DPS self-organizes in semantic clusters with respect to* $\bowtie$ *relation.*

Proof. By construction, each subscriber in DPS searches similar nodes with respect to the $\bowtie$ relation. The $\bowtie$ relation defines a clustering of the network in logical groups, each being identified with an unique label. □

Lemma 5.6 (Self-configuration) *DPS self-configure into logical trees with respect to the group predecessor relation* $\overset{pred}{\rightarrow}$.

Proof. By construction, each subscriber maintains the *group* data structure that includes information about the current view of the node, and the successors and predecessors groups. This structure is updated whenever a subscriber finds its similarity group. □

Lemma 5.7 (Self-healing) *DPS self-heals.*

Proof. By construction, whenever a group (and the corresponding subtree) becomes disconnected, it executes the subscription algorithm again. □

5.3 Complexity measures

We discuss the scalability of DPS with respect to the message complexity. We also analyze the DPS reliability focusing on the probability that a subscriber interested in a particular filter receives events that match the filter. Note that for concurrent publications/subscriptions, a subscriber while searching for its similarity group, may not be aware of some published events. Moreover, we compare our different approaches (root-leader, root-epidemic, generic-leader, and generic-epidemic) with respect to these two complexity measures.

Reliability. We determine the probability that a new subscriber interested in a filter receives a given concurrently published event. Let us consider a publication e and a concurrent subscription s such that e matches s filter. Let T_s be the number of steps needed by s to find its similarity group (T_s is the subscription turnaround time). Let T_e be the number of steps e needs to reach the s group (T_e is the publication turnaround time). Without compromising the generality, we focus on a single attribute (one tree in the DPS logical structure). Note that subscription s may not "see" event e if the time needed for subscriber s to find its group is greater than that by the publication s, i.e., $T_s > T_e$.

In *root-based* DPS, T_s and T_e are very close since both s and e start at the root of the tree and subscriptions have a higher priority over publications for being processed. Thus subscriptions issued concurrently to events are aware of these events if these events match the subscription filter.

In *generic* DPS, both T_s and T_e depend on the chosen contact point. Hence, it may happen that concurrent publications/subscriptions have different turnaround times. Let p_i be the probability to choose a contact point on the i level of the tree and let s_k be the probability that the similarity group of subscription s is on the level k. T_s is greater than T_e if the number of steps between s contact point and s similarity group is greater than the number of steps between e contact point and s similarity group. More precisely, the probability p that s does not see e is the probability that s contact point is at level i, e contact point is at level j, and s similarity group is at level k,

with $i < j < k$. Formally, $p = \sum_{i<j<k} p_i p_k s_k$. Among f events published concurrently with a new subscription, such that all the f events match the subscription filter, only a fraction $f(1-p)$ are received by the subscriber.

Clearly, the root-based DPS causes fewer lost events than the generic DPS scheme, and thus, is more reliable.

Message complexity We study the maximal number of messages sent by the proposed algorithms we propose. As earlier, we focus only one tree in the logical structure. Let h be the depth of the tree, S_i the maximal size of a group at level i of the tree, k be the number of infected neighbors at each round of the epidemic algorithm, and k' the number of nodes contacted on the next level during the epidemic propagation along the tree.

Let us first consider the *leader-based* communication. In *root-based scheme*, the maximal number of messages corresponds to the traversal of a branch in the tree. Formally, the number is equal to $\sum_{i=0,(h-1)} S_i + (h-2)$. If S is the maximal size of a group, then the maximal number of messages is $h(S+1)-2$. In *generic-based scheme* without bridges, as the contact point may be any node in the tree, we need to consider that an event may traverse, in the worst case, the current branch up to the root and the other subtree from the root down to the bottom. The maximal number of messages is then $2h(S+1)-4$.

Let us now consider the *epidemic-based* communication. The *root-based scheme* produces in the worst case $kS_0 + kk' \sum_{i=1,(h-1)} S_i + k'(h-2)$ messages. If S is the maximal size of a group, the maximal number of messages is then $kS(1+k'(h-1)) + k'(h-2)$. Similarly, the *Generic based scheme* produces $2(kS(1+k'(h-1)) + k'(h-2))$.

As expected, the *root-based scheme* costs less than the *generic-based* one. Note that the *generic-based scheme* with bridges has the same message complexity as the *Root-based* scheme.

6 Simulations

In this section, we present the results of an experimental evaluation of our system performed using an event-based simulator we developed. The aim of the simulation is to prove the practical feasibility of our approach and to show that self-* properties can be achieved in a scalable manner without introducing serious overheads on message delivery. We decided not to compare with other systems (e.g., Siena) because they are either not closely related to our work or have very different parameters. As an example, let us consider the event routing. The network performance of Siena is highly influenced by the connections among the brokers and the distribution of subscriptions on the brokers, both being controlled by the users. Thus, all the resulting performance metrics (number of messages and memory consumption) are strongly dependent on the users' choice. So, it is not possible to fairly compare the performance with our approach in which connections among nodes are automatically built by the system according to the distribution of subscriptions. Rather than comparing with other approaches, our motivation behind running the simulation is to show that our system maintains self-organization and reliability without compromising the scalable event dispatching. Following this direction, we compare all the combinations of the approaches presented in the paper.

Simulation Context. A simulation execution consists of a set of consecutive steps. All the nodes in the system act as subscribers. An execution is preceded by an initialization phase in which all nodes subscribe to three subscriptions. Each subscription is composed of a randomly chosen number

of numerical predicates. There can be up to five predicates in each subscription. The attribute within each predicate is chosen from a set of 10 distinct numerical attributes. The distribution of attributes follows a power-law distribution, while distribution of values is uniform. Events have the same characteristics as subscriptions have. Typically, data distribution in pub/sub system may follow a non-uniform trend [18]. However, we chose to follow uniform distribution in order to test conditions which are not favorable for our system. A biased distribution could create less groups and shorter branches, reducing the number of messages especially in epidemic approaches. In the leader-based approaches, each group contains one leader and one co-leader, and the failure detection and recovery mechanism is fully implemented. Each node is able to detect the failure of one of its neighbors. Detection is not instantaneous, it takes from 10 to 25 steps. This simulates a common failure detector based on heartbeat messages.

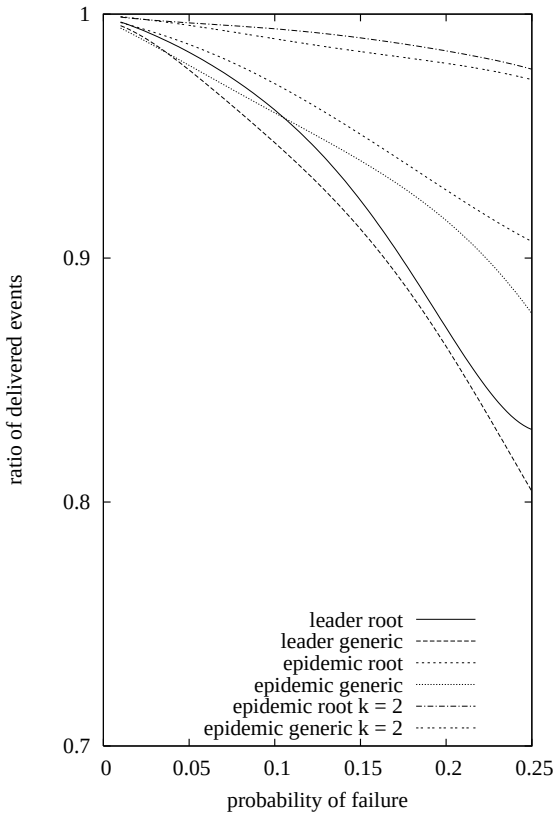
6.1 Dependability and Scalability

In the following we present experimental results related to dependability and scalability. Results are plotted in Figure 6.9

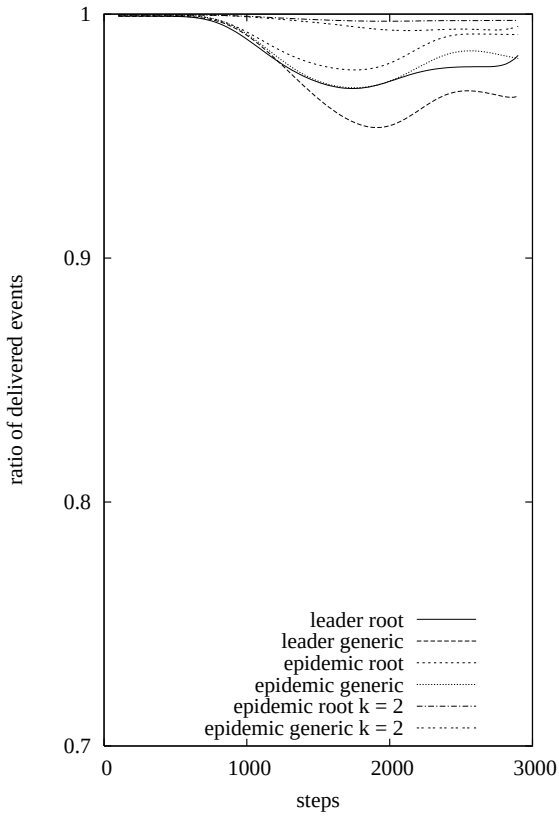
Dependability. In these sets of simulations, we test the ability of the system in delivering messages despite nodes failures. We built two different scenarios aimed at highlighting different aspects of our system. In both scenarios, the system initially contains 1.000 nodes and the execution is 3.000 steps long. In the first scenario, at each step, a node may fail with a probability p that varies between 0.01 and 0.25. In other words, node failures are uniformly distributed in time, with a frequency of $1/p$. To give a more precise idea of this parameter, we say that at the end of the simulation, the system contains, on an average, a number of nodes that ranges from 25% to 97% of the initial nodes with p ranging from 0.25 to 0.01. A new event is published every 10 steps. This scenario tests a realistic situation where nodes disappear independently and in an unpredictable manner. In the second scenario, execution is divided into three phases. Nodes do not fail until step 1000, then one node fails every two steps between steps 1.000 and 2.000, and finally the system resumes normal execution without failures until the end of the execution. This scenario tests the recovery capabilities of the system after a large number of nodes failures. In both cases, we measure the ratio of correctly delivered events, i.e., the percentage of published events that reaches a node with a matching subscriber. In the second scenario, we measure the ratio of delivered events for the ten last events, while we measure the overall ratio in the first scenario.

Figure 6.9.a shows the results of the first scenario. First of all, we point out that in all the approaches, the system can reach a number of delivered events which is at least 80%, and also when most of the nodes have failed. This is obtained with the leader-based approach, which as expected is the less 'robust' one. Increasing the number of co-leaders may offer a way to prevent such situations. The epidemic scheme confirms the expected higher number of delivered events than the leader-based approach. In particular, setting *epidemic k=2* allows a ratio greater than 0.97 even with a significant probability of failures. Results of the second scenario are exhibited in Figure 6.9.b. We can easily observe that the ratio of delivered messages is still high as events are delivered in more than 95% of cases. These curves show that the system is able to self-recover as the ratio increases after step 2000. In particular, the epidemic approaches succeed in recovering the ratio to very close to 1.

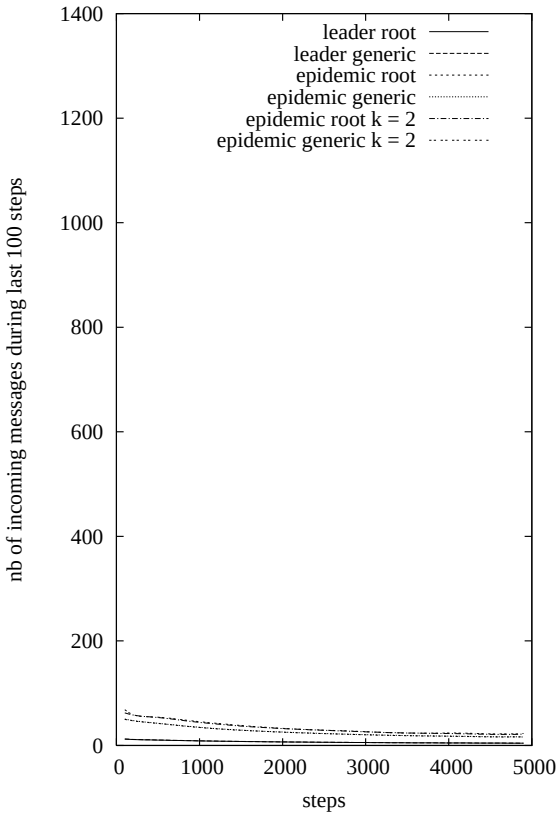
Scalability. We tested the ability of the system to scale by measuring the load managed by nodes when propagating events and subscriptions as the size of the system grows. In this simulation



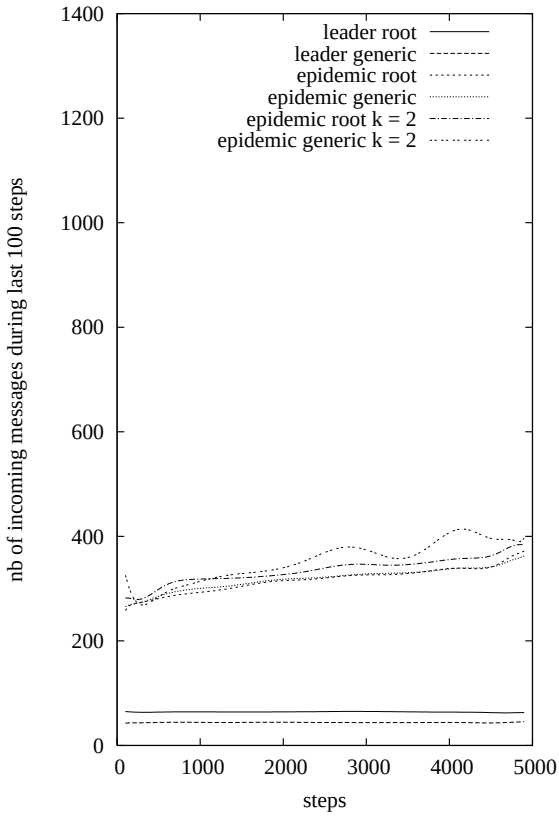
(a) Dependability: ratio of delivered messages



(b) Dependability: ratio of delivered messages

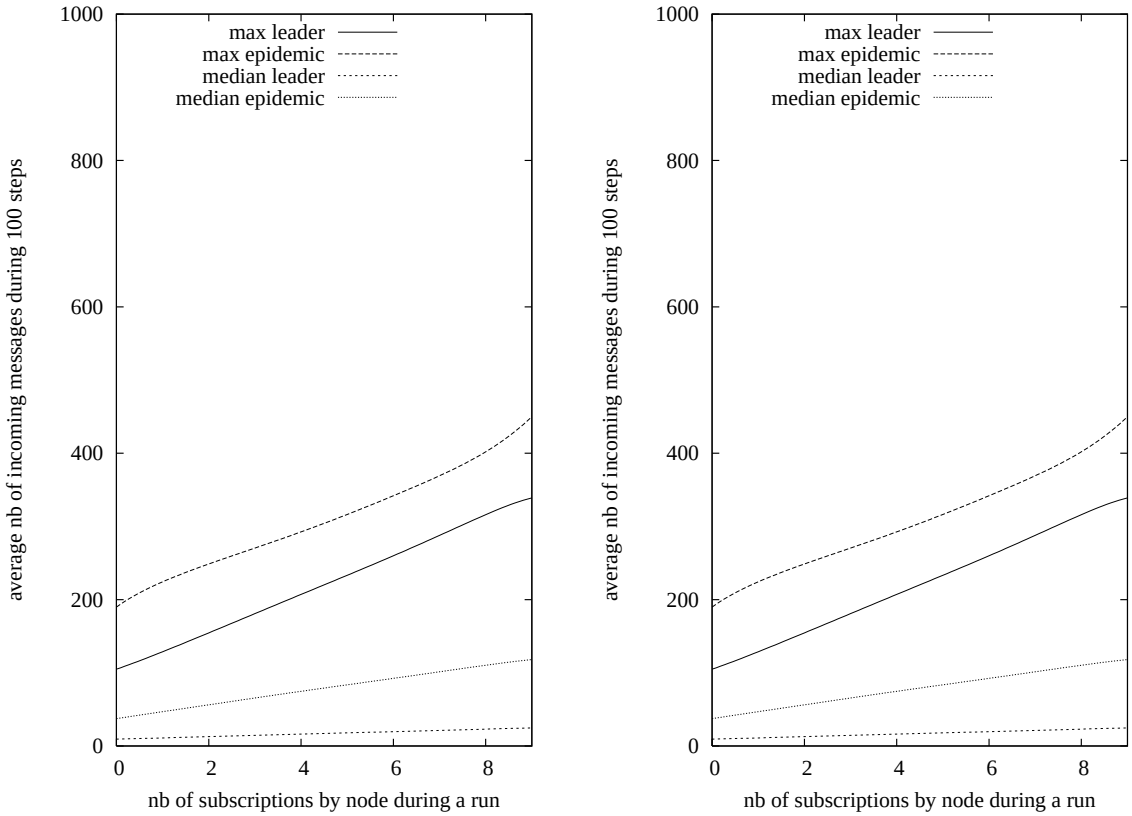


(c) Scalability: messages sent per event (med)



(d) Scalability: messages sent per event (max)

Figure 6.9: Experimental Results



(a) Dependability: ratio of delivered messages

(b) Dependability: ratio of delivered messages

Figure 6.10: Root-based vs. Generic Approach

scenario, the system initially contains 1.000 nodes. A new node enters the system every two steps and immediately emits a new subscription. As the duration of the execution is 5.000 steps, the system contains 3.500 nodes at the end of it. Publications are emitted at a regular rate along system execution (10 new events every 100 steps). Figures 6.9.c and 6.9.d provide the results for this scenario in each configuration. The two plots report on the x axis the number of nodes in the system and on the y the number of messages sent by a node per each event. The plot on the left refers to the median node, defined as the node that sends less messages than half of the nodes and more messages than the other half. The plot on the right refers to the most overloaded node. Median node sends no messages in leader-based approach, hence they are no plotted in Figure 6.9.c. The two plots show that in general the number of messages per event does not increase with the nodes, confirming the overall scalability of our approach. The only exception is the most overloaded node in the leader-based approach which has to handle more messages as system grows, because the size of the groups increases accordingly. Surprisingly, root-based and generic approaches turned out to have the same performance. This is due to the small height of the trees.

6.2 Root vs. Generic

We first compare the root-based and the generic approach. The results are obtained with a leader-based implementation and are presented in Figure 6.10.

Incoming Traffic. In the root-based approach, an owner receives all messages related to its attribute. As the number of subscriptions increases, the number of messages it should treat increases. On the contrary, in the generic approach, we may expect that the most overloaded node is the leader of the group located in a median position in the tree. This node does not receive (1) the events that does not match its filter when the contact node is one of its predecessors, (2) the subscription message when it does not lie to the path between the contact node and the target group. So, this node receives only a part of the traffic. Moreover, as the ratio of subscriptions in the overall traffic increases, the difference between the generic and the root-based approach increases.

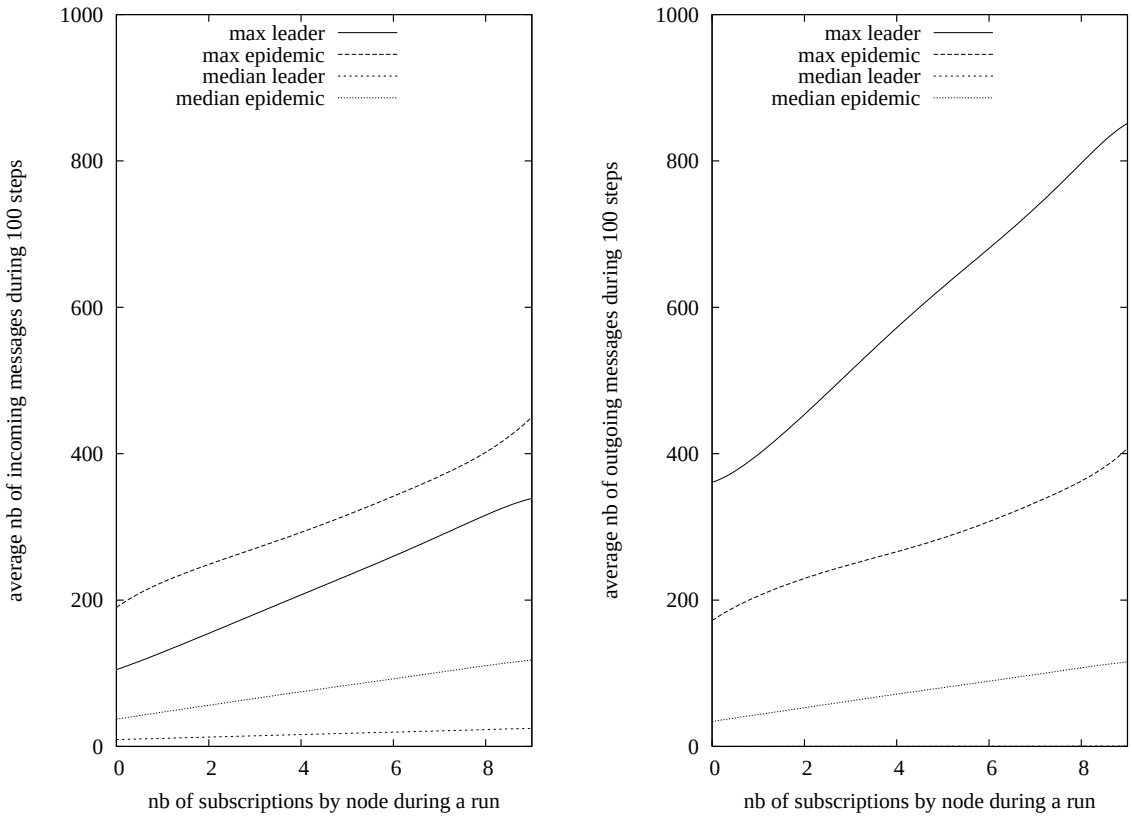
Outgoing Traffic. These experiements reveal few differences between the two approaches. The most overloaded node is a leader which should send every matching events to all members within its group. As the number of subscriptions increases, the number of nodes by group, and subsequently the leader outgoing traffic, increases as well. So, the majority of outgoing traffic of the most overloaded node is due to event diffusion. That is, both approaches exhibit similar results. Yet, the increasing difference shows that the subscription mechanism is less efficient in the root-based approach. Indeed, the average number of nodes impacted by a subscription is higher in the root-based approach than in the generic one. As using the leader-based approach, in both cases, the median node does not send any messages during the run.

6.3 Leader vs. Epidemic

We compare the leader-based and epidemic approaches for inter-group communication. Results are produced with a root-based approach and are presented in Figure 6.11

Incoming Traffic. In the leader-based approach, all nodes receive an event once. On the contrary, in the epidemic approach, the diffusion within a group implies some redundant message reception. This explains the difference between the leader-based and the epidemic communication requirements for the most overloaded node. As this only occurs during event dissemination, the difference is constant when the number of subscriptions increases. The median node in the leader-based scheme is only a receiver of events. In the epidemic approach, a node may be informed when a new predecessor or successor enters in the system. So, subscriptions produce messages that are received by some nodes in the system. This explain why the number of treated messages increase in the epidemic implementation while almost remaining constant in the leader-based scheme.

Outgoing Traffic. The more active are the nodes, the more dense are the groups. In the leader approach, the leader should send a message to all members within its group. Obviously, the outgoing traffic of a leader increases with node activity. But we also observe that the median node does not send any message, whatever the activity of nodes. In fact, the activity increasing is only assumed by the leaders. Naturally, this responsibility may overcome the capabilities of these nodes. In the epidemic approach, a node may theoretically have a constant number of neighbors. However, first nodes are the contact node of numerous predecessors and successors. When a new node joins a



(a) Dependability: ratio of delivered messages

(b) Dependability: ratio of delivered messages

Figure 6.11: Leader vs. Epidemic Approaches

group, it contacts the node known by its predecessor. Hence, the first nodes should admit more neighbors than other node. So, even if nodes may theoretically have a constant number of neighbors, in practice this does not happen. This is the main reason of the increasing of the difference between the most overloaded node and the median node. But, in general, we observe that the overcost due to node activities is distributed on participants.

7 Related Work

Decentralized solutions to the content-based publish/subscribe scheme rely on either multicasting notifications within the broker network or filtering events at intermediate brokers. Filter based approaches require each broker to have the full knowledge of all the active subscriptions. This knowledge is maintained by flooding subscriptions to the network of brokers. Both Siena [7] and Gryphon [4] work along these lines. In Siena, subscription propagation is limited by exploiting containment relationship among them. Moreover, none of these two systems is able to self-organize itself. Both require connections among brokers to be set up manually. Gryphon also requires all the paths for events to be updated by users upon subscription change. Multicasting schemes [14, 15] are based on an underlying multicast primitive used to propagate events. Multicast groups are defined according to a partitioning of the event space, where each partition is associated with a group.

An interesting approach to event delivery is Kyra [6]. Kyra is a hybrid routing scheme based on a network of event brokers in which the event space is partitioned as is done in multicast approach. Each partition corresponds to a separated diffusion tree, in which events are filtered like in SIENA. Subscriptions are moved to the servers that are responsible for the partition in which the subscription falls. Connections among brokers are set up according to physical proximity metrics. Though the scheme is proved to be efficient, Kyra does not provide self-organizing dynamic behavior. When a node joins or leaves the system, the whole partitioning has to be recomputed and subscriptions have to be moved from one broker to another. This process can possibly involve several brokers, so it may not be scalable.

Self-organizing pub/sub systems, Scribe [8] and Bayeux [19] are built on top of DHT facilities. Both systems are based on a simple topic-based addressing schemes. The name of the topic can be associated with a DHT address through hashing, and the corresponding node becomes responsible for matching and delivering all the notifications related to that topic. A diffusion tree for delivering notification to subscribers is built dynamically and maintained by exploiting the DHT self-organization abilities. A method to map content-based subscriptions to DHT addresses is described in [17]. Recently, a scalable content DHT-based publish/subscribe system, namely Meghdoot, was proposed in [10]. Meghdoot works on top of a CAN network and uses CAN nodes to store the system subscriptions. Hence, the persistence of the subscriptions in the system is totally dependent on reliability of the CAN nodes. The replication was used to maintain the reliability.

Our work includes a gossip-based epidemic-like diffusion scheme. A hierarchical gossip-based algorithm for pub/sub is described in [2]. The addressing scheme used is topic-based that allows a simple grouping of nodes but gives users only limited possibilities for expressing their interest. Gossiping techniques for content-based pub/sub have been exploited in [9]. This paper does not include any form of self-organization. The same authors propose a self-organization algorithm for content-based pub/sub in [13]. However, self-organization is limited to repairing of routing tables whenever the connections among nodes are rearranged, for example, due to node mobility.

8 Conclusion and Discussion

We have presented DPS, a distributed reliable and scalable content-based publish/subscribe system that exhibit self-∗ characteristics. In our system, the subscribers self-organize into similar semantic groups (clusters). Furthermore, semantic clusters self-configure into logical trees that are used further to efficiently publish events in the network. In order to mask unexpected failures/unsubscriptions we reinforced the logical connectivity of the DPS trees using additional links between a semantic group and its neighbors. A nice feature of DPS is its ability to self-heal using only local information. That is, it can adjust in an autonomous and local manner yielding a stable structure in the presence of dynamic and mobile nodes.

We proposed different methods of diffusion of subscriptions and publications that can be combined to obtain four different implementations of the system. Results show that the number of messages handled by a node at increasing load (i.e. more subscriptions) is in average lower in the leader-based approach than in epidemic, with nodes in average handling 30% less messages. However, the load is better distributed in epidemic (the most overloaded node handles in epidemic less than half the messages than in leader-based). Based on the simulation results, we can conclude the following: The leader-based approach is more suitable for a relatively small set of nodes that are less prone to failures. The epidemic-approach provides higher dependability, better scalability, and load balancing at the cost of higher message complexity. The possibility of choosing different implementations makes the proposed system very versatile, so it can be deployed in many applications. One potential application area is in sensor networks. In almost all sensor networks, special purpose sensors (e.g., only for measuring temperatures or for detecting movements, etc.) (behaving as publishers) are used. Some other sensors (or agents) acting as subscribers express their interest in measuring or detecting some events by using appropriate filters. A future direction we intend to explore is the evaluation of DPS in such context or in other scenarios where real-world subscriptions and publications are injected.

References

[1] E. Anceaume, A. K. Datta, M. Gradinariu, and G. Simon. Publish/Subscribe Scheme for Mobile Networks. In *Proceedings of the Workshop on Principles on Mobile Computing (POMC'02)*, 2002.

[2] S. Baehni, P. Eugster, and R. Guerraoui. Data-Aware Multicast. In *5th IEEE International Conference on Dependable Systems and Networks (DSN '04)*, 2004.

[3] R. Baldoni, R. Beraldi, S. Tucci Piergiovanni, and A. Virgillito. Measuring Notification Loss in Publish/Subscribe Communication Systems. In *Proceedings of Proceedings of the 10th International Symposium Pacific Rim Dependable Computing (PRDC '04)*, 2004.

[4] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach. Exactly-once Delivery in a Content-based Publish-Subscribe System. In *Proceedings of The International Conference on Dependable Systems and Networks*, 2002.

[5] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

[6] F. Cao and J. Pal Singh. Efficient Event Routing in Content-based Publish-Subscribe Service Networks. In *Proceedings of 23rd Conference on Computer Communications (IEEE INFOCOM 2004)*, 2004.

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

[8] M. Castro, P. Druschel, A. Kermarrec, and A. Rowston. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8), October 2002.

[9] P. Costa, M. Migliavacca, G.P. Picco, and G. Cugola. Epidemic algorithms for reliable content-based publish/subscribe: An evaluation. In *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, 2004.

[10] A. Gupta, O.D. Sahin, D. Agrawal, and A. El Abbadi. Meghdoot: Content-based publish:subscribe over p2p networks. In *ACM/IFIP/USENIX 5th International Middleware Conference (Middleware'04)*, 2004.

[11] A.-M. Kermarrec, L. Massoulié, and A.J. Ganesh. Probabilistic Reliable Dissemination in Large-Scale Systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3), March 2003.

[12] Meng-Jang Lin and Keith Marzullo. Directional gossip: Gossip in a wide area network. In *European Dependable Computing Conference*, 1999.

[13] G. P. Picco, G. Cugola, and A. L. Murphy. Efficient content-based event dispatching in the presence of topological reconfiguration. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 234–243, 2003.

[14] A. Riabov, Z. Liu, J.L. Wolf, P.S. Yu, and L. Zhang. Clustering algorithms for content-based publication-subscription systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), Vienna, Austria*, 2002.

[15] A. Riabov, Z. Liu, J.L. Wolf, P.S. Yu, and L. Zhang. New algorithms for content-based publication-subscription systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, pages 678–686, 2003.

[16] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location androuting for large-scale peer-to-peer systems. In *Proc. of the 4th IFIP/ACMv International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, pages 329–350, 2001.

[17] P. Triantafillou and I. Aekaterinidis. Content-based Publish/Subscribe over Structured P2P Networks. In *Proceedings of the 4th International Workshop on Distributed Event-Based Systems*, 2004.

[18] Y. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. J. Wang. Subscription Partitioning and Routing in Content-based Publish/Subscribe Networks. In *DISC'02*, 2002.

[19] 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 *Int. Workshop on Network and OS Support for Digital Audio and Video*, 2001.