Community-Aware Event Dissemination [⋆]

Sébastien Baehni [a] Patrick Th. Eugster [c]
Rachid Guerraoui [b] Oana Jurca [a]

[a] *School of Computer and Communication Systems, EPFL*

[b] *School of Computer and Communication Systems, EPFL, Tel: +41 21 693 5272, Fax: +41 21 693 7570*

[c] *Department of Computer Sciences, Purdue University*

Abstract

This paper presents a distributed algorithm to disseminate events in a publish/subscribe system, where processes publish events of certain topics, organized in a hierarchy, and expect events of topics they subscribed to. Every topic defines a dynamic notion of "community", gathering the processes which publish on that topic or subscribe to it. Our algorithm is completely decentralized (no brokers), yet does not require from any process to ever receive, store or forward, events from a community it is not part of.

We order the communities according to the topic inclusion relationships to efficiently manage the flow of information within, and between the communities, as well as limit the memory consumption of each process. Processes can control, for each of their communities, the trade-off between the message complexity and the reliability of event dissemination. We convey this trade-off through analysis, simulations and measurements obtained with a full implementation of our algorithm.

Key words: Multicast, peer-to-peer, reliability, topic-based publish/subscribe

[⋆] This paper is a revised and extended version of a paper that appeared in the proceedings of 5th IEEE International Conference on Dependable Systems and Networks (DSN' 04), under the title *Data-Aware Multicast.*

Email address: `Rachid.Guerraoui@epfl.ch` (Rachid Guerraoui).

1 Introduction

1.1 Context

The publish/subscribe interaction scheme is very attractive for building large scale distributed applications, especially in situations where a user has better things to do than poll a specific site to find out what is new. In its *topic-based* form (the most popular variant of publish/subscribe), events are published to "topics", which can be viewed as named logical channels. Subscribers expect to receive all events published to the topics to which they subscribe. With the topic being the focus, publishers and subscribers can continue to operate normally regardless of each other. This is in contrast to the traditional tightly-coupled client-server paradigm, where the client cannot post messages to a disconnected server, nor can the server receive messages unless the client is running.

Clearly, the publish/subscribe interaction pattern provides the opportunity for better scalability than the traditional client-server one. This goes however through designing a scalable underlying publish/subscribe infrastructure. The main functionality of such infrastructure is the delivery of published events from producers (publishers) to interested consumers (subscribers). Publishers publish information events through the infrastructure and subscribers express their interest in specific events through the infrastructure. Building a centralized publish/subscribe infrastructure is convenient but does not scale. Many systems rely on brokers to improve the reliability. Tibco [1], iBus [2] and Vitria [3] were early innovators, but others followed the path: IBM's MQSeries [4], Microsoft's MSMQ [5], and the Java Message Service [6] (as implemented by IBM, Sonic, Sun and BEA).

Broker-based solutions (e.g., NNTP [7]) have limited scalability: brokers represent bottlenecks and may impose unacceptable regulations on the rest of the processes. It is thus intriguing to figure out how to implement a publish/subscribe infrastructure in a completely decentralized (also called *peer-to-peer*) manner. In a sense, we can see in each topic a *community* of processes that collaborate to exchange/forward events of interest[1]. As topics are usually organized according to a hierarchy, this would lead to a hierarchization of communities.

1.2 Challenge

Ideally, we expect from a topic-based publish/subscribe infrastructure to be highly reliable: every process, when subscribing to a given topic, should receive all events

[1] In a publish/subscribe infrastructure, a process is interested in a topic if it either publishes events of that topic or if it subscribes to this topic.

published on that topic as well as on any of its subtopics. In addition, even if it seems natural that processes collaborate to disseminate events within their community, it might not seem fair for processes to help disseminate events related to communities they are not part of; events from such communities might rather be viewed as *parasite* events. To illustrate this, assume an event-based system used to exchange events about the soccer world cup. French fans would hardly accept to store, or forward events related to the Italian team. As we will discuss below, this natural constraint raises some problems when considering large scale event dissemination.

In general, it is appealing to consider *gossip-based* (also called *epidemic* or unstructured) algorithms ([8–12]) to support large scale information dissemination in decentralized systems. According to these algorithms, once a process has joined the system, the process requires only local neighborhood knowledge [11] to disseminate, with a high reliability, events to a large set of processes: this is not surprising as the dissemination scheme is patterned after the spread of a contagious disease among a population. However, implementing the topic-based publish/subscribe abstraction with a gossip-based scheme, while eliminating parasite events, is not a trivial task. In gossip-based algorithms [13], processes are typically merged together into one big community, irrespective of their interests. Consequently, events are gossiped among the entire set of processes, relying on processes storing and forwarding all events, including parasite ones.

The very same problem occurs if, for building a publish/subscribe infrastructure, the dissemination is based on distributed hashtables (called DHTs) as for Scribe [14] using Pastry [15] or HiCan [16] using CAN [17]. With such a scheme, typically considered as *structured* in contrast to gossip-based ones, which are considered *unstructured*, the processes are uniformly distributed according to their identifiers along a virtual ring (representing the DHT). A process only needs to know a small subset of the entire set of processes to route events to every process in the system. However, due to the uniform disposition of the processes in the ring, the creation and use, for instance, of a spanning tree to route events of a specific topic t to all the processes interested in t, is likely to involve processes which are not interested in t, hence generating parasite events. Furthermore, spanning trees are sensitive to failures of processes located at the nodes of those trees and even if fault-tolerance mechanisms can be used, these are resource-consuming and the nodes must have more bandwidth and processing power than regular processes.

Breaking the global community into smaller ones, corresponding each to a topic, is an appealing approach to prevent parasite events. One can indeed disseminate events inside smaller groups without hampering the reliability of the dissemination. However, as we discuss below, this approach significantly increases memory complexity, because inclusion relationships between topics engender an unpleasant phenomenon that we call *hereditary peers*. This phenomenon is related to the need of maintaining enough overlap between multiple communities of processes in order to enable reliable propagation. In short, a straightforward mapping of topics to communities either leads to gather exactly (1) the *publishers* of a topic, or (2) the *subscribers* to a topic. With (1), a subscriber to a topic t not only has to become member of the community *corresponding* to t, but also of any community corresponding to a subtopic of t. This increases the subscriber's memory consumption to a large extent through redundant information, and involves further communication; the subscriber also has to be informed of the creation of any new subtopics of t, and has to join these communities eventually. With (2), a publisher of a topic t has to publish its events within the community corresponding to the topic t and all the super-topics of t. This increases the load on the publishers, making of these bottlenecks in the dissemination of events. The hereditary peer phenomenon is exemplified by [18] that exploits *overlappings* between the communities to limit redundant gossiping of events (hence parasite events). This approach is useful when processes are interested in many *distinct* topics, yet does not take into account *inclusion relationships* between topics, hence its sensibility to the number of subtopics of a given topic.

Content-based publish/subscribe systems might at first glance appear to fulfill our requirements but however are all prone to either hereditary peers or parasite events. For instance, SIENA [19] and Gryphon [20] rely on a network of dedicated application-level routers ("brokers") to achieve efficient content-based filtering and forwarding. In [20], process subscriptions are matched to IP multicast groups. Therefore, the maintenance and the creation of the matching between interests and IP multicast groups involves all processes and is maintained by a central server (a single point of failure). In [19], the published events are routed from the more general filter to the most specific one: brokers responsible for a general filter are heavily loaded. In Hermes [21], the propagation of events is also done with the help of brokers and follows a top-down approach: a parent topic must keep a reference to all its descendants (i.e., hereditary peers phenomenon). In PMcast [22], as in Astrolabe [23], the processes are arranged into a hierarchy to (1) reduce the memory complexity of each process and (2) perform efficient filtering without the help of brokers. However, the processes elected to accomplish the actual filtering receive parasite events and, especially if higher up in the hierarchy, must be capable of handling a large number of events.

To summarize, it is challenging to devise a decentralized topic-based publish/subscribe dissemination algorithm, be it gossip-based or DHT-based, while preventing processes from dealing with parasite events, without leading to a large and unbalanced load on processes.

1.3 Contributions

This paper presents a simple decentralized algorithm, denoted *CAMCAST* (for community-aware multicast), that reliably disseminates events in a large-scale system made up of dynamic communities of processes related by topics of interest. Our algorithm ensures that processes only help disseminating events in their communities and prevents both parasite events and hereditary peers.

CAMCAST organizes the set of communities following the hierarchical disposition of the topics they represent. Published events are propagated *within* every community in a gossip-based manner, and disseminated in a balanced manner *between* communities related by topic inclusionships, following the bottom-up approach of the topic-hierarchy. This organization of communities (or "community-awareness") is key to prevent both hereditary peers and parasite events. Basically, each community representing a topic t must only forward the events, related to the topic t, to their "parent" communities (i.e., the communities representing the parent topics of t). The parent communities in turn forward the events to their parent communities and so on until the events reach the root ancestors of the topic-hierarchy. The processes within a community that actually forward the events are chosen in such a way that the overall load is evenly balanced among them.

CAMCAST induces a memory complexity of any process within a community of a topic t is in the order of $ln\, N_t + c_t + z_t$: N_t denotes the number of processes in the community of a topic t whereas c_t, z_t denote constant values for each t explained in more details later. On the other hand, The message complexity involved in the publication of an event of topic t grows in the order of $O(N_{max(t)} \cdot ln\, N_{max(t)})$, where $max(t)$ denotes the (super) topic of t with most subscribers. Interestingly, the processes can trade, for every topic pertaining to the considered hierarchy, the message complexity of the dissemination with the reliability of this dissemination. This trade-off is useful for adjusting the reliability of the dissemination between communities in the same hierarchy according to the reliability of inter-process communication.

CAMCAST is modular in that the dynamic management of peers within each community is achieved by a separate decentralized membership algorithm: we can use here a gossip-based or a DHT-based membership subprotocol. We exploit the information provided by the underlying membership protocol to gossip events inside a community as well as between topic-related communities. Assuming for instance that *CAMCAST* makes use of an underlying membership subprotocol M (e.g., [9,11,12]) then, in the extreme case where no topic relationship information is available (or if there is only one topic in the system), *CAMCAST* does not degrade in terms of reliability, or increase in memory, message, or latency of the bare membership subprotocol M.

1.4 Evaluation

We convey the performance of *CAMCAST* with analytical results, simulations as well as with performance measurements obtained with a fully decentralized Java 1.5 implementation of our algorithm (over Sun Ultra workstations using Solaris 8 and connected through a 100MB LAN).

Our analytical results compare *CAMCAST*, in terms of message complexity, memory complexity, latency complexity and reliability, with three gossip-based alternatives: (a) gossip-based broadcast, (b) gossip-based multicast and (c) hierarchical gossip-based broadcast. We also compare, in terms of reliability and efficiency, three different versions of a Java implementation of *CAMCAST*, each using a different underlying membership algorithm: (1) an unstructured (i.e., gossip-based) membership algorithm ([24]), (2) a semi-structured membership (i.e., both DHT-based and gossip-based) algorithm ([25]), and (3) a completely structured (i.e., DHT-based) membership algorithm ([25]).

In short, our simulation performed on a hierarchy of three communities made of 1000, 100 and 10 processes respectively demonstrate that, with *CAMCAST*, the reception probability of an event is degraded by only 5% despite the crash of 30% of the processes with respect to a gossip-based approach (that does not preclude parasite events). Furthermore, the measurements conducted over our implementation show that, in a hierarchy of three communities made of 80, 30 and 7 computers respectively, it is possible to achieve 100% reliability while only involving 7% of the processes in the propagation of the events between the communities.

1.5 Roadmap

The rest of the paper is structured as follows: Section 2 introduces the model we consider. Section 3 describes *CAMCAST*. Section 4 analyses it and compares it with alternative approaches. Section 6 presents simulation and implementation results. Finally, Section 7 summarizes the paper.

2 Distributed System Model

2.1 Terminology and Notations

A *community corresponding* to a topic t, denoted by Π_t, is a set of processes that either publish events of topic t, or have subscribed to topic t (in both cases, the processes are said to be *interested* in t). A process p_i is said to *participate* in a community Π_t (e.g., *.soccer.italy*) if p_i is interested in t. For presentation simplicity, we assume that a process participates in one community Π_t in the topic hierarchy S of t only (and as a consequence to all subtopics of t) and a topic t has only one super-topic (single inheritance). We come back to the issue of subscribing to multiple topics as well as to the multiple inheritance problem in Section 3.6. The topic that does not have any super-topic is called a root topic (thus, in the single inheritance case, there is only one root topic in a topic hierarchy S). For instance, in *.soccer.italy*, the first . sign represents the root topic. The single super-topic of a topic t is denoted *super(t)*. The *root community* is the community of processes interested in the root topic. The level of a topic t in a topic hierarchy S is the length of the path from the root topic of S to t. The *depth* of a topic hierarchy S is equal to the highest level of the topic of the hierarchy and is abbreviated by *dpt(S)*. By extension, a super-community $\Pi_{super(t)}$ of a community Π_t is a community of processes interested in *super(t)*. We talk about *inclusions* of topics when a topic u is a super-topic of t (in this case we say that u includes t). The same goes for inclusion of communities. For instance, in *.soccer.italy*, *soccer* includes *italy*. Finally, we say that p_k ($\in \Pi_u$) is a *super-process* of a process p_i ($\in \Pi_t$) if p_k is participating in a super-community Π_u of Π_t.

We denote by Ψ^u_m, a subset of the set of processes interested in topic u and known by process p_m. A process p_i communicates with another process p_j via unreliable, i.e., best effort, channels and processes might crash and recover (a process that is not crashed is said to be "alive"). The number of processes in a community Π_t is denoted by N_t which represents the cardinality of Π_t.

A published event of a specific topic t is denoted by E^t_j. The *topic table* for a specific topic t of a process p_i, denoted by $Table^t_i$, contains information about processes interested in t. The *super-topic table* for a specific topic t of a process p_i, denoted $sTable^t_i$, contains information about processes interested in *super(t)*. In the case where no process is interested in *super(t)*, i.e., no direct super-process(es) exist(s), $sTable^t_i$ contains information about processes interested in the next immediate super-topic of t (e.g., *super(super(t))*), according to the topic hierarchy level that includes t.

2.2 Membership Protocols

The basic concepts underlying *CAMCAST* can be combined with various underlying membership techniques (e.g., DHT-based, gossip-based, see Section 5) to support the dissemination of events. In this paper, to make our presentation concrete, we assume the membership technique of [24] as an underlying building block. Basically, with this technique, each process gossips an event to $ln\, N_t + c_t$ target processes and the probability that every process receives the event goes to $e^{-e^{-c_t}}$ as N_t

goes to infinity. [2]

3 The *CAMCAST* Algorithm

In this section, we describe our *CAMCAST* algorithm. We first the basic idea, then we go through the algorithm in more details.

3.1 Basic Idea

Consider an event of topic d published by a process p_2, and another process p_5 subscribing to topic a, where a is the super-topic of d. There are two basic ways according to which events of topic d can be transmitted to p_5:

(1) A community is created for the *publishers of a topic* (this is done for each topic and corresponds to the dashed arrows in Figure 1); a subscriber (p_5) of topic a becomes a member of the community Π_a and member of all the communities of the subtopics of a (in this case d). When an event of topic d is published, this event is only disseminated within Π_d.
(2) A community is created for the *subscribers of a topic* (this is also done for each topic and corresponds to the plain arrows scenario of Figure 1); the subscriber p_5 for topic a becomes only a member of the community Π_a. When an event of topic d is published, this event is disseminated in the community Π_d and to all the communities of all the super-topics of d.

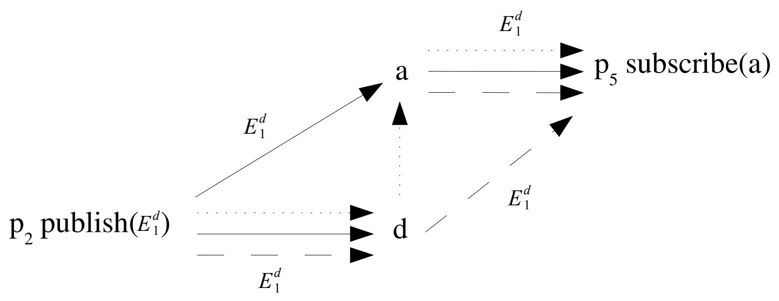

Fig. 1. Publication/Subscription alternatives. The 1st alternative is represented by the dashed arrows. The 2nd alternative is represented by the plain arrows and our approach is represented by the dotted arrows

The first approach overloads the subscribers, which must store redundant membership information, whereas the second overloads the publishers, which must publish

[2] Note that this probability assumes a random membership algorithm [24]. In practice, this can at best be approximated [26].

in several communities. *CAMCAST* follows the second approach but limits the load, for both the publishers and the subscribers (dotted arrows of Figure 1).

We exploit the topic hierarchy to limit the membership information maintained by every process. The processes are split into communities corresponding to the topics they are participating in. These communities are created and maintained dynamically when the processes join or leave the system. To join a community, a process goes through an initialization phase to initialize the topic and the super-topic tables (that compose the membership table) for that process. Once the process joins a community, the underlying membership algorithm takes care of maintaining the topic tables.

The dissemination of an event is depicted in Figure 2. Namely, a process p_1 sends its events to at least one process, p_2, from its super-topic table, and then gossips the event to selected processes (p_3, p_4) in its community. When a process (p_2, p_3 or p_4) receives the event for the first time, it gossips the event within its community and, with a certain probability, disseminates the event to some process in its super-topic table through *upward messages* (inter-community messages employed to disseminate events from a community to its super-community). As long as there is a super-community with participating processes, the event shifts up to the next super-community. When the event reaches the root community, the processes receiving the event only gossip it in their community. Note that it is not mandatory for a publisher p_1 to ensure the propagation of the events into its super-community. If p_1 fails to do so, another process (here p_4) from the community does the job for p_1. Hence, once a publisher (p_1) has transmitted its event to at least another process in its community, the publisher can leave this community. This "infect and die" scheme imposes very little on the availability of the processes.

3.2 Membership

In *CAMCAST*, every process participating in a community Π_t maintains information about other processes participating in Π_t and the direct super-community $\Pi_{super(t)}$.

3.2.1 Membership Tables

The identifiers (IDs) of selected processes participating in Π_t are stored in a topic table ($Table_i^t$, see Figure 3) and maintained by the underlying membership algorithm (which sets the size of the topic table). The second table (super-topic table, $sTable_i^t$) contains IDs of several processes participating in the super-community of the community corresponding the topic of interest. This table has a constant size z_t.

Note that the super-topic table might not contain IDs of processes participating in

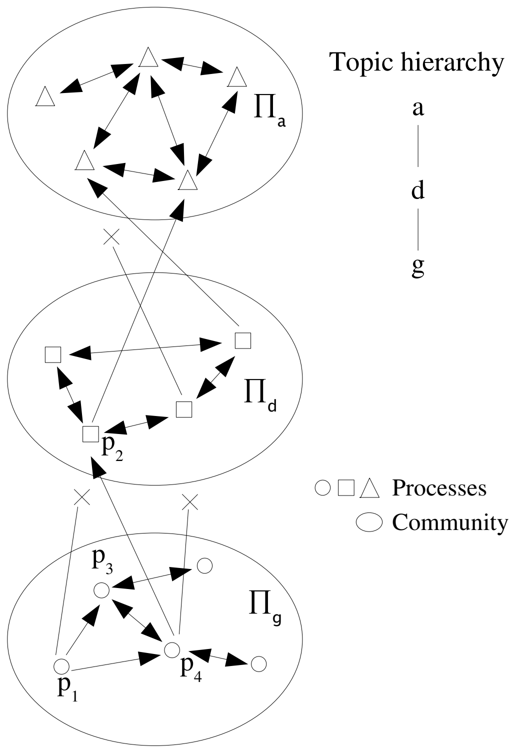

Fig. 2. Dissemination in *CAMCAST*

the direct super-topic of the topic of interest. We will come back later to this issue.

Super Topic Table

$p_2 \in \Pi_{super(t)}$	$p_6 \in \Pi_{super(t)}$
$p_9 \in \Pi_{super(t)}$	

Topic Table

$p_1 \in \Pi_t$	$p_7 \in \Pi_t$
$p_3 \in \Pi_t$	$p_8 \in \Pi_t$
$p_4 \in \Pi_t$	

Fig. 3. Super-topic and topic table for a process interested in topic t

3.2.2 Linking Communities

If a process p_i in a community Π_t receives an event, p_i is responsible for disseminating this event to other processes of that community. The events are also disseminated to the processes participating in $\Pi_{super(t)}$.

An important question here is *how to establish the link* between communities Π_t and $\Pi_{super(t)}$; roughly, two communities, Π_t and $\Pi_{super(t)}$, are said to *be linked* if there is at least one process in Π_t that can send a message to a member of community $\Pi_{super(t)}$. This problem can be separated into two sub-problems: (1) creating links between the different communities when initializing the super-topic tables (i.e., the bootstrapping, see below) and (2) maintaining the information of the super-topic tables consistent (i.e., keeping the communities in touch, see below).

Because any process p_i has to maintain membership information of only its direct super-topic, the new subtopics can be added dynamically into the system in a completely transparent manner for p_i (p_i does not have to maintain any membership information about processes interested in subtopics to receive events from them). Moreover, the processes have to take care of only two membership tables for each topic t of interest, irrespective of the total number of the subtopics of t. This allows dynamic changes of the topic hierarchy. As we pointed out in the introduction, if the topic hierarchy contains only one topic, *CAMCAST* does not use (1) the initialization and (2) the maintenance algorithms and hence simply falls back into the underlying membership algorithm with no degradation.

3.3 Bootstrapping

If a process that wants to join the system (see Figure 5 for the pseudo-code) is provided with contacts belonging to the community $\Pi_{super(t)}$, then the link is directly established (lines 5–8, Figure 5). This bootstrap mechanism is however not always feasible in dynamic systems. The second possibility is for the process to ask other processes (i.e., *neighborhood(p_i)*),[3] via an initialization message specifying the topic of interest, about processes that are participating in $\Pi_{super(t)}$, and so on recursively until a (or a set of) process(es) participating in $\Pi_{super(t)}$ is (are) found.[4] This is done via a task (FIND_SUPER_CONTACT, lines 14–28, Figure 5). As soon as a process is found, the super-topic table is initialized and the FIND_SUPER_CONTACT task can be stopped (lines 31–32, Figure 5).

If no such process exists[5], a new initialization message is sent specifying two topics of interests: *super(t)* and *super(super(t))*. Recursively, if no process participating in either $\Pi_{super(t)}$ or $\Pi_{super(super(t))}$ is found, after a timeout period associated with the message, the scope of the search is enlarged further by adding, to the initialization message, the super-topic of the previous topic of interest, and so on until the root topic is contained in the initialization message (lines 19–27, Figure 5). Note that the reason why we do not include, in the initialization message, all the topics of interest, up to the root topic, is to prevent processes interested in the root topic to reply to all initialization messages. This would basically corresponds to undesirable situation where a community is created for the *publishers of a topic* (see

[3] For instance, *neighborhood(p_i)* can contain processes that are in the same local area-network than process p_i (i.e., that can be reached by p_i via IP multicast messages).
[4] Note that the initialization message can contain a time to live indication (TTL) to not flood the network.
[5] The fact that no process is participating in $\Pi_{super(t)}$ does not imply that no process is participating in $\Pi_{super(super(t))}$ or any of its super-communities (in some sense, there is a lack of participating processes at a specific topic in the topic hierarchy)

Section 3.1).

As soon as a process p_i interested in one of the topics specified in the initialization message send by a process p_j is found, the super-topic table of p_j is initialized with p_i. A joining process p_j keeps on sending initialization messages until it finds such a process p_i . However, it may happen that the process p_i found is not participating in $\Pi_{super(t)}$ but instead in a super-community of $\Pi_{super(t)}$. In this case, p_j participating in Π_t keeps searching for processes participating in $\Pi_{super(t)}$ (line 34, Figure 5). Figure 4 depicts the bootstrapping protocol.

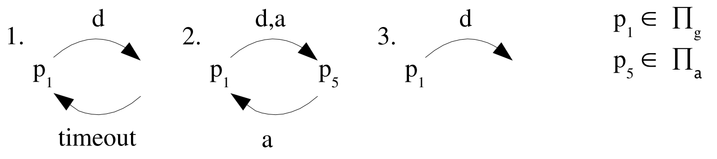

Fig. 4. Bootstrapping

As soon as p_j has an initialized super-topic table, p_j disseminates its super-topic table using the updates of the underlying membership algorithm, to the other processes of the community. When a process p_k receives a message containing a super-topic table, p_k merges that information with its own super-topic table (lines 6–9, Figure 6, the MERGE function consists in replacing the failed processes with the fresh ones obtained from the membership service). This bootstrapping technique relies here only on a weakly consistent global membership. A strongly consistent overlay network (e.g., [15,17]) would also make it easier to find processes participating in a specific topic (see Section 5.2).

3.4 Keeping Communities in Touch

We discuss now how to maintain the super-topic table of a process p_i (see Figure 6 for the pseudo-code), especially when processes, whose IDs are stored in the super-topic table of p_i, crash or leave the community. In this case the super-topic table of p_i is outdated and it is not anymore possible for p_i to propagate events to its super-community. For that purpose, each process, with a probability p_t^{sel} (*sel* in p_t^{sel} stands for *selected*, see below for a precise definition of this probability), tries to find out if the processes in its super-topic table are alive (lines 16–23, Figure 6, the CHECK function consists in returning the total number of processes that are alive in the super-topic table). The detection of alive processes relies on timeouts. If the number of super-processes that are alive is smaller than a certain threshold τ ($0 \leq \tau \leq z_t$), then the process asks all alive processes in its super-topic table to provide it with information (identifiers) about $z_t - \tau$ "new" processes participating in the super-community (lines 19–21, Figure 6). This information is then disseminated using the underlying membership algorithm. A pro-active protocol is used to avoid restarting the bootstrapping protocol if we detect that no super-process is available when an

```
Executed by all $p_i \in \Pi_t$

1: initMsg = []
2: {Done only the first time the message is received}
3: {$u$ super-topic of $t$}
4: upon RECEIVE(REQCONTACT,$p_i$,initMsg) by $p_m \in \Pi_u$ from $p_i$ do
5:     if $\Psi^u_m \neq \emptyset$ then
6:         SEND(ANSCONTACT,$\Psi^u_m$) to $p_i$;
7:         RETURN;
8:     end if
9:     {We try to find a contact up to a certain timeout}
10:    if initMsg has not expired then
11:        SEND(REQCONTACT,$p_i$,initMsg) to neighborhood($p_m$);
12:    end if
13: end upon

14: {Executed each time a timeout occurs, if started}
15: task FIND_SUPER_CONTACT
16:     {A contact is known}
17:     if contact known then
18:         add the contact to $sTable^t_i$;
19:     else
20:         {done at the first time}
21:         if initMsg = [] then
22:             initMsg[0] = $t$;
23:         else
24:             add $super(initMsg[initMsg.length])$ to initMsg;
25:         end if
26:         SEND(REQCONTACT,$p_i$,initMsg) to neighborhood($p_i$);
27:     end if
28: end

29: {$u$ super-topic of $t$}
30: upon RECEIVE(ANSCONTACT,$\Psi^u_m$) by $p_i$ from $p_m \in \Pi_u$ do
31:     if $u == t$ then
32:         stop FIND_SUPER_CONTACT;
33:     else
34:         remove all $v$ in initMsg that include $u$;
35:     end if
36:     $sTable^t_i$ = MERGE($sTable^t_i$,$\Psi^u_m$);
37: end upon
```

Fig. 5. Initialization algorithm used to find processes interested in the super-topics of a topic.

event is published. The use of a reactive protocol would have implied a bigger dependency between the propagation of an event and the availability of the process responsible for that propagation (as the bootstrapping protocol can take some time). For the sake of reliability and load-balancing, it is also possible to replace super-processes in the super-topic table even if those are available.

3.5 Dissemination

Once the bootstrap achieved, a process p_i willing to disseminate an event of topic t proceeds as follows: the event is disseminated (1) to the processes of p_i's super-topic table and (2) to selected processes of its topic table. The super-process dis-

```
Executed by all $p_i \in \Pi_t$

1: {$u$ super-topic of $t$}
2: upon RECEIVE(NEWPROCESS,$p_i$) by $p_m \in \Pi_u$ from $p_i$ do
3:     {The super-process sends a set of available super-processes to $p_i$}
4:     SEND(NEWPROCESS,$\Psi^u_m$) to $p_i$;
5: end upon

6: upon RECEIVE(NEWPROCESS,$\Psi^u_m$) by $p_i$ from $p_m \in \Pi_u$ do
7:     {The super-topic table is updated}
8:     $sTable^t_i$ = MERGE($sTable^t_i$,$\Psi^u_m$);
9: end upon

10: {Executed periodically}
11: task KEEP_TABLE_UPDATED
12:     if $sTable^t_i == \emptyset$ then
13:         start FIND_SUPER_CONTACT;
14:     else
15:         {Test for some processes if their super-processes are up}
16:         if RAND() $\geq p^{sel}_t$ then
17:             {If the total number of processes up is below a certain
                threshold, we send a message to the super-processes that are
                up to receive new fresh membership information}
18:             if CHECK($sTable^t_i$) $\leq \tau$ then
19:                 for all $p_y$ that are up $\in sTable^t_i$ do
20:                     SEND(NEWPROCESS,$p_i$) to $p_y$;
21:                 end for
22:             end if
23:         end if
24:     end if
25: end
```

Fig. 6. Algorithm used to maintain the link between a community Π_t and a community Π_u, where u is a super-topic of t

semination (1) can be summarized as follows: with a probability $p^{sel}_t = \frac{g_t}{N_t}$[6], a process takes part in the dissemination of the event to its super-community (line 3, Figure 8). If a process decides to act as link for a given event, the process sends the event to each of the processes of its super-topic table with probability $p^a_t = \frac{a_t}{z_t}$[7]. The parameter a_t is set according to the average probability of successful transmission. The dissemination of events within a community (2) goes as follows: the process sends the event to $ln\,(N_t)+c_t$ processes, randomly selected in its topic table (lines 9–14, Figure 8). When receiving a new event for the first time, every process (of either the super-community or the community in which the event was initially published) forwards once the event using the dissemination algorithm (lines 5–10, Figure 7).

[6] $1 \leq g_t \leq N_t$, where g_t represents the number of processes that try to contact processes that are in the super-topic table of p_i.

[7] $1 \leq a_t \leq z_t$, where a_t determines the number of processes in the super-topic table that receive the event, lines 4–6, Figure 8

```
Executed by all $p_i \in \Pi_t$

1: function SUBSCRIBE($t$) by $p_i$
2:     Start membership algorithm for $t$ if not already done;
3:     Start maintain links algorithm for $super(t)$ if not already done;
4: end
```

```
Executed by all $p_i$

5: function RECEIVE($E_j^t$) by $p_i$
6:     if $E_j^t$ not received then
7:         DISSEMINATE($E_j^t$);
8:         DELIVER($E_j^t$);
9:     end if
10: end
```

Fig. 7. Subscription/Reception algorithm

```
Executed by all $p_i \in \Pi_t$

1: function DISSEMINATE($E_j^t$) by $p_i$
2:     SUBSCRIBE($t$);
3:     if RAND()$\geq p_t^{sel}$ then
4:         for all $p_y \in sTable_i^t$ do
5:             with probability $p_t^a$ SEND($E_j^t$) to $p_y$;
6:         end for
7:     end if
8:     $\Omega = \emptyset$;
9:     for (j=0;j$\leq$ln($N_t$)+$c_t$;j++) do
10:        {Send the message randomly to processes in our group}
11:        select randomly a process $p_y \in Table_i^t - \Omega$;
12:        SEND($E_j^t$) to $p_y$;
13:        add $p_y$ to $\Omega$;
14:    end for
15: end
```

Fig. 8. Dissemination algorithm

3.6 Subscribing to Unrelated Topics and Dealing with Multiple Inheritance

In the case where a process p_i wants to subscribe to multiple topics that do not include each other or must deal with a multiple inheritance topic hierarchy, p_i has obviously to take care of multiple topic and super-topic tables and one topic and multiple super-topic tables respectivelly. This is also the case with the alternatives presented in Section 3.1 and this is not a drawback induced by our *CAMCAST* algorithm. However, as we will see in Section 4, p_i needs to take care of only two membership tables per topic t p_i is interested in, no matter the number of subtopics of t.

4 Analysis and Comparisons

We discuss here the scalability of our *CAMCAST* algorithm with respect to message and memory complexity as well as reliability and latency. We furthermore compare each of these characteristics with three alternative approaches used to disseminate events in unstructured systems: (a) a gossip-based broadcast, (b) a gossip-based multicast and (c) a hierarchical gossip-based broadcast. For the sake of fairness, all approaches use the same membership technique (i.e., that of [24]).

In short, *CAMCAST* is better in terms of memory complexity than approaches (b) and (c) and sometimes also than approach (a) without hampering the message and latency complexities and without receiving any parasite events (this is not the case with approach (a)). We also show that it is possible for *CAMCAST* to achieve the same reliability than the alternative approaches in tuning its internal parameters (p_t^{sel}, p_t^a and z_t).

4.1 Assumptions

For clarity of presentation, we denote in this section a topic by t_i instead of simply t (here, $i \in [0, x-1]$, where $x \in \mathbb{N}^*$). The topic t_i has a super-topic *super(t_i)* which is denoted by t_{i-1}. This super-topic has itself a super-topic *super(super(t_i))* which is denoted by t_{i-2}, and so on recursively until the root topic t_0. The depth of the topic hierarchy *dpt(S)* is denoted by x. We assume in the analysis that each community corresponding to a topic contains at least one process. This is required for measuring message complexity, as well as reliability and latency. Finally, we consider the case where the topics induce each other.

4.2 Alternative Approaches

In the gossip-based broadcast approach (a), each time an event is sent, it is broadcast in the entire system. This uses membership tables of size $ln\,n + c$, as explained in [27]. In the gossip-based multicast approach (b), the process has one membership table for every topic of interest (this is the approach where a community is created for the publishers of a topic, see Section 3.1). This approach is used in several algorithms (e.g., [12,9,10]), which do not exploit the topic inclusion relationships. Finally, the hierarchical gossip-based broadcast approach (c), presented in [24], creates subcommunities (that do not depend on the interests of the processes in each community) and connects these subcommunities to reduce the overall memory complexity. The system is split into two levels. The first level contains communities of processes that exchange events between them (intra-communities events). The second level is responsible for propagating the events between the communities.

4.3 Message Complexity

4.3.1 Analysis

In the ideal case (according to [27], cf. also [24]), all processes in a community Π_{t_i} receive every event that is published in Π_{t_i}. The overall number of events sent in the community Π_{t_i} is upper bound by: $N_{t_i} \cdot (ln\, N_{t_i} + c_{t_i})$, as each process sends $ln\, N_{t_i} + c_{t_i}$ events. Furthermore, several processes in Π_{t_i} additionally disseminate the events to the processes in $\Pi_{t_{i-1}}$. This number of *upward messages* is equal to: $nbUpwardMsg_{t_i} = N_{t_i} \cdot p^{sel}_{t_i} \cdot p^{a}_{t_i} \cdot z_{t_i} \cdot p^{succ}_{t_i}$.

This corresponds to the average sum of events sent by the processes of Π_{t_i} (N_{t_i}), which have decided to act as links ($p^{sel}_{t_i}$), to the processes chosen ($p^{a}_{t_i}$) within those from the super-community (z_{t_i}) and effectively received ($p^{succ}_{t_i}$). This probability depends on the availability of the processes as well as on the reliability of the links. For the sake of generality, we make this probability depend on the topic to simulate weakly interconnected communities.

Hence, the total number of events sent from the community Π_{t_i}, all the way up to the root community, is:

$$\sum_{i=x-1}^{0} (N_{t_i} \cdot (ln\, N_{t_i} + c_{t_i})) + \sum_{i=x-2}^{0} (N_{t_i} \cdot p^{sel}_{t_i} \cdot p^{a}_{t_i} \cdot p^{succ}_{t_i} \cdot z_{t_i}) \tag{1}$$

The two sums follow from the fact that the processes participating in the root community do not need to disseminate events to any higher level. In the worst case (in terms of message complexity), Equation 1 is upper bound by ($\forall t_i$):
$z_{max} = max(z_{t_i})$, $N^{max}_{t_i} = max(N_{t_i})$, $c_{max} = max(c_{t_i})$ (where *max()* outputs the maximum of the values given in parameter) and by $p^{sel}_{t_i} = p^{a}_{t_i} = p^{succ}_{t_i} = 1$. Furthermore, as $(N^{max}_{t_i}) > 1$, Equation 1 is upper bound by $ln\, N^{max}_{t_i}$:

$$maxNbMsgSent \leq x \cdot N^{max}_{t_i} \cdot (ln\, N^{max}_{t_i} + c_{max}) + x \cdot N^{max}_{t_i} \cdot ln\, N^{max}_{t_i} \cdot z_{max} \tag{2}$$

$$(2) \leq x \cdot N^{max}_{t_i} \cdot ln\, N^{max}_{t_i} \cdot (1 + c_{max} + z_{max}) \tag{3}$$

Finally, if x can be upper bound by a constant, we have: *maxNbMsgSent* $\in O(N^{max}_{t_i} \cdot ln\, N^{max}_{t_i})$.

Of course, this holds iff x is constant (otherwise *maxNbMsgSent* $\in O(x \cdot N_{t_i}^{max} \cdot ln\, N_{t_i}^{max})$), a common hypothesis which is not limiting when considering real-world applications according to [28]. [8]

4.3.2 Comparisons

As described above, the message complexity of *CAMCAST* is $O(N_{t_i}^{max} \cdot ln\, N_{t_i}^{max})$. The other approaches perform as follows:

(a) Gossip-based broadcast. The total membership information of each process is: *totalMbInfo* $= (c + ln n)$. This means that the total number of events sent is: *nbMsgSent* $= n \cdot (c + ln n)$. As $n > 1$ and as c is a constant, we have: *maxNbMsgSent* $\in O(n \cdot ln n)$.

(b) Gossip-based multicast. For a process participating in the root community, the total membership information is: *totalMbInfo* $= \sum_{i=x-1}^{0}(c_{t_i} + ln\, N_{t_i})$. If a process wants to publish an event, the total number of disseminated events is:

$$\textit{nbMsgSent} = \sum_{i=x-1}^{0} N_{t_i}(c_{t_i} + ln\, N_{t_i}) \tag{4}$$

If we upper bound equation (4) by $N_{t_i}^{max}$ and c_{max}, we have: (4) $\leq x \cdot N_{t_i}^{max} \cdot (c_{max} + ln\, N_{t_i}^{max})$. As $N_{t_i}^{max} > 1$, c_{max} is a constant and x can be upper bound by a constant, we have: *maxNbMsgSent* $\in O(N_{t_i}^{max} \cdot ln\, N_{t_i}^{max})$.

(c) Hierarchical gossip-based broadcast (c) The total membership information of each process is given by (where C represents the total number of communities and m the number of processes inside a community): *totalMbInfo* $= c_1 + c_2 + lnC + lnm$. The total number of events sent in the system is:

$$\textit{nbMsgSent} = C \cdot m(c_1 + c_2 + lnC + lnm) \tag{5}$$

If we upper bound equation (5) by $N_{t_i}^{max}$, we have: *nbMsgSent* $\leq C \cdot N_{t_i}^{max}(c_1 + c_2 + lnC + ln\, N_{t_i}^{max})$. As c_1, c_2 are constants, then if we upper bound C by a constant (same assumptions as the one for x), we have: *maxNbMsgSent* $\in O(N_{t_i}^{max} \cdot ln\, N_{t_i}^{max})$. Thus, in short, applying *CAMCAST* on a membership algorithm does not hamper its overall message complexity performance.

[8] Note that we consider here the message complexity and not the actual value of the total number of messages sent (this last value depends on x).

4.4 Memory Complexity

4.4.1 Analysis

In the model we consider, topics include one another and each process participating in a community maintains two tables: (1) a topic table and (2) a super-topic table. The only exception is for the processes participating in the root community: these only maintain the topic table.

The size of the topic table depends logarithmically on the number of processes participating in the community. The super-topic table is of size z_{t_i}, which is constant. Hence, the memory complexity of every process is: $ln\, N_{t_i} + c_{t_i} \leq$ *totalMbInfo* $\leq ln\, N_{t_i} + c_{t_i} + z_{t_i}$.

Note that this complexity depends neither on the number of super-topics of a topic of interest, nor on the number of its subtopics.

4.4.2 Comparisons

As described above, the maximal number of membership tables in *CAMCAST* is 2 (1 if the process is participating in the root community). This number does not depend on the number of communities a process is participating in, when these include one another.

(a) Gossip-based broadcast. An event is disseminated to all the processes in the system. Thus every process has one membership table only and its memory complexity is: $ln\, n + c$, where $n = (\sum_{i=x-1}^{0} N_{t_i}^{max}) \gg N_{t_i}^{max}$.

(b) Gossip-based multicast. Every process maintains a membership table for each community it is participating in (of size: $ln\, N_{t_i} + c_{t_i}$). With a maximum of x levels in a topic hierarchy, and assuming that each subtopic has exactly one super-topic (except the root), a process deals with at most x tables. The total memory complexity of each process is: $\sum_{i=x-1}^{j}(ln\, N_{t_i} + c_{t_i})$.

(c) Hierarchical gossip-based broadcast. Each process maintains two membership tables. The first table is used when disseminating events to the processes that are randomly selected to "represent" their community, and a second membership table to disseminate events in the community itself. The first table has a size of $ln\, C + c_2$ and the second table has a size of $ln\, m + c_1$, where C represents the total number of communities and m the number of processes inside a community. So each process has a memory complexity of: $ln\, m + ln\, C + c_1 + c_2$.

The number of tables in *CAMCAST* is just majored by one, with respect to (a), and this can be neglected given the huge gain obtained with *CAMCAST* by avoiding any parasite events (see Section 6). As discussed in Section 3.6, when a process subscribes to k unrelated topics, it has to deal with $2k$ topic and super-topic tables (this is also the case with the traditional gossip-based multicast approach but not with the broadcast ones). There is a trade-off between the number of topic tables to store and the number of parasite events: the more topics a process is interested in, the less parasite events it receives. This trade-off is determined by the topic hierarchy as well as the interests of the processes. Nevertheless, as discussed in Section 3.6, it is likely that the drawback of subscribing to multiple topics can be overcome by subscribing to one common super-topic of those topics.

Finally, the memory complexity for a process in community Π_{t_i} is $ln\, N_{t_i}+c_{t_i}+z_{t_i}$ in *CAMCAST*. This means that the memory complexity of a process is always smaller in our algorithm than in approaches (b) and (c) and sometimes even in approach (a).

4.5 Reliability

4.5.1 Analysis

By reliability, we mean here the probability that *every* process interested in topic t_i receives a given event published for t_i. The results of [24] imply that, if all processes interested in the same topic t_i disseminate an event to $ln\, N_{t_i} + c_{t_i}$ processes, then the probability that every process interested in t_i receives the event is $e^{-e^{-c_{t_i}}}$. The worst case is when the events are disseminated in all levels of the topic hierarchy (i.e., in the x levels). This occurs when an event is of the bottom-most topic and has to be disseminated up to the processes interested in the root topic. This is the worst case because it sums the passing between topics and super-topics over the established links.

The number of processes *susceptible* to send an event from one community Π_{t_i} to its super-community is defined by: $\textit{nbSuscProc}_{t_i} = N_{t_i} \cdot p_{t_i}^{sel} \cdot \pi_{t_i}$.

Where π_{t_i} is the proportion of processes that actually receive the event through the underlying membership algorithm for a community Π_{t_i} (cf. [13]) and hence are able to propagate the event to $\Pi_{t_{i-1}}$. The probability that no event is received by a member of $\Pi_{t_{i-1}}$ is consequently defined by: $\textit{pbNoUpwardMsg}_{t_i} = (1 - p_{t_i}^{succ})^{\textit{nbSuscProc}_{t_i} \cdot p_{t_i}^{a} \cdot z_{t_i}}$.

Where p_t^{succ} is the probability that an event sent from one community of processes is received in the super-community (for the definition of the other values, we refer to Section 3). The probability of the propagation of the message to a super-community is then defined as:

$$pit_{t_i} = 1 - pbNoUpwardMsg_{t_i} \tag{6}$$

With Equation 6, the probability that all processes belonging to a community Π_{t_j} receive the event is: *reliability* $= \prod_{i=x-1}^{j}(e^{-e^{-c_{t_i}}} \cdot pit_{t_i})$.

The first term of the reliability equation (i.e., $e^{-e^{-c_{t_i}}}$) comes from the gossiping technique we use (i.e., [24]) and determines the reliability of the dissemination of an event of topic t_i in the community Π_{t_i}. We can tune c_{t_i} to trade the reliability of the dissemination in the community Π_{t_i} and the total number of messages sent in this community. The second term of the reliability equation (i.e., pit_{t_i}) comes from the specificity of *CAMCAST* (i.e., "community-awareness"). We can also tune this parameter (via $p_{t_i}^{sel}$, $p_{t_i}^{a}$ and z_{t_i}) dynamically to trade the number of messages sent between a community Π_{t_i} and its super-community. This tuning might turn out to be important in dynamic systems where the number of processes are constantly changing. [9] If the number of processes in a community becomes very small, we make all processes to propagate the events to their super-community.

4.5.2 Comparisons

As described above, the reliability of *CAMCAST* is: $\prod_{i=x-1}^{j}(e^{-e^{-c_{t_i}}} \cdot pit_{t_i})$.

With this respect, the other approaches perform as follows:

(a) Gossip-based broadcast. With the memory complexity presented in Section 4.4.2, the reliability is: $e^{-e^{-c}}$

(b) Gossip-based multicast. With the memory complexity presented in Section 4.4.2, the reliability is: $\prod_{i=x-1}^{j} e^{-e^{-c_{t_i}}}$.

(c) Hierarchical gossip-based broadcast. As explained in Section 4.4.2, the reliability is (see [24] for a complete analysis): $e^{-Ce^{-c_1}-e^{-c_2}}$.

The probability that *every* processes receive an event is thus smaller with *CAMCAST* than with other algorithms in the general case, especially for the processes interested in the root topic. [10] This is because, in *CAMCAST*, the reliability depends on the event propagation between communities. However, it is possible to tune this and achieve, in specific cases, the same reliability as other algorithms. For the sake

[9] For example, if the number of processes is growing in a community, we can reduce pit_{t_i} to reduce the total number of inter-community messages sent but without hampering the reliability (as there are a lot of processes).

[10] If we have considered the *average* number of processes that receive an event, the result would be much more in our favor (because we would have made an average over the reliability of each community, instead of a multiplication).

of simplicity, we consider in the following of this analysis the average case, where, for every t_i, z_{t_i} is z, N_{t_i} is N_t and pit_{t_i} is pit:

(a) Gossip-based broadcast. CAMCAST achieves the same reliability as (a) when $0 \leq c \leq -ln\,(-x{\cdot}ln\,pit)$. Here c denotes the constant used to determine the number of processes to disseminate events to in the gossip-based broadcast algorithm (e.g., $ln\,n + c$). In this case, the memory complexity of *CAMCAST* is smaller iff: $z \leq ln\,n + ln\,(1 + x \cdot e^c \cdot ln\,pit) - ln\,N_t - ln\,x$.

Proof. For *CAMCAST* to achieve the same reliability as gossip-based broadcast, we must have:

$$\prod_{i=x-1}^{j} (e^{-e^{-c_{1_{t_i}}}} \cdot pit_{t_i}) = e^{-e^{-c}} \tag{7}$$

In the worst case, $j = 0$, because *CAMCAST* has the worst performance when a process is interested in the top most topic. From (7) it follows that:

$$e^{\sum_{i=x-1}^{0} -e^{-c_{1_{x_i}}}} \prod_{i=x-1}^{0} pit_{t_i} = e^{-e^{-c}} \Leftrightarrow \sum_{i=x-1}^{0} e^{-c_{1_{t_i}}} - ln \prod_{i=x-1}^{0} pit_{t_i} = e^{-c} \tag{8}$$

If we assume (for simplification purpose) that all $c_{1_{t_i}}$ are equal to c_1, and all pit_{t_i} are equals to pit, we have then (8) $\Leftrightarrow$

$$e^{-c_1} - ln\,pit = \tfrac{e^{-c}}{x} \overset{(\textcircled{1})}{\Leftrightarrow} ln\,e^{-c_1} = ln\,\tfrac{e^{-c}+xln\,pit}{x}$$

$$\overset{(\textcircled{2})}{\Leftrightarrow}$$

$$c_1 = ln\,\tfrac{1}{e^{-c}+xln\,(pit)} + ln\,x \Leftrightarrow c_1 = c - ln\,(1 + xe^c ln\,pit) + ln\,x$$

With the following conditions:

- $\textcircled{1} \Leftrightarrow \frac{e^{-c}+xln\,pit}{x} > 0$ and as $x \geq 1$, thus, $e^{-c}+xln\,pit > 0 \Leftrightarrow c < -ln\,(-xln\,pit)$.
- $\textcircled{2}$ We want $c_1 \geq 0$, this means that $ln\,\frac{e^{-c}+xln\,pit}{x} \leq 0 \Leftrightarrow e^{-c} + xln\,pit \leq x \overset{(\textcircled{3})}{\Leftrightarrow} c \geq -ln\,(x(1 - ln\,pit))$. As $1 - ln\,pit > 0$ and as $x \geq 1$ (see $\textcircled{3}$), condition $\textcircled{2}$ implies $-ln\,(x(1{-}ln\,pit)) \leq 0$. This means that $c \geq 0$ encompasses $c \geq -ln\,(x(1 - ln\,pit))$.
- $\textcircled{3}$ This equation holds only if $t(1 - ln\,pit) > 0$ which is always the case, as $x \geq 1$ and as $0 \leq pit \leq 1$.

From $\textcircled{1}$, $\textcircled{2}$ and $\textcircled{3}$ we can setup c_1 with respect to c if and only if $0 \leq c \leq -ln(-xlnpit)$. If $c < 0$ or if $c > -ln(-xlnpit)$, we cannot set c_1 in *CAMCAST* to have the same reliability as the gossip-based broadcast. This means that if c does not satisfy the previous property (i.e., $c < 0$ or $c > -ln(-xlnpit)$), no matter what is the size of the super-topic table in *CAMCAST*, it is not possible to achieve the same reliability. However, if c satisfies the property, we can replace c_1 with its value depending in c in the $ln N_{t_i} + c_1 + z_{t_i}$ equation (remember that all $c_{1_{t_i}}$ are equal to c_1) and this leads to satisfying the following inequality (to simplify, we assume that all $N_{t_i} = N_t$, that all $z_{t_i} = z$, which corresponds to the average case):

$$ln N_t + c - ln(1 + xe^c lnpit) + ln x + z \overset{?}{\leq} ln n + c$$

The total membership information of *CAMCAST* is smaller then the total membership information of the gossip-based broadcast algorithm iff:

$$z \leq ln n + ln(1 + xe^c lnpit) - ln N_t - ln x$$

In short, this means that $ln(1 + xe^c lnpit)$ must tend to its maximum (which is 0) and $ln n > ln N_t + ln x$.

□

(b) Gossip-based multicast. CAMCAST achieves the same reliability as (b) when $0 \leq c \leq -ln(-lnpit)$. Here, c denotes the constant used to determine the number of processes to disseminate events to in the gossip-based multicast algorithm (e.g., $ln N_{t_i} + c_{t_i}$, where all c_{t_i} are the same and equal to c). In this case, the memory complexity of *CAMCAST* is smaller iff: $z \leq (x-1)\cdot(ln N_t + c) + ln(1 + e^c \cdot lnpit)$.

Proof.

For *CAMCAST* to achieve the same reliability as (b), we must have:

$$\prod_{i=x-1}^{j} (e^{-e^{-c_{1_{x_i}}}} \cdot pit_{t_i}) = \prod_{i=x-1}^{j} e^{-e^{-c_{2_{t_i}}}} \tag{9}$$

And in the worst case $j = 0$, because *CAMCAST* has the worst performance when a process is interested in the top most topic. This means that (9) $\Leftrightarrow$

$$e^{\sum_{i=x-1}^{0} -e^{-c_{1_{x_i}}}} \prod_{i=x-1}^{0} pit_{t_i} = e^{\sum_{i=x-1}^{0} -e^{-c_{2_{t_i}}}}$$

$$\Leftrightarrow$$

$$\sum_{i=x-1}^{0} e^{-c_{1_{t_i}}} - ln \prod_{i=x-1}^{0} pit_{t_i} = \sum_{i=x-1}^{0} e^{-c_{2_{t_i}}} \tag{10}$$

If we assume (for simplification) that all the $c_{1_{t_i}}$ are equal to c_1, all $c_{2_{t_i}}$ are equal to c, and all pit_{t_i} are equal to pit, we have then (10) $\Leftrightarrow$

$$e^{-c_1} - ln\,pit = e^{-c} \overset{(\textcircled{1}\textcircled{2})}{\Leftrightarrow} ln\,e^{-c_1} = ln\,(e^{-c} + ln\,pit) \tag{11}$$

$$\Leftrightarrow$$

$$c_1 = c - ln\,(1 + e^c ln\,pit)$$

With the following conditions:

- ① $\Leftrightarrow$ $e^{-c} + ln\,pit > 0$, otherwise the equivalence in (11) does not hold. This means that $e^{-c} > -ln\,pit \Leftrightarrow c < -ln\,(-ln\,pit)$. We can have the equivalence here between the two terms because $ln\,pit$ is always less then 0 as pit is always less or equal to 1 (remember that pit is a probability; in ③ we show what happens when pit is equal to 1).
- ② c_1 must be greater or equal to 0, this means that $ln\,(e^{-c} + ln\,pit) \leq 0 \Leftrightarrow e^{-c} \leq 1 - ln\,pit \Leftrightarrow c \geq -ln(1 - ln\,pit)$ (we can have the equivalence if $1 - ln\,pit > 0$ which is always the case, because $e > pit$). But $1 - ln\,pit \geq 1$ as $0 \leq pit \leq 1$, which means that $-ln\,(1 - ln\,pit) \leq 0$. In other terms, this means that the condition $c \geq 0$ encompasses the condition $c \geq -ln\,(1 - ln\,pit)$.
- ③ In the case of $pit = 1$ this means, according to (9) that $c_1 == c$.

From ①, ② and ③, it is clear that c_1 can be set with respect to c iff $0 \leq c \leq -ln\,(-ln\,pit)$. If $c < 0$ or if $c > -ln\,(-ln\,pit)$, we cannot set c_1 in *CAMCAST* to have the same reliability as in the gossip-based multicast algorithm. This means that if c does not satisfy the previous property, no matter the size of the super-topic table of *CAMCAST*, then it is not possible to achieve the same reliability. However, if c satisfies the property, we can replace c_1 in the $ln\,N_{t_i} + c_1 + z_{t_i}$ equation (remember that all $c_{1_{t_i}}$ are equal to c_1) and this leads to (to simplify we assume that all $N_{t_i} = N_t$, that all $z_{t_i} = z$, which corresponds to the average case):

$$ln\,N_t + c - ln\,(1 + e^c ln\,pit) + z \overset{?}{\leq} \textstyle\sum_{i=x-1}^{0}(ln\,N_t + c)$$

$$\Leftrightarrow$$

$$ln\,N_t + c - ln\,(1 + e^c ln\,pit) + z \overset{?}{\leq} x(ln\,N_t + c)$$

This means that the total membership information of *CAMCAST* is smaller than the total membership information of the gossip-based multicast algorithm iff:

$$z \leq (x-1)(ln\, N_t + c) + ln\,(1 + e^c ln\, pit)$$

Remember that the upper equation holds only for values of z that are greater than 0 (if z is less or equal to 0, no dissemination can be made to the upper levels and hence there is no point in trying to compare the different algorithms).

$\square$

(c) Hierarchical gossip-based broadcast. *CAMCAST* achieves the same reliability as (c) when $-ln\,\frac{x\cdot(1-ln\,pit)}{C+1} \leq c \leq -ln\,\frac{-x\cdot ln\,pit}{C+1}$. Here c denotes the number of processes that disseminate events in the hierarchical algorithm, see [24]. In this case, the memory complexity of *CAMCAST* is smaller iff: $z \leq c + ln C + ln\,(C + 1 + x \cdot e^c \cdot ln\,pit) - ln\,x$.

Proof.

For *CAMCAST* to have the same reliability than in hierarchical gossip-based broadcast algorithm, we must have:

$$\prod_{i=x-1}^{j}(e^{-e^{-c_{1_{t_i}}}} \cdot pit_{t_i}) = e^{-Ce^{-c_1}-e^{-c_2}}$$

In the worst case, $j = 0$, because *CAMCAST* has the worst performance when a process is interested in the top most topic. Moreover if we assume (again for simplicity) that all $c_{1_{t_i}}$ are the same (equal to c_t), all pit_{t_i} are the same and equal to pit, and that $c_1 = c_2 = c$ we have:

$$e^{\sum_{i=x-1}^{0} -e^{-c_t}} pit^x = e^{-(C+1)e^{-c}} \Leftrightarrow xe^{-c_t} - xln\,pit = (C+1)e^{-c}$$

$$\Leftrightarrow$$

$$e^{-c_t} = \frac{(C+1)e^{-c} + xln\,pit}{x} \overset{(\textcircled{1})}{\Leftrightarrow} c_t = -ln\,\frac{(C+1)e^{-c} + xln\,pit}{x}$$

$$\overset{(\textcircled{2})}{\Leftrightarrow}$$

$$c_t = ln\,x + c - ln\,(xe^c ln\,pit + C + 1)$$

With the following conditions:

- $\textcircled{1} \Leftrightarrow \frac{(C+1)e^{-c}+xln\,pit}{x} > 0$ and as $x \geq 1$, this implies that $(C+1)e^{-c} + xlnpit > 0 \Leftrightarrow c < -ln\,\frac{-xln\,pit}{C+1}$.
- $\textcircled{2}$ We want c_t to be greater or equal to 0, this means that $ln\,\frac{(C+1)e^{-c}+xln\,pit}{x} \leq 0 \Leftrightarrow (C+1)e^{-c} + xlnpit \leq x \Leftrightarrow e^{-c} \leq \frac{x(1-ln\,pit)}{C+1} \Leftrightarrow c \geq -ln\,\frac{x(1-ln\,pit)}{C+1}$ (as x and C are greater then 0 and as $0 \leq pit \leq 1$).

From $\textcircled{1}$ and $\textcircled{2}$, we can setup c_t with respect to c iff $-ln\,\frac{x(1-ln\,pit)}{C+1} \leq c \leq -ln\,\frac{-xln\,pit}{C+1}$. If $c < -ln\,\frac{x(1-ln\,pit)}{C+1}$ or if $c > -ln\,\frac{-xln\,pit}{C+1}$, we cannot set c_t in *CAMCAST* to have the same reliability than in the hierarchical gossip-based broadcast algorithm. However, if c_t satisfies the property, we can replace c_t with its value depending in c in the $ln\,N_{t_i} + c_t + z_{t_i}$ equation (remember that all $c_{1_{t_i}}$ are equal to c_t) and this leads to the following inequality (to simplify we assume that all $N_{t_i} = N_t$, that all $z_{t_i} = z$, which corresponds to the average case):

$$-ln\,(C+1+xe^{c}lnpit) + ln\,x + z \overset{?}{\leq} ln\,C + c$$

In this case, the total membership information of *CAMCAST* is smaller than the total membership information of the hierarchical gossip-based broadcast algorithm iff:

$$z \leq c + ln\,C + ln\,(C+1+xe^{c}lnpit) - ln\,x$$

$\square$

4.6 Latency

4.6.1 Analysis

By latency, we mean here the number of *rounds* needed by our algorithm to propagate an event into the entire system. A round corresponds to the propagation of an event from one set of processes to another without transiting by any other processes. To measure the latency of a specific community, we assume that the event is propagated from one community to its super-community with probability 1 (pit_{t_i}). Of course a message may never reach it's destination, incurring a latency of ∞. This eventuality is however taken into account by our notion of reliability.

According to [29], the number of rounds needed to propagate an event in a community Π_{t_i} of N_{t_i} processes is:

$$R_{N_{t_i}} = \frac{ln\, N_{t_i}}{ln\, ln\, (N_{t_i})} + O(1) \tag{12}$$

Applying Equation 12 to our algorithm, we compute three cases:

Best case: In this case, the event is directly propagated from one community to its super-community. Hence, to propagate an event from community Π_{t_j} to community Π_{t_i} $(i \leq j)$: *latency*$_{min}$ $= (j-i) + R_{N_{t_i}} \leq x-1+R_{N^{max}_{t_i}}$. This implies, *latency*$_{min}$ $\in O(x + R_{N^{max}_{t_i}})$. And if x is a constant, *latency*$_{min}$ $\in O(R_{N^{max}_{t_i}})$.

Worst case: In this case, the event is propagated entirely to one community before being sent to its super-community. This means: *latency*$_{max}$ $= \sum_{k=j}^{i} R_{N_{t_k}} + (j-i) \leq x \cdot R_{N^{max}_{t_i}} + (x-1) \leq x \cdot (R_{N^{max}_{t_i}} + 1)$. This implies, *latency*$_{max}$ $\in O(x \cdot (R_{N^{max}_{t_i}} + 1))$. And if x is a constant, *latency*$_{max}$ $\in O(R_{N^{max}_{t_i}})$.

Average case: In this case, we compute the average latency in which we assume that, after $\frac{R_{N_{t_i}}}{2}$ rounds, the event is propagated to the super-community. This means that: *latency*$_{avg}$ $= \sum_{k=j}^{i} \frac{R_{N_{t_k}}}{2} + (j-i) \leq x \cdot (\frac{R_{N^{max}_{t_i}}}{2} + 1)$. This implies, *latency*$_{avg}$ $\in O(x \cdot (\frac{R_{N^{max}_{t_i}}}{2} + 1))$. And if x is constant, *latency*$_{avg}$ $\in O(R_{N^{max}_{t_i}})$.

4.6.2 Comparisons

The latency is $O(R_{N^{max}_{t_i}})$ for all algorithms except for the gossip-based broadcast which has a latency of $O(R_n)$. This means that *CAMCAST* is equivalent, in terms of latency, to all the other algorithms.

5 Implementation

We report here on a full implementation of a Java 1.5 *CAMCAST* framework. In this framework, we have implemented three variants of *CAMCAST*, each relying on a specific membership service: unstructured, semi-structured and structured. The code of the unstructured version of our implementation is available at: *http://lpd.epfl.ch/baehni/camult.tgz.*

5.1 Application Programming Interfaces

To subscribe to a community or to publish a message in a community, we must first go through an initialization phase.

```
CommunityManager manager = CommunityManager.getDefaultManager();
```

The `CommunityManager` class is responsible for retrieving the different communities that are in the system (either with the help of statically defined gateways in the unstructured version or with the help of an overlay network).

Once the `CommunityManager` has been initialized, it is possible to request the community a process wants to join with the following line of code:

```
ICommunity community = manager.getCommunity(Class.forName("g"));
```

Each community is mapped with a Java interface representing the topic of interest, such that the hierarchy of communities (or topics) corresponds to a hierarchy of Java interfaces. It is then possible, with the `ICommunity` object, to subscribe to the community represented by the previously given Java interface.

```
MySubscriber subscriber = new MySubscriber();
community.subscribe(subscriber);
```

The `subscribe()` method contacts a gateway in order for the process to join the community. The list of available gateways is provided through a configuration file in the unstructured case or with the overlay network in the semi-structured and structured cases. A current limitation of our implementation relies in the fact that at least one gateway must be available to join a specific community. Hence, the case where a community is empty is not allowed in our implementation. The `MySubscriber` class implements the `ICommunitySubscriber` callback interface in order to receive the published events asynchronously through the `onMessage()` method.

The publication of an event is done through the `publish()` method:

```
community.publish(new Message("Hello World"));
```

The first call to the `publish()` method does not return immediately as the process must first join the community it wants to publish an event to. Once the community has been joined, the event is published following the *CAMCAST* pattern.

Finally, to cleanly quit the application, a subscriber has to unsubscribe and a publisher has to leave the community. This is illustrated with the following lines of code:

```
// Unsubscribing
community.unsubscribe(subscriber);

//Stop publishing
community.stopPublishing();
```

5.2 *Architecture*

Our *CAMCAST* framework uses three different membership services: (1) an unstructured membership protocol, (2) a semi-structured membership protocol and (3) a structured membership protocol.

The bootstrapping used in the different implementations is the one where contact nodes are directly available when a subscriber wants to join a community (i.e., this corresponds to the lines 16–18 of Figure 4). As presented below, the unstructured approach is directly provided with contact nodes (or gateways) whereas the semi-structured and the structured approaches use an overlay network to retrieve those contacts.

5.2.1 *The Unstructured Approach*

In this approach, *CAMCAST* uses a gateway mechanism for bootstrapping (see Figure 9). Each community has its own *gateways*; these are highly available processes, known by all other processes participating in the same community. They provide bootstrapping contacts to new processes that want to join the system.

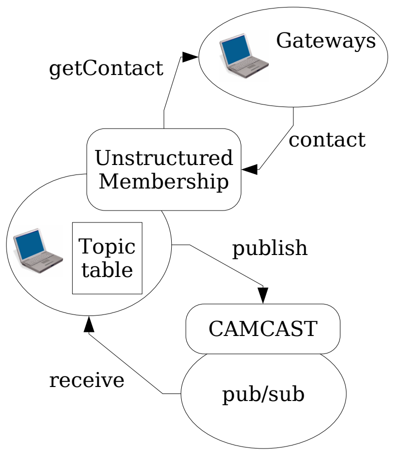

Fig. 9. Unstructured approach

The underlying membership service is the one of [24] and uses the acquired contacts to construct the topic table for each process participating in a community. This table is updated when processes join or leave by the membership algorithm. However, as opposed to our simulations, where the topic table has the ideal size of $ln\,N_t + c_t$, this table has, in our implementations, different sizes averaging $ln\,N_t$ entries (depending on the membership service we use). As we will see in Section 6, this major difference plays an important role in the results we obtain when disseminating an event. The links between a community and its super-community are also maintained in an unstructured way, by exchanging super-process information between the processes of a given community.

5.2.2 *The Semi-Structured Approach*

The semi-structured approach uses the membership algorithm mentioned in [24], but uses an underlying structured overlay network [25] that offers strong search capabilities for the bootstrapping mechanism (see Figure 10).

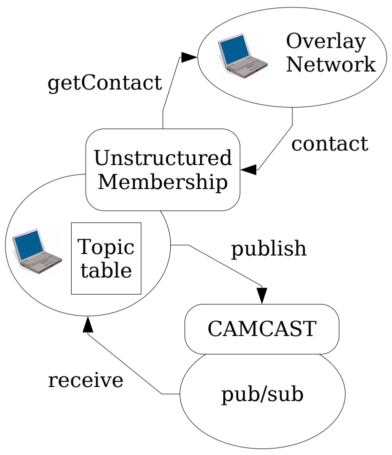

Fig. 10. Semi-structured approach

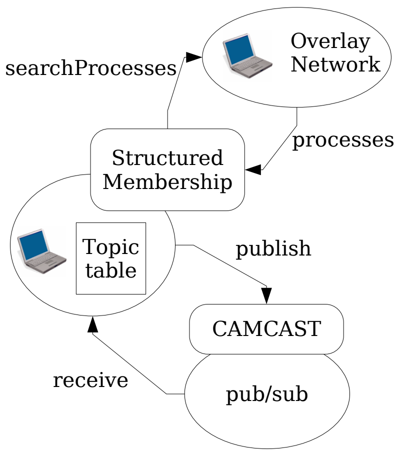

Fig. 11. Structured approach

In this scheme, we use the structured overlay network to retrieve contacts for the super-community (via the `getContact()` method provided by the underlying overlay network). However we do not use the structured overlay to populate the topic table. The membership algorithm used is still the one of [24].

To take advantage of the structured overlay, a process first has to join it, with the help of overlay-specific gateways. But these gateways are not related to any community (as opposed to the gateways required in the unstructured approach) and do not take part in the *CAMCAST* algorithm.

5.2.3 *The Structured Approach*

In the third approach, we take advantage as much as possible of the underlying overlay network in making both the bootstrapping phase and the membership service rely on it (thanks to the `searchPeers()` method of the underlying overlay). This basically means that the structured overlay gives us directly the both the topic and super-topic tables (i.e., we do not make use anymore of the membership algorithm of [24]).

Joins and leaves are reflected in subsequent calls to the membership service, which provides dynamic and uniform random membership tables. As the overlay network gives us the tables, those have all the same size of $ln\, N_t + c_t$. We update the membership tables at a time interval that is correlated to the subscription change frequency. The overlay network also takes care of the inter-community links via search requests for a super-community.

5.3 Design Overview

The implementation of our framework is decomposed in several packages: (1) `p2p`, (2) `pubsub` and (3) `camcast`. The different dependencies between the packages is depicted in Figure 12. To be the most generic as possible, almost each class of the different packages implements an interface. This let the possibility to have another implementation without changing the applications.

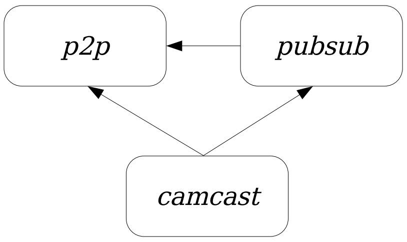

Fig. 12. Dependencies between the packages

5.3.1 P2P Package

This package encapsulates both the membership and communication primitives used between the processes of the system. This package contains different classes we present hereafter.

5.3.1.1 LocalPeer and RemotePeer. A `LocalPeer` is a singleton class and represents a local process. During its initialization, a `MembershipManager` and a
`PipeManager` are created (see below). These are used by the membership algorithm as well as by our *CAMCAST* implementation. A `LocalPeer` is essentially defined through its IP address.

A `RemotePeer` represents a remote process. It is mainly a container for an IP address and a port to which a process can send messages.

5.3.1.2 RemotePeerWrapper, SuperPeerWrapper and JoinPeerWrapper. These classes are wrappers for `RemotePeer` instances. These wrappers are used in the class
`Message` to send or receive information about `RemotePeer` as well as to distinguish the kind of `RemotePeers` we are dealing with: (1) a remote peer of the community, (2) a remote peer of the super-community and (3) a remote peer used to join a community (e.g., a gateway).

5.3.1.3 Message. This class represents a low-level message that is sent between the different processes of the system. A `Message` can be used to transport events (i.e., `pubsub.Event`) as well as different kind of system messages: (1) to join or leave a community, (2) to request processes in a community, (3) to request processes in a super-community and (4) to request gateways (if needed). Each `Message` has (1) an `header` which defines the message, (2) a `community` variable defining in which community the message was sent and (3) `data` containing the data of the message.

5.3.1.4 InputPipe, OutputPipe and PipeManager. The communication between different processes of the system is done through an `InputPipe` and `OutputPipe` instances. An `InputPipe` simply receives instances of `Message` and forwards them to a
`MessageDispatcher`. A process can have one `InputPipe` only.

An `OutputPipe` is used to send messages between two processes. Thus, a process can have multiple instances of `OutputPipe` if it wants to send messages to different processes. The creation of an `OutputPipe` is done via a `RemotePeer` instance and the sending of a `Message` is achieved through the `sendMessage()` method.

The creation of both the `InputPipe` and `OutputPipe` is done via a `PipeManager`. This class contains a method `getInputPipe()` for retrieving the `InputPipe` as well as for creating different instances of `OutputPipe` (i.e., `createOutputPipe()`).

5.3.1.5 MessageDispatcher and IPipeListener. An instance of a `MessageDispatcher` is used to dispatch the `Messages` received from the `InputPipe` to the classes that have implemented the `IPipeListener` interface. This interface has an `onMessage()` method that is called by the `MessageDispatcher` when a new `Message` is received.

The set of classes that are interested to be notified when a new `Message` is received by the dispatcher is given as a parameter of the constructor.

5.3.1.6 MembershipManager. The `MembershipManager` class makes the interface between the implementation of our *CAMCAST* algorithm and the low-level membership algorithm used to join or leave a community. Joining and leaving a community is done through the `joinCommunity()` and `leaveCommunity()` methods. The implementation of these methods depends on the underlying algorithm used (structured, unstructured and semi-structured).

In the unstructured version, the `MembershipManager` is responsible for providing the gateways for joining a specific community. This is done with the `getGateway()` method. This method is replaced in the structured version by the `searchPeers()` method which returns a set of `RemotePeers` needed for constructing the different membership tables.

5.3.2 Pubsub Package

The `pubsub` package contains the primitives of the topic-based publish/subscribe abstraction. This package makes the link between the `p2p` and `camcast` packages and defines the callback interface needed by the subscribers to receive events.

5.3.2.1 Event. The class `Event` represents an event. An event contains data, an unique identifier, a reference to the `RemotePeer` that created the event as well as a round number. For convenient purpose, it is also possible to set the probability of upward propagation of a event via the `setUpwardProb()` method.

As our *CAMCAST* algorithm ensures that no process ever receives an event it is not interested in, an event does not contain information about the type of the data. When a process receives an `Event` it can directly makes use of the data without having to perform a subtyping test.

5.3.2.2 Community and ICommunitySubscriber. A `Community` instance represents the community abstraction of our *CAMCAST* algorithm. As we will see, a `Community` is retrieved with the help of a `CommunityManager`. A `Community` makes the link between our *CAMCAST* implementation and the high-level applications using it. A `Community` is hence responsible for dispatching the different method calls it receives to the correct methods of the *CAMCAST* implementation.

A `Community` defines several methods for subscribing to a community (i.e., `subscribe()`), for leaving a community (i.e., `unsubscribe()` or `stopPublishing()`) and for publishing an event (i.e., `publish()`). The `subscribe()` method takes an instance of a `ICommunitySubscriber` in parameter. This interface represents a callback and its `onEvent()` method is called whenever a new event is received through the wire.

5.3.2.3 CommunityManager. The `CommunityManager` is responsible to retrieve a specific `Community` from the topic it represents. The communities in our implementation all relates to interfaces and thus, to retrieve a community, an instance of `java.lang.Class` needs to be provided to the `getCommunity()`

method of the `CommunityManager`. It exists only one instance of a `CommunityManager` per process (this is a singleton class).

Together with the retrieval of a community and a super-community (via the `getCommunity()` and `getSuperCommunity()` methods respectively), the `CommunityManager` implements the `IPipeListener` callback interface (see upper). This allows the `CommunityManager` to receive instances of `Message` and to dispatch them to the corresponding `Community` which in turn forwards them to the instance of our *CAMCAST* implementation.

5.3.3 Camcast Package

This package contains all the classes that implement our *CAMCAST* algorithm.

5.3.3.1 IAlgFactory and SuperAlgFactory. `IAlgFactory` represents an interface used for retrieving the algorithm which updates the instances of `TypeTable`. The retrieval of the algorithm is done via a call to the `getUpdateAlg()` method. This method returns an instance of a `IUpdateAlg`.

The `SuperAlgFactory` represents an implementation of the `IAlgFactory` interface and its `getUpdateAlg()` returns simply an instance of a `SuperUpdateAlg` (which is a subtype of `IUpdateAlg`).

5.3.3.2 SuperUpdateAlg. This class implements the algorithm for updating the super-topic table, see Section 3. It basically contacts the different processes in the super-topic table to check if they are still alive and if this is not the case replace the faulty processes with new ones.

5.3.3.3 TypeTable. This class represents a membership table of our algorithm; this can be either a topic table or a super-topic table. [11] An instance of a `TypeTable` stores an instance of a `IAlgFactory` which provides the algorithm responsible for updating the table. Furthermore a `TypeTable` is linked to a specific `Community` and contains a list of instances of `RemotePeer` that are interested in the same topic.

In the unstructured and semi-structured implementation, the membership algorithm associated with a `TypeTable` is SCAMP [24]. The implementation of SCAMP is done in the main `Camcast` class. In the structured version of the implementation, the membership algorithm is done via P-Grid [25].

[11] We named this class `TypeTable` instead of `TopicTable` to stress the fact that, in our implementation, a topic is represented by an abstract type (i.e., a Java interface)

Super-topic tables are instances of `TypeTable` with a specific `IAlgFactory` (i.e., a `SuperAlgFactory`). Super-topic tables contain references to processes interested in the super-topic of the community given as a parameter of the constructor.

5.3.3.4 Camcast. This class represents the implementation of our *CAMCAST* algorithm. It provides the different methods for disseminating an event in a hierarchy of communities. A `Camcast` instance is initialized for a specific `Community` and contains two instances of `TypeTable`: one topic table to store references to processes in the same community and one super-topic table that stores references to processes interested in the super-community.

The dissemination of an event is done through the `disseminate()` method, which implements the dissemination algorithm presented in Section 3. Moreover, in the unstructured version available on the web site, `Camcast` implements the SCAMP algorithm through the `join()` and `forward()` methods which in turn calls the `peers()` and `superPeers()` methods to retrieve processes in a community or a super-community respectively.

6 Performance

We present in this section performance measurements of both a simulated version of our *CAMCAST* algorithm together with a full Java 1.5 implementation presented in Section 5.

In short the results we obtain convey our claims of high reliability and low latency complexity, as well as confirm the analytical evaluation of Section 4.

6.1 Simulation Results

Before detailing the results we obtained for the simulated version of our algorithm, we first describe the setting and environment we used.

6.1.1 Configuration Parameters

The number of levels x in the topic hierarchy is set to 3 (a, d, g and *super(g)* = d, *super(d)* = a, a being the root topic). The number of subscribers, N_t, is 1000 for Π_g, 100 for Π_d and 10 for Π_a. The number of processes any event is disseminated to, c_t, is equal to 5 for all communities and g_t (which determines p_t^{sel}) is set to 5 for all communities. The number a_t (which determines p_t^a) is 1 for all communities. The size of the super-topic table, z_t, is 3 for all communities. The probability for an event to be received is set to an arbitrary value of 0.85, to simulate unreliable, e.g. best effort, channels. The probability for a process to crash varies. In the simulation, the membership tables (topic table and super-topic table) of a process are determined statically. These tables are initialized at the beginning of the simulation and do not change, during the entire simulation. Pessimistically, we assume that the membership algorithm does not "replace" crashed processes, and that these crash at the very beginning. Indeed, according to Section 4, such an omission provides the worst performance for *CAMCAST*, and this is exactly what we want to measure in this section.

Note that the events disseminated in the simulation belong to topic g. Our simulator (written in *C#*) simulates synchronous gossip rounds among processes in a Windows task. We use a Pentium 4, 2.6GHz, 512MB of RAM on Windows 2000 SP3. All results are averaged over 100 experiments.

6.1.2 Results

Figure 13 depicts the maximal number of events sent within a community, according to the number of processes having failed (here the state of a process – active or failed – is set at the beginning of the simulation and does not change throughout the simulation). The message complexity is of an order of $N_t \cdot ln\, N_t$ as expected.

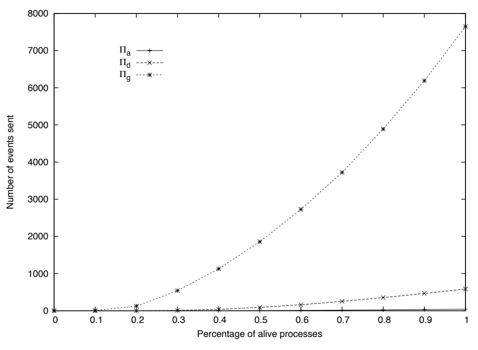

Fig. 13. Number of events sent in each community

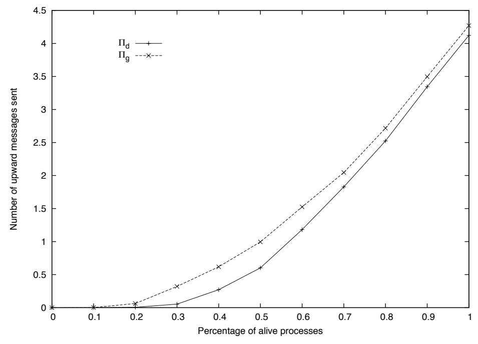

Fig. 14. Number of upward messages sent between communities to their respective super–community

Figure 14 depicts the number of events sent from community Π_g to Π_d, and from Π_d to Π_a respectively (i.e., number of *upward* messages). We can conclude that even if almost half of the processes fail, at least one event is sent to the community of processes participating in the super-community. This is enough for disseminating the event to the upper communities.

Figure 15 depicts the probability for a non-crashed process to receive an event according to the percentage of processes having crashed. Not surprisingly, the reception probability depends on the average probability that a process crashes. Of course, the reception probability is smaller for processes interested in a as the reception of an event of topic g, by the community Π_a, depends on the success of the dissemination of this event in the community Π_g and Π_d. Moreover, the gap between the results obtained for Π_g and Π_d can be explained by the fact that, as the super-topic tables are not updated, it might happen that all super-topic tables of the processes in Π_g that are responsible for propagating the events from Π_g to Π_d have crashed. Hence the event is not received in Π_d.

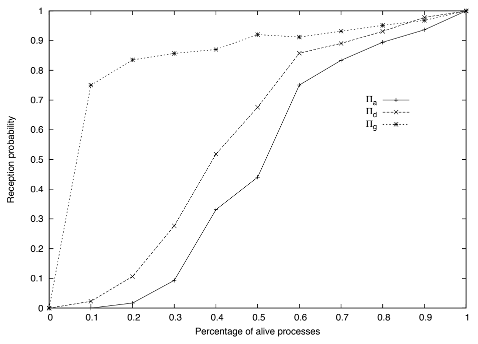

Fig. 15. Reception probability (only stillborn processes)

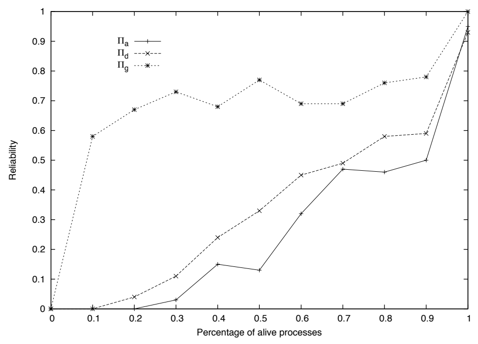

Fig. 16. Reliability (only stillborn processes)

Figure 16 depicts the probability that *every* non-crashed process receives an event according to the percentage of processes that have crashed (i.e., the reliability metric). As expected, we can notice that the reliability depends on the community in which the event is disseminated to. As described previously, the reliability is very sensible to invalid super-topic tables. As soon as the percentage of alive processes increases in Π_d and Π_a, the reliability increases rapidly. Furthermore, to achieve better reliability, we can easily adjust z_t, p_t^a and g_t.

Figure 17 and Figure 18 present the same results as, respectively, Figure 15 and Figure 16, except that now a process can appear to be crashed for a process while appearing alive for another one (to simulate a weakly consistent membership algorithm). We achieve a much better reliability for a weakly connected system than in the preceding scenarios.

We consider in Table 1 the average number of rounds needed to disseminate an event from community Π_{t_j} to community Π_{t_0} (with *super(t_i)* equals t_{i-1}, t_0 being the root topic). Three different topologies are considered: (a) $N_{t_2} = 1000$, $N_{t_1} = 100$ and $N_{t_0} = 10$, (b) $N_{t_j}..N_{t_0} = 100, j = 1..2$ and (c) $N_{t_j}..N_{t_0} = 100, j = 1..4$. We compare (1) our approach with (2) a general gossip-based protocol [27] and with (3) PMcast [22] (for (2) and (3), the values have been calculated using the analytical equations and are not taken from simulations). The results confirm the analytical results given in Section 4.6 and convey the impact of the number of levels in a hierarchy on the latency complexity.

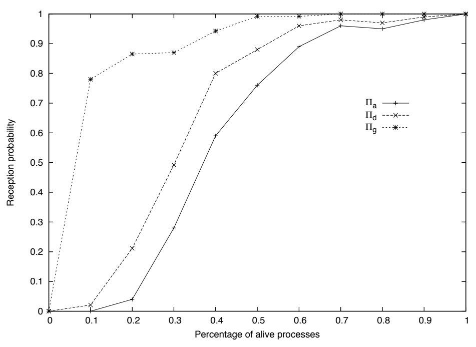

Fig. 17. Reception probability

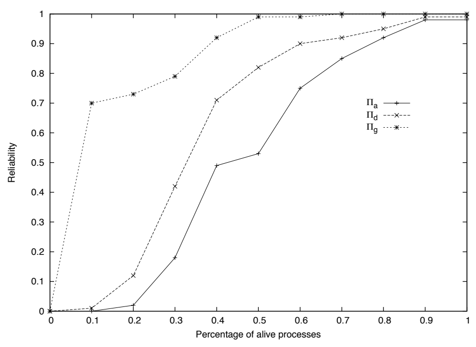

Fig. 18. Reliability

	N_{t_j},$N_{t_{j-1}}$,...,N_{t_0}		
	1000, 100, 10	100 ($j = 2$)	100 ($j = 4$)
(1)	8.91	8.83	13.08
(2)	4.63	4.39	4.31
(3)	6.61	5.44	5.89

Table 1
Average number of rounds to disseminate an event in all communities

In Table 2, we consider the average number of parasite events sent in the system. To calculate the maximal number of parasite events, we assume that a publisher publishes an event of the root topic t_0. For PMcast, the results obtained come from analytical results. Table 2 depicts the gain of *CAMCAST* with respect to alternative approaches.

	N_{t_j},$N_{t_{j-1}}$,...,N_{t_0}		
	1000, 100, 10	100 ($j = 2$)	100 ($j = 4$)
(1)	0	0	0
(2)	11216	1699	3739
(3)	453	369	615

Table 2
Total number of parasite events

6.2 Implementation Results

We present now the different performance measurements of the Java implementation (see Section 5) or our *CAMCAST* algorithm. We measured the latency complexity, efficiency, reception probability, reliability of *CAMCAST*, as well as the number of upward messages sent between communities.

In short, our results convey the results we had in both Section 4 and Section 6.1, for all the three different membership algorithms. We can however notice that the more structured the membership algorithm, the better the overall performance of *CAMCAST*.

6.2.1 *Configuration Parameters*

The environment used to test our prototype consisted of 71 computers, each running two processes participating in *CAMCAST* (the total number of processes vary depending on the membership service used). The computers were SUN Ultra workstations, running Solaris 8, with UltraSPARC-IIi processors running at 440MHz or 360MHz and 256MB of RAM, connected in a 100 MB LAN.

The same hierarchy as the one described in Section 6 was used to test our implementation (i.e., 3 topics, a, d, g, where *super*(g) = d, *super*(d) = a, a being the root topic). The parameter p_t^{sel} (inter-community dissemination probability) varies from 0.1 to 1 for all communities. The parameter a_t (which determines p_t^a) is 2 for all communities. The size of the super-topic table, z_t, is 4 for all communities. With the unstructured membership algorithm, the number of subscribers N_g is equal to 84, N_d is equal to 27 and N_a is equal to 7. With the semi-structured membership algorithm, the number of subscribers, N_g is equal to 79, N_d is equal to 28 and N_a is equal to 7. With the structured membership algorithm, the number of subscribers, N_g is equal to 79, N_d is equal to 28 and N_a is equal to 7. The parameter c_t (in $ln\, N_t + c_t$) is set to 0 for all communities. This parameter is set to this value because we were unable to perform our experiments on very large networks (see upper). In such small networks, and as we populate the topic tables randomly but following the entry of the processes in the system (and not randomly when all the processes are present in the system), the bigger the topic tables, the higher the probability to create partitions among this system with the processes entering the system at the end of the bootstrapping phase. Our implementations use TCP channels (with disrespect to the channels used in our simulations) and no process crash during the runs (unlike the simulations). All the results shown are for one event of topic g that is disseminated from Π_g to Π_a via Π_d. We do not show the results for community Π_g, as these are not dependent on the inter-community dissemination probability. Finally, note that we use the logarithmic scale for the X abscissa.

6.2.2 *Results*

During the tests, we measured the framework performance, for each of its membership service, through its event reception probability, reliability, latency complexity, number of upward messages and *efficiency* by varying p_t^{sel}. The efficiency measures how many of the forwarded messages are actually useful, that is, have not already been received by the process. We calculate the efficiency as the percentage of useful messages from the total amount of messages received:

$$Efficiency_t = \frac{useful\ messages_t}{duplicate\ messages_t + useful\ messages_t}$$

The three membership services used in *CAMCAST* are shown together in each of all the figures presented, each of which represents the results for a specific community. The performance for V_1 (unstructured), V_2 (semi-structured) and V_3 (structured) are shown using plain, interrupted and dotted lines, respectively. All the results are averages over 10 experiments for V_1 and 5 experiments for V_2 and V_3.

We can notice that all three approaches converge to a reception probability of 100%, both for community Π_d (see Figure 19) and for community Π_a (see Figure 20). The structured approach ensures very good results even for a low inter-community dissemination probability.

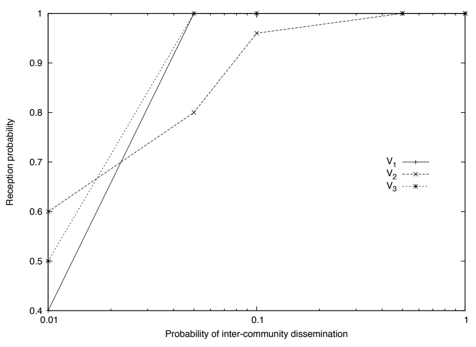

Fig. 19. Reception probability for Π_d, while varying the probability of inter-community dissemination from Π_g to Π_d

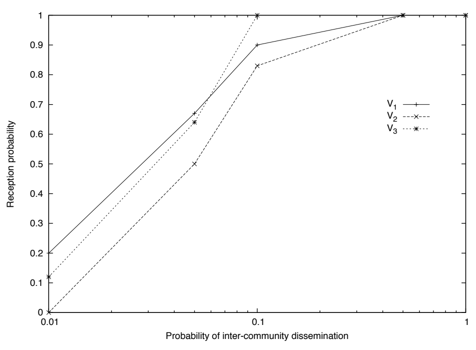

Fig. 20. Reception probability for Π_a, while varying the probability of inter-community dissemination from Π_d to Π_a

The structured approach converges the fastest thanks to the uniformity and randomness of the membership tables. The inter-community dissemination probability (p_t^{sel}) influences reception probability because the more upward messages there are, the more disconnected processes from the super-community are reached. As we can see in Figure 19 and Figure 20, for the test where $p_t^{sel} < 0.5$, the unstructured (V_1) and semi-structured (V_2) approaches perform differently (this is due to the choice of contacts provided by the bootstrapping mechanism).

As we can see in Figure 21 and Figure 22, the reliability of all approaches is very high, even with low inter-community dissemination probability (0.1). This is due to the fact that we consider failure free runs during our simulation. Hence, when an event managed to be propagated from one community to another, it is highly susceptible to be propagated to all the processes in the community as well.

The efficiency metric test results, depicted in Figure 23 and Figure 24 confirms that a higher inter-community dissemination probability increases the number of duplicate messages in the system, both directly (the number of upward messages increases proportionally) and indirectly (as the reliability increases, the number of intra-community messages also increases). Please note that in Figure 24, the value for V_2 corresponding to an inter-community dissemination probability of 0.01 is equal to 0 (i.e., no event is propagated from Π_d to Π_a). Our results relate to the theory presented in Section 4.3, which presents the upper bound for the total number of messages. The structured approach behaves better than the semi-structured approach, which is better than the unstructured approach. This can again be ex-

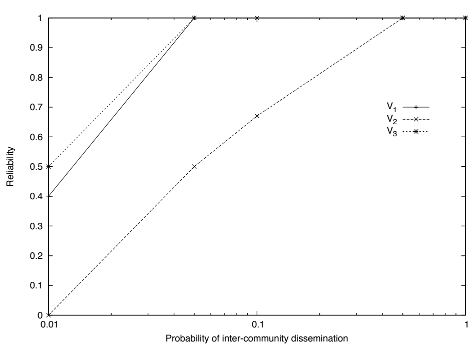

Fig. 21. Reliability for Π_d, while varying the probability of inter-community dissemination from Π_g to Π_d

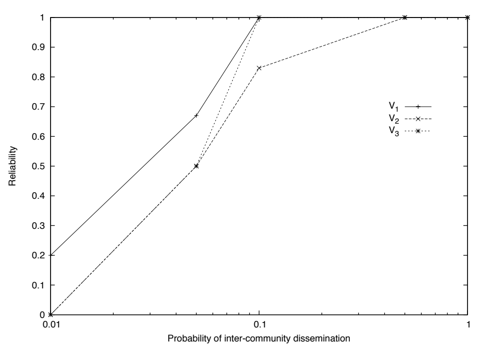

Fig. 22. Reliability for Π_a, while varying the probability of inter-community dissemination from Π_d to Π_a

plained by the uniformity of the membership tables. As the structured approach as the greatest reliability, it has the greatest numerator in the efficiency equation.

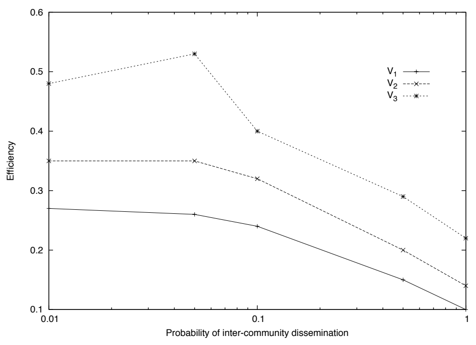

Fig. 23. Efficiency for Π_d, while varying the probability of inter-community dissemination from Π_g to Π_d

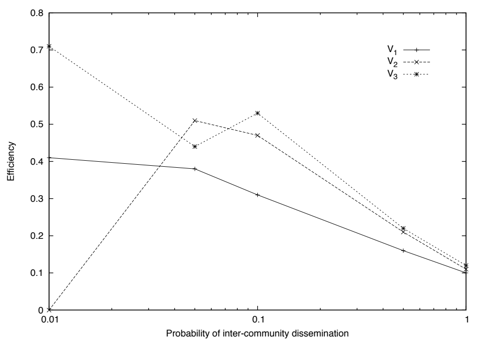

Fig. 24. Efficiency for Π_a, while varying the probability of inter-community dissemination from Π_d to Π_a

Figure 25 and Figure 26 depict the latency complexity results (some results equal 0 when it happened that an event was not propagated up to that community). As presented in Section 4.6, the size of a community and whether the event is directly propagated to the super-community, greatly influence this metric. The topology created by the membership tables also influences the latency complexity, by how much it deviates from a random graph (which requires the minimal hops to propagate an event into the whole community). Clearly, in Figure 25, Figure 26, we are in the case in which the event is directly propagated to the super-community. We can see that the latency complexity decreases while the inter-community dissemination probability increases. This is because, as depicted in Section 4.6, the sooner the event is propagated to the super-community, the less rounds are needed to propagate the event into the whole community.

Finally, the number of upward messages (Figure 27 and Figure 28) describes the inter-community communication and is directly influenced by the inter-community dissemination probability, as expected from the analysis (see Sec-

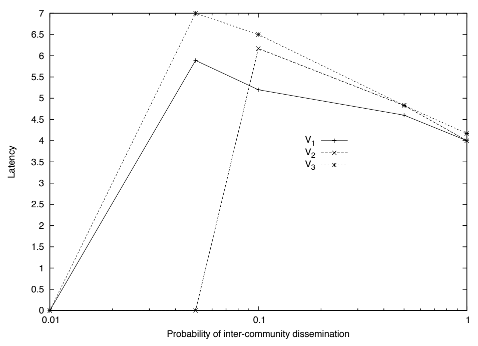

Fig. 25. Latency complexity for Π_d, while varying the probability of inter-community dissemination from Π_g to Π_d

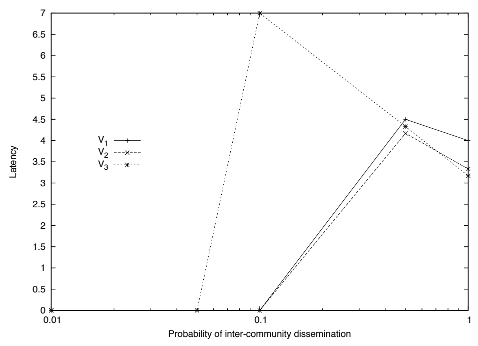

Fig. 26. Latency complexity for Π_a, while varying the probability of inter-community dissemination from Π_d to Π_a

tion 4.3). The minor differences between the three approaches come from the different number of reached super-processes.

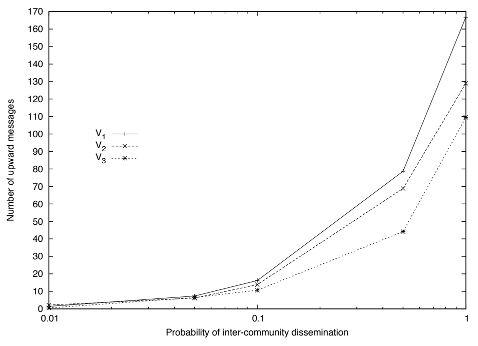

Fig. 27. Upward messages received by Π_d from Π_g

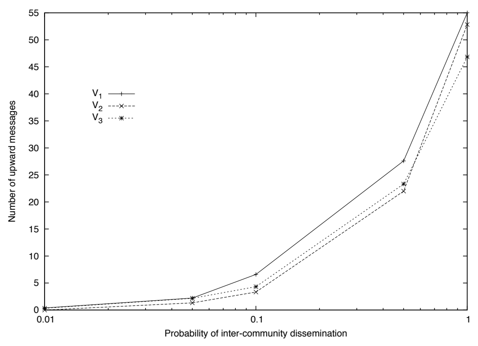

Fig. 28. Upward messages received by Π_a from Π_d

7 Concluding Remarks

We presented *CAMCAST*, a simple algorithm for disseminating events in a hierarchical decentralized topic-based publish/subscribe system. Each topic defines a dynamic notion of community. *CAMCAST* is decentralized and prevents processes from receiving and forwarding events from communities they do not belong to. This is achieved while ensuring that the membership information each process must maintain depends on the size of its communities.

We demonstrated the viability of *CAMCAST* through analytical results, simulations and performance measurements. Our experiments and simulations convey a fairly high dissemination probability by tuning the different parameters of *CAMCAST*, without increasing the memory complexity. For instance, we showed that it is possible to achieve 100% reliability among a hierarchy of three communities of 80, 30 and 7 processes, respectively, while only involving 7% of the processes in the propagation of the events in between the communities.

For presentation simplicity, we tackled the case where a topic has only one parent. Multiple parents could be easily supported by adding a super-topic table for each parent. This would not hamper the overall performance of the algorithm, under the assumption that there is an upper bound on the number of parents for each topic.

References

[1] TIBCO, TIB/Rendezvous White Paper, http://www.rv.tibco.com/ (1999).

[2] M. Altherr, M. Erzberger, S. Maffeis, iBus - A Software Bus Middleware for the Java Platform, in: Proceedings of the International Workshop on Reliable Middleware Systems of the 13th IEEE Symposium On Reliable Distributed Systems, 1999, pp. 43–53.

[3] D. Skeen, Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview, http://www.vitria.com (1998).

[4] IBM, MQSeries: Using Java, IBM, 2000.

[5] Microsoft, Microsoft Message Queuing, http://www.microsoft.com/windows2000/technologies/communications/msmq (2005).

[6] S. M. Inc., Java Message Service - Specification, version 1.1, http://java.sun.com/products/jms/docs.html (2005).

[7] B. Kantor, P. Lapsley, Network News Transfer Protocol, Request for Comments (1986).

[8] R. M. Karp, C. Schindelhauer, S. Shenker, B. Vocking, Randomized Rumor Spreading, in: Proceedings of the 41st IEEE Symposium on Foundations of Computer Science, 2000, pp. 565–574.
URL `citeseer.nj.nec.com/article/karp00randomized.html`

[9] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, Y. Minsky, Bimodal Multicast, ACM Transactions on Computer Systems 17 (2) (1999) 41–88.

[10] M.-J. Lin, K. Marzullo, Directional Gossip: Gossip in a Wide Area Network, in: Proceedings of the 3rd European Dependable Computing Conference, 1999, pp. 364–379.

[11] P. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, P. Kouznetsov, Lightweight Probabilistic Broadcast, ACM Transactions on Computer Systems 21 (4) (2003) 341–374.

[12] A. J. Ganesh, A.-M. Kermarrec, L. Massoulié, SCAMP: Peer-to-Peer Lighweight Membership Service for Large-scale Group Communication, in: Proceedings of the 3rd International Workshop on Networked Group Communication, 2001.

[13] P. Eugster, R. Guerraoui, A.-M. Kermarrec, L. Massoulié, A. J. Ganesh, From Epidemics to Distributed Computing, in: IEEE Computer, Vol. 37, 2004, pp. 60–67.

[14] A. Rowstron, A.-M. Kermarrec, M. Castro, P. Druschel, SCRIBE: The Design of a Large-Scale Event Notification Infrastructure, in: Proceedings of the 3rd International Workshop on Networked Group Communication, 2001.

[15] A. Rowstron, P. Druschel, Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems, in: Proceedings of the 4th IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, 2001, pp. 329–350.

[16] S. Ratnasamy, M. Handley, R. Karp, S. Shenker, Application-Level Multicast Using Content-Addressable Networks, Lecture Notes in Computer Science 2233 (2001) 14–29.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A Scalable Content Addressable Network, in: Proceedings of the 7th ACM Conference on Special Interest Group on Data Communications, 2001.

[18] K. Jenkins, K. Hopkinson, K. Birman, A Gossip Protocol for Subgroup Multicast, in: Proceedings of the 1st International Workshop on Applied Reliable Group Communication, 2001.

[19] A. Carzaniga, D. Rosenblum, A. Wolf, Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service, in: Proceedings of the 19th ACM Symposium on Principles of Distributed Computing, 2000, pp. 219–227.

[20] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, D. Sturman, Exploiting IP Multicast in Content-Based Publish-Subscribe Systems, in: Proceedings of the 3rd IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, 2000, pp. 185–207.

[21] P. R. Pietzuch, J. M. Bacon, Hermes: A distributed event-based middleware architecture, in: Proceedings of the 1st International Workshop on Distributed Event-Based Systems, 2002.

[22] P. Eugster, R. Guerraoui, Probabilistic Multicast, in: Proceedings of the 3rd IEEE International Conference on Dependable Systems and Networks, 2002, pp. 313–322.

[23] R. van Renesse, K. P. Birman, W. Vogels, Astrolabe: A Robust and Scalable Technology For Distributed Systems Monitoring, Management, and Data Mining, ACM Transactions on Computer Systems 21 (3).

[24] A.-M. Kermarrec, L. Massoulié, A. J. Ganesh, Probabilistic Reliable Dissemination in Large-Scale Systems, in: IEEE Transactions on Parallel and Distributed Systems, Vol. 14, 2003, pp. 248–258.

[25] K. Aberer, P-Grid: A Self-Organizing Access Structure for P2P Information Systems, in: Proceedings of the 6th International Conference on Cooperative Information Systems, 2001.

[26] M. Jelasity, R. Guerraoui, A. Kermarrec, M. van Steen, The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-based Implementations, in: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, Vol. 78, 2004, pp. 79–98.

[27] P. Erdös, A. Renyi, On the Evolution of Random Graphs, in: Mat Kutato Int. Közl, Vol. 5, 1960, pp. 17–60.

[28] Y. Zibin, J. Gil, Efficient Subtyping Tests with PQ-Encoding, in: Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications, 2001.

[29] B. Bollobás, Random Graphs, Cambridge University Press, 2001.