Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation

Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, Gianpaolo Cugola
Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
{costa,migliava,picco,cugola}@elet.polimi.it

Abstract— **Distributed content-based publish-subscribe middleware is emerging as a promising answer to the demands of modern distributed computing. Nevertheless, currently available systems usually do not provide reliability guarantees, which hampers their use in dynamic and unreliable scenarios, notably including mobile ones. In this paper, we evaluate the effectiveness of an approach based on epidemic algorithms. Three algorithms we originally proposed in [1] are here thoroughly compared and evaluated through simulation in a challenging unreliable setting. The results show that our use of epidemic algorithms improves significantly event delivery, is scalable, and introduces only limited overhead.**

I. Introduction

Publish-subscribe middleware is emerging as a promising tool to tackle the demands of modern distributed computing. In particular, distributed content-based systems (e.g., [2]–[6]) provide high levels of scalability, flexibility, and expressiveness by exploiting a distributed architecture for event dispatching, and by using a content-based scheme for matching events and subscriptions.

Our research in this field is motivated by the desire to exploit the good properties of distributed content-based publish-subscribe in scenarios where the topology of the dispatching infrastructure is continuously under reconfiguration, e.g., mobile computing and peer-to-peer applications. Achieving this goal requires the solution of several problems. In [7] we tackled the problem of efficiently reconfiguring subscription information and restoring event routing. The topic of this paper, instead, is the complementary problem of recovering events lost during reconfiguration. Our approach is based on epidemic algorithms [8], [9]. In [1] we described three different solutions to the problem. In this paper, we complete and validate our initial proposal by thoroughly evaluating its effectiveness in challenging unreliable scenarios.

The contribution put forth by this paper is relevant under many respects. Reliability is an aspect rarely considered in content-based publish-subscribe systems—let apart being evaluated quantitatively. Our epidemic algorithms, whose effectiveness and efficiency is quantitatively demonstrated by the simulations discussed in this paper, provide a viable solution for improving reliability. In addition, although our work is driven by the need to recover events lost due to topological reconfiguration, it does not make any assumption about the source of event loss. Hence, our solutions enjoy general applicability. Finally, as discussed in the rest of the paper, epidemic algorithms have been applied to a number of applicative domains but, with the exception of [10], not to *content-based* publish-subscribe systems. By devising original solutions in this domain, we explore new uses for this technique.

The paper is structured as follows. Section II is a concise overview of content-based publish-subscribe systems. Section III describes the epidemic algorithms we originally proposed in [1] for achieving reliability in content-based publish-subscribe systems. An extensive evaluation of these algorithms, based on simulation, is the subject of Section IV and the core contribution of this paper. Finally, Section V places our work in the context of related research efforts, and Section VI ends the paper with brief concluding remarks.

II. Content-Based Publish-Subscribe

The last few years have seen the development of a large number of publish-subscribe middleware, which differ along several dimensions[1]. Two are usually considered fundamental: the expressiveness of the subscription language and the architecture of the event dispatcher.

The expressiveness of the subscription language draws a line between *subject-based* systems, where subscriptions identify only classes of events belonging to a given channel or subject, and *content-based* systems, where subscriptions may contain expressions (called *event patterns*) enabling increased flexibility and expressiveness through sophisticated matching on the event content.

The architecture of the event dispatcher can be either centralized or distributed. In this paper, we focus on publish-subscribe middleware with a distributed event dispatcher. In such middleware, a set of *dispatching servers*[2] are connected in an overlay network, as shown in Figure 1. The servers cooperate in collecting subscriptions coming from clients and in routing events, with the goal of reducing the network load and increasing scalability. Systems exploiting a distributed dispatcher can be further classified according to the interconnection topology of the dispatching servers, and the strategy exploited to route subscriptions and events. In this work we consider a subscription forwarding scheme on an unrooted tree topology as this choice covers the majority of existing systems.

In a subscription forwarding scheme [2], subscriptions are delivered to every dispatcher along a single unrooted tree overlay network connecting all the dispatchers, and are used to establish the routes that are followed by published events.

[1]For more detailed comparisons see [2], [6], [11].

[2]Unless otherwise stated, in the following we refer to a *dispatching server* simply as *dispatcher*, although the latter represents the whole distributed component in charge of dispatching events instead of a specific server.

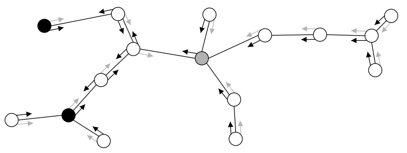


Fig. 1: *A dispatching network with subscriptions laid down according to a subscription forwarding scheme.*

When a client issues a subscription, a message containing the corresponding event pattern is sent to the dispatcher the client is attached to. There, the event pattern is inserted in a subscription table, together with the identifier of the subscriber. Then, the subscription is propagated by the dispatcher, which now behaves as a subscriber with respect to the rest of the dispatching network, to all of its neighboring dispatchers on the overlay network. In turn, they record the subscription and re-propagate it towards all their neighboring dispatchers, except for the one that sent it. This scheme is usually optimized by avoiding subscription forwarding of the same event pattern in the same direction. The propagation of a subscription effectively sets up a route for events, through the reverse path from the publisher to the subscriber. Requests to unsubscribe from an event pattern are handled and propagated analogously to subscriptions, although at each hop entries in the subscription table are removed rather than inserted.

Figure 1 shows a dispatching network with two dispatchers subscribed for a "black" pattern, and one for a "gray" pattern. Arrows represent the routes laid down according to these subscriptions, and reflect the content of the subscription tables of each dispatcher in the network. Solid lines are links of the tree overlay network. As a consequence of the subscription forwarding process, the routes for the two subscriptions are laid down on this single tree. This choice is typical of content-based systems and is motivated by the fact that a single event may match multiple patterns. Routing on multiple independent trees, as typically done by subject-based systems, would lead to inefficient duplication of events along the separate trees.

Finally, here and in the rest of the paper we ignore the presence of clients and focus only on dispatchers. Accordingly, with some stretch of terminology we say that a dispatcher is a subscriber if at least one of its clients is, although in principle only clients can be subscribers.

III. Introducing Reliability

Existing distributed content-based publish-subscribe systems rarely address reliability through dedicated mechanisms. This section describes three epidemic algorithms we developed to overcome this limitation. They are designed to recover lost events by operating on top of a best effort content-based publish-subscribe system behaving as described in Section II. Section IV thoroughly evaluates their effectiveness by simulating their behavior in challenging unreliable scenarios.

A. Epidemic Algorithms

The general idea behind epidemic (or *gossip*) algorithms [8], [9] is for each process to communicate periodically its partial knowledge about the system "state" to a random subset of other processes, thus contributing to build a common, shared view of the system state. The mode of communication can exploit a push or pull style. In a *push* style, each process gossips periodically to disseminate its view of the system to other processes. Instead, in a *pull* style each process solicits the transmission of information from other processes to compensate for local losses. Usually a push style of communication uses gossip messages containing a *positive* digest of the system state to be disseminated. Conversely, a pull approach usually exploits *negative* digests, and gossip messages hence contain the portion of the state that is known to be missing.

Independently from the scheme adopted, the probabilistic and decentralized nature of epidemic algorithms yields a number of desirable properties. They impose a constant, equally distributed load on the processes in the system, and are very resilient to changes in the system configuration, including topological ones. Moreover, these properties are preserved as the size of the system increases, thus leading to good scalability. Finally, they are usually very simple to implement and rather inexpensive to run. Hence, epidemic algorithms appear as good candidates for the dynamic distributed scenarios we target. Nevertheless, their application to the case of recovering lost events in content-based publish-subscribe system is not straightforward, as discussed in the next section.

B. Our Approach

In our solution, the state that must be reconciled through gossip is the set of events appeared in the system. Missing events are recovered through one or more "gossip rounds" during which other dispatchers, potentially holding a copy of the event, are contacted. This apparently simple task is greatly complicated by the nature of content-based publish-subscribe systems. Unlike subject-based publish-subscribe and IP multicast, events are not associated at the source to a subject or group which determines its routing. Moreover, an event may match multiple subscriptions, instead of a single group. These characteristics make more difficult to identify the subset of dispatchers that may hold missing events, and prevent a direct use of solutions developed for the aforementioned domains. This section presents three epidemic algorithms designed for content-based publish-subscribe systems. Presentation of the algorithms is kept concise, since the emphasis of this paper is on their evaluation. The interested reader can find a more extensive presentation, including a formalization, in [1].

The solutions we describe share a common structure. Each dispatcher periodically starts a new round of gossip. When playing this *gossiper* role, a dispatcher builds a gossip message and sends it to some of its neighbors, which in turn propagate it on the dispatching tree. The content of the gossip message and its routing along the tree vary according to the algorithm at hand. The sending of a missing event takes place using a direct link, i.e., out of band with respect to the normal publish-subscribe operations. Hence, we assume the existence of a unicast transport layer (not necessarily reliable, e.g., UDP-based), and that each dispatcher caches the events received.

Push. The first algorithm we developed uses proactive gossip push with positive digests. At each gossip round, the gossiper chooses randomly a pattern p from its subscription table, constructs a digest including the (globally unique) identifiers[3] of all the cached events matching p, builds a gossip message containing the digest, and labels it with p. The message is then propagated along the dispatching tree as if it were a normal event message matching p. The only difference with respect to event routing is that, to limit overhead, the gossip message is forwarded only to a random subset of the neighbors subscribed to p, according to the probability $P_{forward}$.

Observe that p is selected by considering the whole subscription table instead of only the subscriptions issued locally to the gossiper. Based on the routing strategy described in Section II, this means that p can be indirectly known to the gossiper because the latter is on the route towards a subscriber for p. Hence, by considering also these subscriptions, we increase the chance of eventually finding all the dispatchers interested in the cached events, and thus speed up convergence.

When a dispatcher receives a gossip message labelled with p, it checks if it is subscribed to this pattern and if all the identifiers contained in the digest correspond to events that have already been received. The identifiers of missed events are included in a request message sent to the gossiper, to which it replies by sending a copy of the actual events. Both messages are exchanged by exploiting the out of band channel.

Pull. In some situations a proactive push approach may converge slowly or result in unnecessary traffic. In these cases, an approach using reactive pull with negative digests may be preferable. Nevertheless, this requires the ability to detect which messages have been lost. In subject-based systems, this is easily achieved by using a sequence number per source and per subject. In content-based systems this task is complicated by absence of a notion of subject and by the fact that each dispatcher receives only those events whose content matches the patterns it is subscribed to. As detailed in [1], this problem can be solved by tagging each event with enough information to enable loss detection. The event identifier in this scheme contains the event source, information about all the patterns matched by the event and, for each pattern, a sequence number incremented at the source each time an event is published for that pattern. This information is associated to each event at its source: this is made possible by the fact that a subscription forwarding strategy is chosen, and hence subscriptions are known to all dispatchers.

Whenever a dispatcher receives an event matching a pattern p, but for which the sequence number associated to p in the event identifier is greater than the one expected for that pattern and source, it can detect the loss of an event and trigger the appropriate actions. In [1] we elaborated two algorithms that rely on this technique for detecting event loss, but use different routing strategies: the first one steers gossip messages towards the event receivers (the subscribers), while the other steers them towards the event sender (the publisher).

- *Subscriber-Based Pull.* In this pull scheme, when a dispatcher detects a lost event it inserts the corresponding information, i.e., the source, the matched pattern, and the sequence number associated to the pattern and source, in a buffer $Lost$. When the next gossip round begins the dispatcher, now a gossiper, chooses a pattern p among the ones associated to subscriptions issued locally, selects the events in $Lost$ related with p, and inserts the corresponding information in a digest attached to a new gossip message. Note that, unlike in push, subscriptions are not drawn from the whole subscription table, since here the goal is to retrieve events relevant to the gossiper rather than disseminating information about received events. Finally, the gossip message is labelled with p and routed in a way analogous to the push solution. A dispatcher receiving the gossip message checks its event cache for events requested by the gossiper and, if any are found, sends them back to it. Observe that, in this case, it is not necessarily the case that the dispatcher is a subscriber for the pattern p specified by the gossiper. The dispatcher could have received the gossip message because it is on a route towards a subscriber for p, and could have received (and cached) some of the events missed by the gossiper because they match also a pattern $p' \neq p$ the dispatcher is subscribed to.
- *Publisher-Based Pull.* Our second pull scheme requires published events to be cached not only by the dispatchers that received them but also by the source. Moreover, it also requires that the address of each dispatcher encountered during the travel towards a subscriber is appended to the event message. The algorithm behaves then similarly to the previous one, but routes gossip messages towards publishers rather than subscribers.
  In particular, while $Lost$ contains the same information as before, a new buffer $Routes$ is necessary to store the route towards a given publisher (e.g., based on the route information stored in the event most recently received from it). Moreover, the information distinctive of a gossip message is now the event source rather than the pattern, and the gossip message is augmented with the information necessary to be routed back to the publisher, taken from $Routes$. It is worth noting that there is no guarantee that the route stored in $Routes$ is the same originally followed by the missing event, since changes in the dispatching network could have occurred meanwhile. On the other hand, it is likely that the two share at least the first portion or, in the worst case, the publisher.

IV. Evaluation

This section presents simulation results that allow one to critically evaluate our use of epidemic algorithms. We evaluate the performance of our push, subscriber-based pull, and publisher-based pull algorithms. We also consider the case where the two pull approaches are combined because, as discussed in the following, this option enables a significant performance improvement. In addition, we also simulate the behavior of a random pull approach where routing of gossip messages is performed entirely at random. This alternative

[3]The pair given by the source identifier and a monotonically increasing sequence number associated to the source is sufficient.

Parameter	Default Value
number of dispatchers	$N = 100$
maximum number of patterns per subscriber	$\pi_{max} = 2$
publish rate	50 publish/s
link error rate	$\epsilon = 0.1$
interval between topological reconfigurations	$\rho = +\infty$
buffer size	$\beta = 1500$
gossip interval	$T = 0.03s$

Fig. 2: *Simulation parameters and their default values.*

allows us to evaluate if the extra effort of deciding how to route gossip messages is worthwhile. Simulations of a similar random push approach are omitted since their performance is extremely poor.

Section IV-A describes the simulation setting, while Section IV-B illustrates the results. Simulations consider two very different unreliable scenarios: one where links are lossy and the percentage of events lost is then *directly* determined by the link error rate, and one where event loss is *indirectly* determined by a topological reconfiguration taking place in the overlay network. Most of the results focus on the former scenario, as it is more general and its effects more easily isolated and controlled.

A. Simulation Setting

In absence of reference scenarios for comparing content-based publish-subscribe systems, we defined our own by choosing what we believe are reasonable and significant values. The simulation parameters are discussed below and summarized, together with their default values, in Figure 2.

Modeling content-based publish-subscribe. For this set of parameters, we built upon the simulation parameters used in previous work by some of the authors [7].

- *Events, subscriptions, and matching.* Events are represented as randomly-generated sequence of numbers, where each number represents a pattern of the system. We chose a uniform distribution for this sequence. An event pattern is represented as a single number. An event matches a subscription if it contains the number specified by the event pattern in the subscription. Each dispatcher can subscribe to a maximum number π_{max} of event patterns, drawn randomly from the overall number Π of patterns available in the system, which in our simulation is set to 70. Given these parameters, it is possible to calculate the number of subscribers per pattern as $N_{\pi} = (N\pi_{max})/\Pi$ whose default value, given the values in Figure 2, is 2.85.
- *Publish rate.* Our simulations are run with dispatchers continuously publishing events on a network with stable subscription information (i.e., no (un)subscriptions are being issued). The frequency at which the publish operation is invoked by each dispatcher determines the system load in terms of event messages that need to be routed. As a default, we choose a high publishing load scenario with about 50 publish/s per dispatcher. In some of the simulations we also consider a low publishing load scenario of about 5 publish/s per dispatcher.
- *Overlay network topology.* The results we present here are all obtained with configurations where each dispatcher is connected, in the dispatching tree, with at most four others. Clients are not modeled explicitly, as their activity ultimately affects only the dispatcher they are attached to.

Modeling the sources of event loss. The relevant parameters differ according to the unreliable scenario considered:

- *Channel reliability.* We assume that each link connecting two dispatchers in the overlay network behaves as a 10 Mbit/s Ethernet link. For the lossy link scenario, we simulated scenarios with an error rate $\epsilon = 0.1$ (leading to 75% of events lost) and $\epsilon = 0.05$ (leading to a 55% loss). In the case of topological reconfiguration, the links are instead assumed to be fully reliable.
- *Frequency of reconfiguration.* For the scenario with topological reconfigurations we relied on the algorithm and simulations described in [7], where the interested reader can find more details. A reconfiguration in this setting is the breakage of a link, and its replacement with another that maintains the network connected. We assume that the overlay network is repaired in $0.1s$. Reconfigurations are triggered with a frequency determined by the duration of the interval ρ between two reconfigurations. We simulated non-overlapping reconfigurations ($\rho = 0.2s$) where a link is replaced by another before a new link breaks, as well as overlapping ones ($\rho = 0.03s$). Clearly, the former cause less disruption and hence less event loss.

Gossip parameters. The parameters governing the behavior of our gossip algorithms are the following:

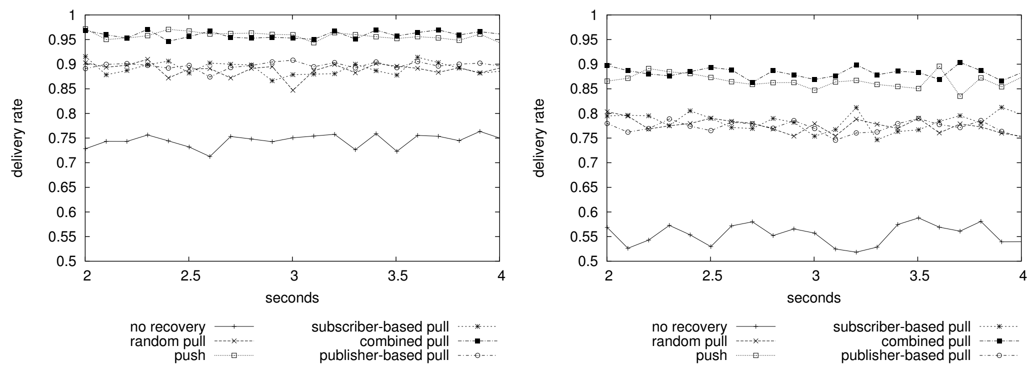
- *Buffer size.* Each dispatcher is equipped with a buffer where events are stored, to satisfy retransmission requests. The buffer has a size of β elements. In our simulations we adopted a simple FIFO buffering strategy where each dispatcher caches only events for which it is either the publisher or a subscriber.
- *Gossip interval.* The frequency of gossiping is controlled by the gossip interval T between two gossip rounds.
- *Combining pull approaches.* As discussed later, we combined these two pull approaches to improve performance. Which approach is used at a given moment is determined in probabilistic terms, using the P_{source} parameter.

Effect of randomization. The results of 10 simulations ran with different random seeds showed that, differently from [7], variations are limited, around 1%-2%. Hence, we present here the results of a single simulation, rather than averaging over runs with different seeds.

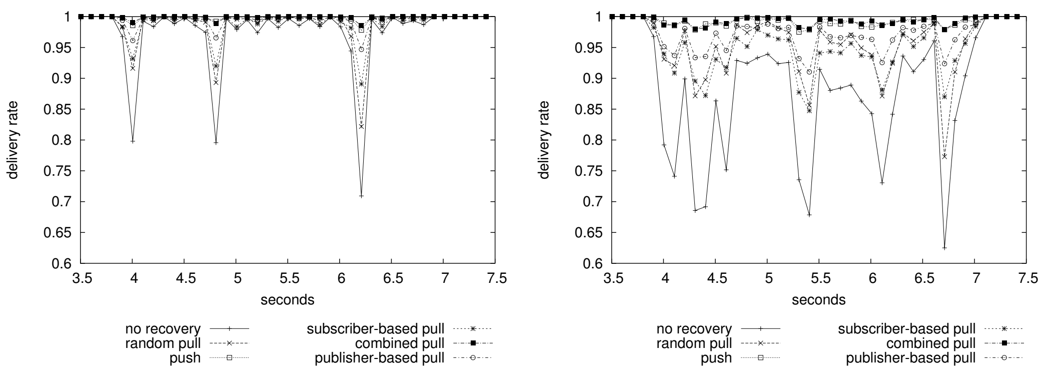
Simulation tool. Our simulations are developed using OMNeT++ [12], a free, open source discrete event simulation tool.

B. Event Delivery

As mentioned in Section I, our initial and driving motivation for tackling reliability was to cope with event loss induced by the dynamic reconfiguration of the dispatching infrastructure, e.g., due to mobility. Nevertheless, thus far *we did not make any hypothesis about the cause of event loss.* Hence, our algorithms enjoy general applicability, and in principle can



(a) Scenario with lossy links, $\epsilon = 0.05$ (left) and $\epsilon = 0.1$ (right).



(b) Scenario with topological reconfigurations, $\rho = 0.2s$ (left) and $\rho = 0.03s$ (right).

Fig. 3: *Event delivery.*

improve reliability in any situation where an event loss may occur. In this section, we evaluate the effectiveness of our approach in improving event delivery, by considering both the common scenario of a stable topology with lossy links, and the scenario where events are lost because of changes in the topology of the overlay network.

Figure 3(a) shows the simulation results[4] for the former case, by comparing the performance of the various solutions when the error rate $\epsilon = 0.05$. The performance metric we choose is the delivery rate, i.e., the ratio between the number of events correctly received by a process and those that would be received in a fully reliable scenario. The delivery rate in the chart is averaged, and shown in percentage. The time on the x-axis is the simulation time.

In this scenario, our baseline is the delivery rate obtained without any form of recovery, which is around 75%. The chart shows how neither of the pull solutions alone is sufficient to achieve a satisfactory delivery rate. This can be easily understood by focusing on the special case where only one dispatcher is subscribed for a given pattern. A subscriber-based approach is not very effective, because there are no other dispatchers to gossip with. Instead, a publisher-based is more convenient in this case. Nevertheless, in a situation where there are many dispatchers subscribed to the same pattern a publisher-based approach is less appealing, since gossip involves a much smaller fraction of the dispatchers. Hence, the two variants essentially complement each other and, as shown by the simulations, perform best when combined, by enabling a delivery rate close to 98%. Analogous performance is achieved by the push algorithm.

This behavior and the associated benefits can be better appreciated in the more challenging scenario considered in the right-hand side of Figure 3(a), where $\epsilon = 0.1$ yields a baseline delivery rate of 55%. Again, neither of the pull approaches alone is enough, but together they boost the delivery rate up to 90%, similar to what achieved by the push algorithm. Hence, in this scenario *the recovery phase performed with our algorithms is responsible for the delivery of almost half of the events being dispatched in the system.*

The effect of our algorithms is evident when topological reconfigurations occur. While in the scenario with lossy links errors are by and large uniformly spread, in the case of topological reconfigurations (over fully reliable links) they are instead concentrated around the time when the reconfiguration occurs. The left-hand side of Figure 3(b) shows a situation where reconfigurations occur every $\rho = 0.2s$, leading to a sequence of non-overlapping reconfigurations, i.e., the system reaches a stable state where subscription routes are correctly restored and hence events correctly delivered, before a new

[4]Hereafter, we assume that parameters whose value is not explicitly mentioned in the text are set to their default value, defined in Figure 2.

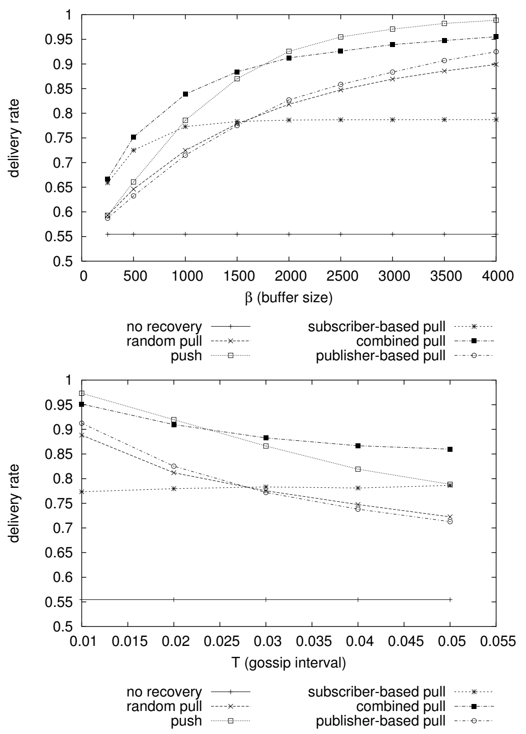


Fig. 4: *Effect of buffer size (top) and gossip interval (bottom) on delivery.*

link breaks. In this case, depending on where in the tree the reconfiguration actually occurs, the delivery rate may drop as low as 70%. All of our algorithms have a beneficial effect, by reducing the fraction of events lost. Nevertheless, again the push approach and the combined pull approach effectively "level" the delivery rate in proximity of 100%, by eliminating all the negative spikes corresponding to event loss.

The right-hand side of Figure 3(b) shows instead a scenario where reconfigurations occur every $\rho = 0.03s$ and hence are overlapping. Clearly, this is a very challenging scenario, that can be regarded as an approximation of the case where a non-leaf dispatcher is detached from the network and multiple links are broken at once. In any case, it defines a good, extreme test case for our analysis. It can be seen that the baseline delivery rate may drop as low as 60%. Again, our best algorithms cut all the negative spikes, and never fall below a 95% delivery rate. Hence, they introduce a high degree of robustness in the system, by masking to a great extent the perturbations caused by topological reconfiguration.

In the remainder of the section we focus on scenarios characterized by lossy links, since they represent the most general case. In particular, we set the error rate to the value of $\epsilon = 0.1$ to better appreciate the variations in performance determined by changes in the other parameter values.

C. Gossip Interval and Buffer Size

The key parameters of our epidemic algorithms are the gossip interval T, determining how frequently dispatchers communicate for the sake of recovery, and the size β of the buffers where events are cached. Figure 4 shows how changes in these parameters affect event delivery. The chart at the top shows simulations results for a buffer size β ranging from 500 to 4000 buffered events, which in our scenario translates into a time of persistence of an event in the buffer ranging between $1.3s$ and $9.2s$, against an overall simulation run time of $25s$. It is evident how subscriber-based pull alone cannot improve beyond a given limit. The reason for this behavior is the same we discussed earlier, and lies in the scarcity of dispatchers with the same pattern. The publisher-based and random pull approaches perform better than subscriber-based pull, but nevertheless exhibit a much slower convergence to 100% delivery. Again, push and the combined pull approach exhibit the best performance. Interestingly, combined pull has better performance than push with small buffers, while push approaches much faster 100% delivery as the buffer increases. This is easily explained by observing that the push approach relies more heavily on the persistence of events in the buffer. In fact, as known from the literature on epidemic algorithms [8], the push approach has a bigger recovery latency than pull. Moreover, in our push approach each gossip round involves only one of the potentially many patterns matching an event. Hence, the event recovery may involve several gossip rounds. Instead, the pull approach gossips more precise information about the lost event, and hence exhibits a smaller latency.

It is worth noting at this point that since the buffer capacity is a key factor in determining the performance of our algorithms, in all the simulations presented in this section we were very careful in setting the value of β, to minimize bias. In particular, as we discuss next, we increased linearly the buffer size together with the system scale. This is a rather conservative choice, since it is shown in the literature that buffer requirements grow as $O(fN \log N)$, being f the publish frequency. Moreover, we are currently investigating if and how some of the published results (e.g., [13]) that enable a significant buffer optimization are applicable in our context.

The chart at the bottom of Figure 4 shows instead how event delivery is affected by the gossip interval. The considerations that can be drawn are similar to those we made about the buffer size. Once more, subscriber-based pull has a limit at about 78%, while push and combined pull are the best solutions, with the former improving much faster as gossip rounds become

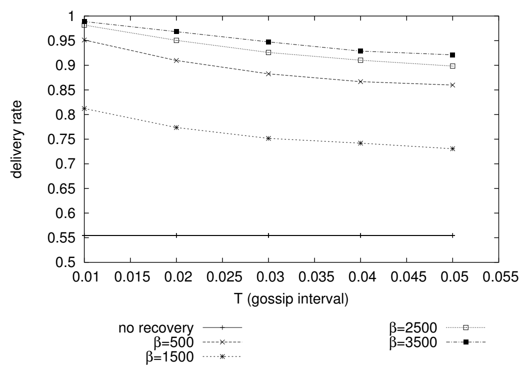


Fig. 5: *Effect on delivery of simultaneous changes to β and T (combined pull).*

more frequent, and the latter performing better as the interval between rounds increases.

The interplay between the two parameters is shown in Figure 5, where we plotted the event delivery obtained with the combined pull approach against the gossip interval, and varied the buffer capacity with increments of 1000 elements, starting with a buffer of 500. (Simulations of push show a similar behavior, and hence are not shown here.) The chart evidences a number of interesting phenomena. First, increments in the buffer size do not bear any significant impact after a given threshold. This is particularly evident when T is very small. Moreover, it can be seen how the sensitivity of our algorithms, and in particular of the combined pull approach considered in the figure, to changes in T is greater when the buffer size is smaller. This is evident from our previous discussion: when the buffer is big, less frequent gossip rounds are compensated by a longer persistence of events in the buffer.

D. Scalability

The charts we presented thus far were derived for an overlay network of $N = 100$ nodes. An open question is how an increase of N affects event delivery. The answer is in Figure 6. In each run we increased the number of dispatchers in the system and, to compensate for the increased scale, we also increased the buffer size accordingly, so that a given event persists in the buffer for a constant time (of about 4s). The simulation results show that our solutions exhibit good scalability with respect to the number of nodes. This is not surprising, given that this is a characteristic of epidemic algorithms, and one of the reasons why we chose this approach. Again, the best performance in terms of delivery is achieved by push and the combined pull approaches. The two pull solutions are more sensitive to scale when applied alone, with the publisher-based one being the best when the number of nodes is limited. The graph shows also that push becomes more convenient as the system size increases. Since the total number of possible patterns is kept constant in the chart, the introduction of new nodes increases the probability that a given pattern is gossiped, and hence an event recovered.

The system size, however, is not the only parameter characterizing scalability. In a content-based system, the distribution of patterns is another key factor. We evaluate this aspect by intervening on the maximum number π_{max} of patterns a dispatcher can be subscribed to. The effect of this parameter in terms of scalability can be grasped by looking at Figure 7, where π_{max} is plotted against the average number of subscribers that receive a single event. It can be seen how $\pi_{max} = 5$ is already sufficient to reach about 25% of dispatchers; this percentage raises to 80% with $\pi_{max} = 30$, essentially making communication more akin to a broadcast rather than a content-based one[5].

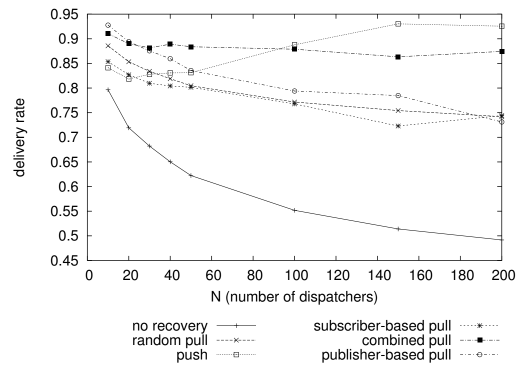


Fig. 6: *Delivery as the system size increases.*

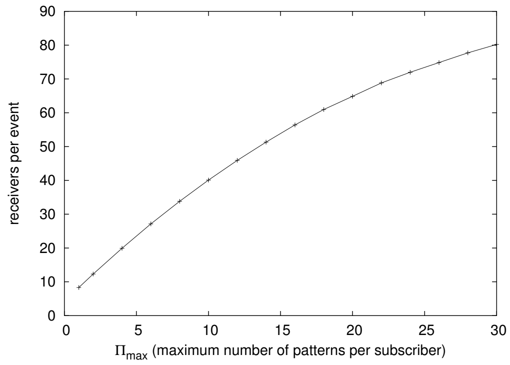


Fig. 7: *Number of dispatchers receiving an event as the number π_{max} of subscriptions per dispatcher increases.*

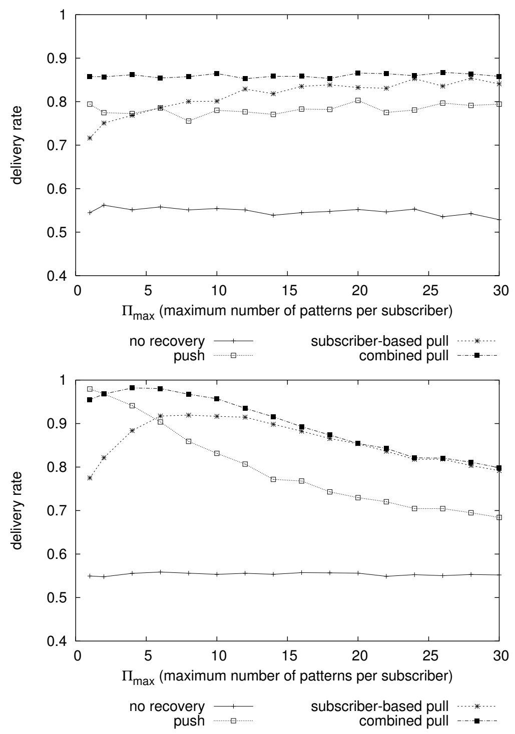
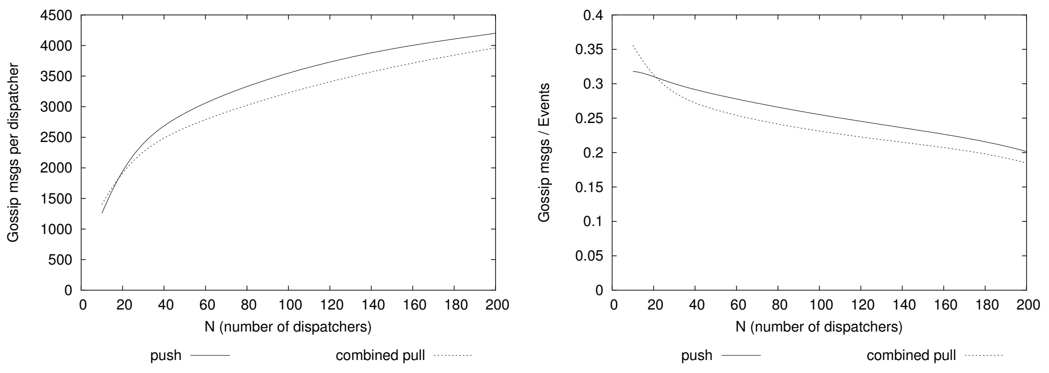
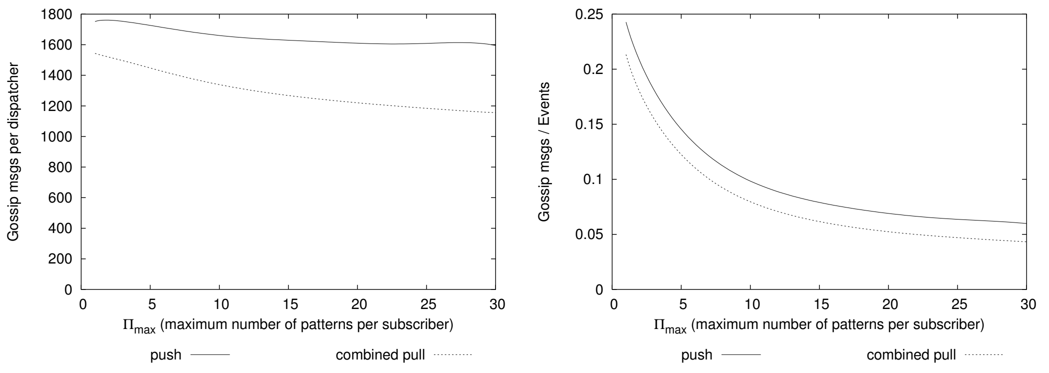


Fig. 8: *Delivery as the number of subscribers per pattern increases, under a low (top) and high (bottom) publish load.*

[5]All of our simulations assume that an event can match at most 3 patterns. In a content-based system this is a quite conservative assumption, since the need for a single tree is motivated precisely by the fact that a single event is likely to match several patterns. A higher matching rate would make the curve in Figure 7 even steeper; additional simulations we ran show how this noticeably improves further the performance of our algorithms.



(a) Overhead vs. the system size N.



(b) Overhead vs. the number π_{max} of subscriptions per dispatcher.

Fig. 9: *Overhead introduced by gossip: in absolute terms (left) and relative to the events in the system (right).*

The impact of π_{max} on the delivery rate is then analyzed in Figure 8, under different publishing loads. The chart at the top shows how, under a low publish rate of 5 publish/s, the delivery rate of push and combined pull is basically unaffected by increases in π_{max}, with the former performing slightly better than the latter. The subscribed-based pull improves a little since more dispatchers are now caching an event. The chart at the bottom, derived under the usual high publish rate of 50 publish/s, shows a more interesting behavior. For a small number of subscriptions per dispatcher, about $\pi_{max} < 6$, combined pull improves delivery, while push makes it worse. This is explained by observing that the push approach proceeds by gossiping about a pattern at a time: the higher the number of patterns, the higher the number of gossip rounds potentially required to recover an event, and the higher the likelihood that the event is actually discarded from all the caches before being recovered—especially under a high publish load. Instead, in a pull approach the increase in the number of subscribers is beneficial, since it increases the probability to contact a dispatcher that has actually cached the event. For $\pi_{max} > 6$, instead, performance decreases significantly for all solutions. This can be explained by noting that both charts in Figure 8 were derived with a buffer size $\beta = 4000$. Since the number of subscriptions per dispatcher increases, each subscriber receives more events: this value of β is more than enough for a low publishing load, but it is insufficient to keep up with a high publishing load—hence the decrease in performance.

E. Overhead

After we verified that our solutions significantly improve event delivery even when the system scale increases, the next question is about the overhead they introduce. Moreover, overhead can be regarded as another facet of scalability. Figure 9 contains the results of our evaluation. It considers the system size N and the number π_{max} of subscriptions per dispatcher as a measure of scalability, similarly to what we did in Section IV-D. The overhead is presented in two ways: as the number of gossip messages sent by each dispatcher, to evaluate the overhead on the single dispatcher, and as the ratio between the gossip and event messages dispatched in the overall system, to evaluate the impact of gossip on the overall bandwidth available to event dispatching.

The left-hand side of Figure 9(a) shows that the number of gossip messages sent by each dispatcher as the dispatching network grows increases with the scale of the system, but well below a linear trend. This very desirable behavior is a direct consequence of the decentralized nature of gossip algorithms: the local effort of a dispatcher, in term of gossip messages sent, is independent from the system size. Hence, the growth of gossip traffic is proportional to the number of hops made by each gossip message, which in our case increases logarithmically. The right-hand side of Figure 9(a) shows instead that the traffic caused by event forwarding rises faster than the one caused by gossip—under the assumption of continuous publishing. Again, this is a desirable property of our algorithms that leads to high scalability. It can be explained by noting that while event forwarding essentially implements a multicast scheme that must reach all the recipients, gossip involves only a fraction of them; moreover, the propagation of a single message is often "short-circuited" by the first dispatcher that owns the requested message.

Figure 9(b) shows the impact on overhead of the number π_{max} of subscriptions per dispatcher. The overhead on a single dispatcher, shown in the left-hand side, is only marginally affected by π_{max}. The overhead decreases a little for increasing values of π_{max}, which can be explained by observing that an increase in the number of patterns increases the number of dispatchers at which an event gets cached, and hence increases the likelihood of retrieving the event close to the gossiper. Instead, the ratio of gossip messages and event messages in the system decreases significantly with the increase of π_{max}, which is a desirable property. The reason can be grasped by looking back at Figure 7. An increase in π_{max} determines an increase in the number of receivers, hence the number of events dispatched in the system rises much faster than the number of gossip messages, especially in the scenario with high publishing load we considered.

It is worth pointing out some additional issues related with overhead. First, we note that the overhead against the system scale might look relatively high, since in Figure 9 it ranges from about 28% for 40 nodes, down to about 20% for 200 nodes. Still, it should be remembered that we performed these simulations in extremely challenging scenarios, where the system load is very high and so is the chance of losing an event. Given this tough setting and the remarkable improvement in event delivery, the overhead plotted in Figure 9 does not seem unreasonable. In general, however, the tradeoffs between overhead and event delivery are essentially determined by the application and networking scenario at hand, and can be tuned appropriately by intervening on the gossip interval and buffer size whose impact has been described in Section IV-C.

Moreover, if the assumptions about load and error rate are made less challenging, the relative performance of the push and pull approaches changes significantly, as the reactive pull approach triggers communication only when a recovery is needed while the proactive push approach gossips continuously, and hence may result in wasted bandwidth. This behavior is shown in Figure 10, which plots the total number of gossip messages sent against the error rate. The top chart plots the overhead at the high publish rate of 50 publish/s, while the bottom plot is derived with 5 publish/s. In the latter case, the pull approach clearly wastes less bandwidth, especially when communication is more reliable: from the chart it can be seen that when $\epsilon = 0.01$, corresponding to a baseline delivery rate of 95%, pull overhead is one third of push. The pull approach, in this case, may skip some gossip rounds due to fact that no event has been detected as lost in the meantime, while a push approach must proactively push at each gossip round. To remove the potential source of inefficiency of the push

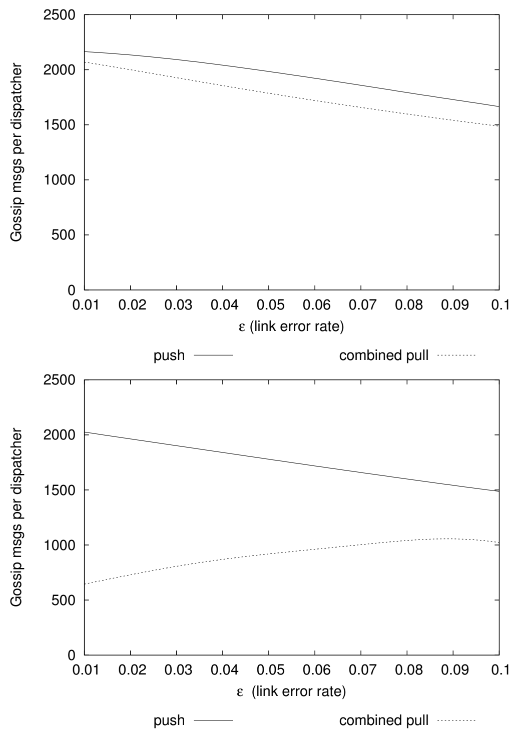


Fig. 10: *Overhead under a high (top) and low (bottom) publishing load.*

algorithm, an adaptive approach could be exploited where the gossip interval T is changed dynamically according to the current state of the system, as suggested in [14].

In general, note also that in these simulations we assumed that the size of event and gossip messages is the same. Hence, the plots actually show only an upper bound for overhead: the size of gossip messages is likely to be in reality much smaller than the one of event messages, bringing the relative overhead below the curves shown.

Finally, another issue is the one of computational overhead, which we did not investigate through our simulations. Qualitatively, the pull-based solutions require that, when an event e is published by a dispatcher, the latter performs a match of e against *all* the patterns in its subscription table. This is more than normally required, since usually the match processing needed to route a message towards a neighbor stops as soon as the first matching pattern is found. While we are currently investigating optimizations to limit this overhead, we also observe that only the publisher experiences it: the other dispatchers route events according to the normal processing.

V. Related Work

Several centralized publish-subscribe systems (e.g., all the JMS [15] compliant) and also some of the existing distributed subject-based publish-subscribe systems provide a reliable service [5], [16]–[19]. Similarly, researchers working on reliable multicast and group communication proposed several protocols for reliable multicast where routing is group or subject-based. Unfortunately, none can be used for the systems we target here, due to the peculiarity of content-based routing and of the scenarios we target.

Little research work addresses reliability in content-based publish-subscribe systems. In [20], the authors describe a guaranteed delivery service for the Gryphon system. Content-based routing is provided through a collection of spanning trees each rooted at one of the publishers. Guaranteed delivery is ensured by an acknowledgment-based scheme that requires stable storage only at the publisher. However, the solution described is not amenable to the highly dynamic scenarios motivating our work, since the solutions for facing a publisher crash (e.g., shared and replicated logs) are not applicable, and a topological change would trigger a high-overhead reconfiguration of several trees. Instead, Hermes [21] provides a form of content-based routing based on constraints on type attributes. It exploits Pastry [22] as the basic transport layer, and hence inherits the ability to tolerate topological changes. Nevertheless, the authors do not give details about how to recover events lost during reconfiguration.

The closest match to our work is *hpcast* [10]. In hpcast nodes are organized in a hierarchy where the leaves represent event subscribers and publishers, and intermediate nodes represent *delegates*, i.e., special nodes which are chosen to represent aggregate interests of their children. A gossip push approach is used to distribute events starting from the root of the hierarchy and moving down each time a delegate retrieves an event that could interest its children. The idea of using gossip not just to improve event delivery but as the only routing mechanism is simple and elegant, but suffers from several drawbacks. First, in absence of faults it increases the overhead since events are not routed only to interested nodes, but they can reach also non-interested nodes or even be sent more than once to the same node. Second, even in absence of faults it does not guarantee that events are delivered correctly. Third, it forces the adoption of a push approach in which gossip messages include the entire event content instead of a simple digest, thus further increasing the network traffic. Finally, the nodes near to the root of the hierarchy experience high traffic, and hence must maintain very big event caches to increase the probability of correctly delivering events.

VI. Conclusions

Modern distributed computing fosters scenarios that are increasingly large scale, unreliable, and highly dynamic. Distributed content-based publish-subscribe is emerging as an effective tool to tackle the new challenges. Nevertheless, reliable event delivery, a fundamental requirement in the new distributed scenarios, has largely been ignored by researchers.

In this paper, we provided a thorough evaluation of an approach to reliability based on epidemic algorithms. Simulations show that our use of epidemic algorithms improves significantly event delivery, is scalable, and introduces only limited overhead. Our results are derived without making assumptions about the source of event loss, and hence enjoy general applicability. Our ongoing work aims at complementing the results we described here with those we enabling reconfiguration of the dispatching infrastructure [7] and convey them in a new generation of distributed content-based publish-subscribe systems able to tolerate topological reconfigurations and minimize event loss.

References

[1] P. Costa, M. Migliavacca, G.P. Picco, and G. Cugola, "Introducing Reliability in Content-Based Publish-Subscribe through Epidemic Algorithms," in *Proc. of the* 2^{nd} *Int. Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003, To appear. Available at www.eecg.utoronto.ca/debs03.

[2] A. Carzaniga, D. Rosenblum, and A. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Trans. on Computer Systems*, vol. 19, no. 3, pp. 332–383, 2001.

[3] P. Sutton, R. Arkins, and B. Segall, "Supporting Disconnectedness—Transparent Information Delivery for Mobile and Invisible Computing," in *Proc. of the IEEE Int. Symp. on Cluster Computing and the Grid*, May 2001.

[4] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward, "Gryphon: An information flow based approach to message brokering," in *Int. Symp. on Software Reliability Engineering*, 1998.

[5] L. F. Cabrera, M. B. Jones, and M. Theimer, "Herald: Achieving a global event notification service," in *Proc. of the* 8^{th} *Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001, pp. 87–94.

[6] G. Cugola, E. Di Nitto, and A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE Trans. on Software Engineering*, vol. 27, no. 9, 2001.

[7] G.P. Picco, G. Cugola, and A.L. Murphy, "Efficient Content-Based Event Dispatching in the Presence of Topological Reconfigurations," in *Proc. of the* 23^{rd} *Int. Conf. on Distributed Computing Systems (ICDCS)*, 2003, pp. 234–243.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," *Operating Systems Review*, vol. 22, no. 1, pp. 8–32, 1988.

[9] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM Trans. on Computer Systems*, vol. 17, no. 2, pp. 41–88, 1999.

[10] P. Eugster and R. Guerraoui, "Probabilistic multicast," in *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'02)*, June 2002.

[11] D.S. Rosenblum and A.L. Wolf, "A Design Framework for Internet-Scale Event Observation and Notification," in *Proc. of the* 6^{th} *European Software Engineering Conf. (ESEC/FSE)*. Sept. 1997, LNCS 1301, Springer.

[12] A. Varga, "OMNeT++ Web page," www.omnetpp.org.

[13] O. Ozkasap, R. van Renesse, K. Birman, and Z. Xiao, "Efficient Buffering in Reliable Multicast Protocols," in *Proc. of Int. Workshop on Networked Group Communication (NGC99)*, Nov. 1999.

[14] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen, "PlanetP: Infrastructure support for P2P information sharing," Tech. Rep. DCS-TR-465, Rutgers University, Nov. 2001.

[15] Sun Microsystems, Inc., *Java Message Service Specification Version 1.1*, April 2002.

[16] TIBCO Inc., *TIBCO Rendezvous*, www.rv.tibco.com.

[17] Real-Time Innovations, Inc., *NDDS: Network Middleware for Distributed Real Time Applications*, www.rti.com.

[18] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination," in *Proc. of the* 11^{th} *Int. Workshop on Network and Operating System Support for Digital Audio and Video*, June 2001.

[19] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE J. on Selected Areas in Communications*, vol. 20, no. 8, 2002.

[20] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach, "Exactly-once Delivery in a Content-based Publish-Subscribe System," in *Proc. of the Int. Conf. on Dependable Systems and Networks*, 2002, pp. 7–16.

[21] P. R. Pietzuch and J. M. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," in *Proc. of the Int. Workshop on Distributed Event-Based Systems (DEBS'02)*, July 2002, pp. 611–618.

[22] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Int. Conf. on Distributed Systems Platforms (Middleware)*, Nov. 2001, LNCS 2218, pp. 329–350.