# 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 [5] are 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.*

## 1. Introduction

Publish-subscribe middleware is emerging as a promising tool to tackle the demands of modern distributed computing. In particular, distributed content-based systems [3, 4, 7, 15, 17] 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. This goal demands the solution of several problems. In [11] we tackled the efficient reconfiguration of subscription information, required to restore event routing. The topic of this paper, instead, is the complementary problem of recovering events lost during reconfiguration and, in general, improving reliability. In [5] we described three solutions based on epidemic algorithms [2, 8]. Here, we complete and validate this initial proposal by thoroughly evaluating its effectiveness in challenging unreliable scenarios.

The contribution put forth by this paper is relevant under many respects. Our algorithms, whose effectiveness and efficiency we quantitatively demonstrate in this paper, provide a viable solution for recovering events lost during reconfiguration. Moreover, they do not rely on any assumption about the source of event loss, therefore they enjoy general applicability towards improving reliability in content-based publish-subscribe systems. Finally, epidemic algorithms have been applied to a number of domains but, with the exception of [9], never 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 2 is a concise overview of content-based publish-subscribe systems. Section 3 describes the epidemic algorithms we originally proposed in [5] for achieving reliability in content-based publish-subscribe systems. An extensive evaluation of these algorithms, based on simulation, is the subject of Section 4 and the core contribution of this paper. Finally, Section 5 places our work in the context of related research efforts, and Section 6 ends the paper with brief concluding remarks.

## 2. Content-Based Publish-Subscribe

A large number of publish-subscribe middleware exist, which differ along several dimensions[1]. Two are usually considered fundamental: the architecture of the event dispatcher and the expressiveness of the subscription language. The former can be either centralized or distributed. The latter draws a line between *subject-based* systems, where subscriptions identify classes of events belonging to a given channel or subject, and *content-based* systems, where subscriptions contain expressions (called *event patterns*), which enable sophisticated matching on the event content.

In this paper, we consider distributed content-based systems. A set of *dispatching servers*[2], as shown in Figure 1, are connected in an overlay network and 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 archi-

1 For more detailed comparisons see [4, 7, 13].
2 Hereafter, 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.

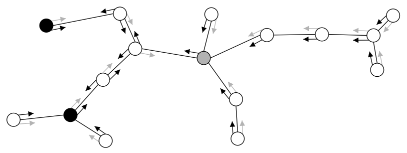


**Figure 1. A dispatching network with subscriptions laid down according to a subscription forwarding scheme.**

tecture can be further classified according to the interconnection topology of dispatchers, and the strategy for routing subscriptions and events. In this work we consider a subscription forwarding scheme [4] on an unrooted tree topology, as this choice covers the majority of existing systems.

In this approach, subscriptions are delivered to every dispatcher along a single unrooted tree overlay network connecting all the dispatchers, and are used to establish the routes followed by published events. 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. The subscription is then propagated by the dispatcher—which now behaves as a subscriber w.r.t. the rest of the dispatching network—to all of its neighboring dispatchers on the overlay network. These, in turn, record the subscription and repropagate it towards all of their neighbors, 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 similarly to subscriptions, although at each hop entries in the subscription table are removed rather than inserted.

Hereafter, we ignore 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. Figure 1 shows a system where two dispatchers are subscribed to a "black" pattern, and one to a "gray" pattern. Arrows denote the routes laid down according to these subscriptions, and reflect the content of each dispatcher's subscription table. Solid lines are links of the tree overlay network. As a consequence of subscription forwarding, 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 trees, typical of subject-based systems, would lead to inefficient event duplication.

## 3. 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.

### 3.1. Epidemic Algorithms

The idea behind epidemic (or *gossip*) algorithms [2, 8] 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 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. Instead, in a *pull* style each process solicits the transmission of information from other processes. Usually a push style of communication uses gossip messages containing a *positive* digest of the system state to be disseminated, while a pull approach exploits *negative* digests, and gossip messages hence contain the portion of the state that is known to be missing.

Independently from the scheme adopted, epidemic algorithms enjoy many desirable properties, thanks to their probabilistic and decentralized nature. 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. Finally, they are usually very simple to implement and rather inexpensive to run. Therefore, epidemic algorithms appear as good candidates for the dynamic distributed scenarios we target, although their exploitation for recovering lost events in content-based publish-subscribe system is not straightforward, as discussed in remainder of this section.

### 3.2. Our Approach

In our solutions, the state to 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 determining their routing. Moreover, an event may match multiple subscriptions, instead of a single group. These characteristics make it difficult to identify the subset of dispatchers that may hold missing events, and prevent a direct use of solutions already developed for the aforementioned domains. This section presents three epidemic algorithms designed for content-based publish-subscribe systems. Presentation is kept concise, as the emphasis of this paper is on the algorithms' evaluation. The interested reader can find more details, including a formalization, in [5].

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 along 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 w.r.t. the normal publish-subscribe operations. Hence, we assume the existence of a unicast transport layer 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 of the 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 w.r.t. 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 $P_{fwd}$. To increase the chance of eventually finding all the dispatchers interested in the cached events, and thus speed up convergence, $p$ is selected by considering the whole subscription table instead of only the subscriptions issued locally to the gossiper.

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 it already received. The identifiers of the missed events are included in a request message sent to the gossiper, which replies by sending a copy of the 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, and therefore a reactive pull with negative digests may be preferable. Nevertheless, this requires the ability to detect lost messages. 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 the 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 [5], this problem can be solved by tagging each event with enough information to enable loss detection. In this scheme the event identifier 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—an opportunity enabled by subscription forwarding, where subscriptions are known to all dispatchers. Event loss is detected when a dispatcher receives an event matching a pattern $p$ whose sequence number, associated to $p$ in the event identifier, is greater than the one expected for $p$ from that event source.

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

In [5] we defined two algorithms that rely on this kind of detection but use different routing strategies: the first one steers gossip messages towards the event subscribers, while the other steers them towards the event publisher.

- *Subscriber-Based Pull.* In this scheme, when a dispatcher detects a lost event it inserts the corresponding information (i.e., source, matched pattern, and sequence number associated to 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. Unlike with 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 similar to the push solution. A dispatcher receiving the gossip message checks its cache against events requested by the gossiper and, if any are found, sends them back to it. Note how, in this case, the dispatcher need not be a subscriber for the pattern $p$ specified by the gossiper. The dispatcher could have received the gossip message because it sits 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 that published events are cached not only by the dispatchers that received them but also by the source, and that the address of each dispatcher encountered on the route towards a subscriber is appended to the event message. The algorithm behaves similarly to the previous one, but it routes gossip messages towards publishers instead of subscribers. While $Lost$ contains the same information as before, a new buffer $Routes$ is needed 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, gossip messages are distinguished based on the event source rather than the pattern, and augmented with the information necessary to be routed back to the publisher, as found in $Routes$. As the topology of the dispatching network may change, there is no guarantee that the route in $Routes$ is the same originally followed by the missing event. However, it is likely that the two share some portion or, in the worst case, the publisher.

## 4. Evaluation

As mentioned in Section 1, our initial and driving motivation for tackling reliability was to cope with event loss in-

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

**Figure 2. Simulation parameters and their default values.**

duced 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 improve reliability in any situation where an event loss may occur. Consequently, in this section we evaluate the performance of our algorithms by considering both the scenario where events are lost because of changes in the topology of the overlay network and the more common scenario of a stable topology with lossy links.

Besides considering each algorithm in isolation, we also evaluate the performance of the combination of the two pull approaches, as it enables a significant performance improvement. Moreover, to evaluate the effectiveness of our strategies to route gossip messages, we also compare them against a pull approach where such routing is entirely random. Simulations of a similar random push approach are omitted as their performance is extremely poor.

Section 4.1 describes the simulation setting, while Section 4.1 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 simulations focus on the former scenario, as it is more general and its effects more easily isolated and controlled.

## 4.1. 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, 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 [11].

- *Events, subscriptions, and matching.* Events are represented as randomly-generated sequences of 3 characters (out of a total of 70), while event patterns are represented as a single character. An event matches a pattern if it contains the corresponding character. Each dispatcher can subscribe to at most $\pi_{max}$ different event patterns.
- *Publish rate.* Dispatchers continuously publish events on a network with stable subscription information, i.e., no (un)subscriptions are being issued. 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.* Each dispatcher is connected with at most other four in the dispatching tree. Clients are not modeled, as their activity ultimately affects only the dispatcher they are attached to.

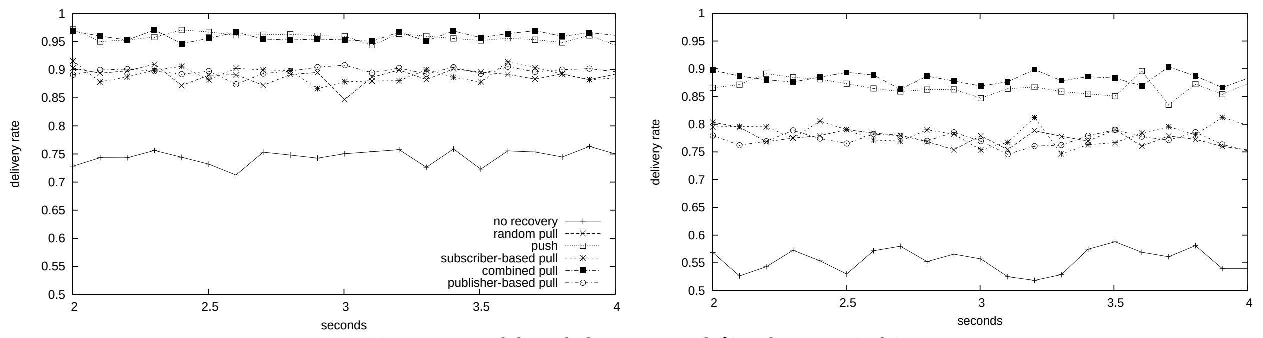
**Modeling the sources of event loss.** The relevant parameters differ according to the unreliable scenario considered.

- *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 45% of events lost) and $\epsilon = 0.05$ (leading to a 25% 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 [11], where the interested reader can find more details. A reconfiguration in this setting is the breakage of a link, followed by its replacement with another that maintains the tree connected. We assume that the overlay network is repaired in $0.1s$. The interval $\rho$ separates 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.

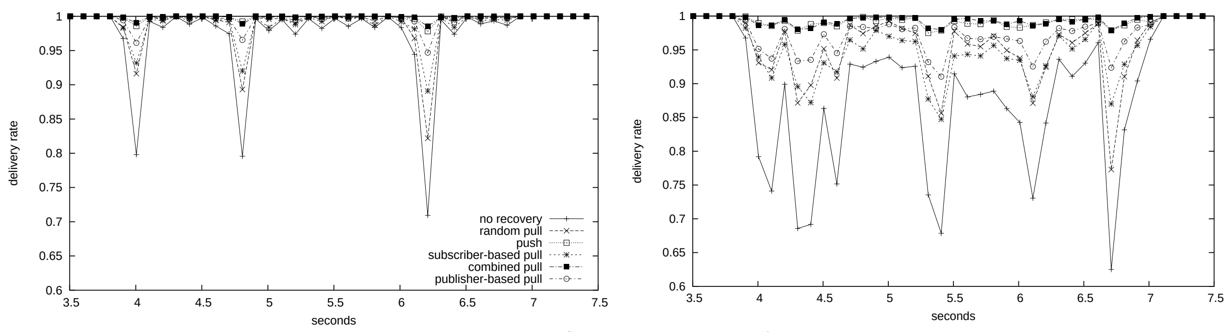
**Gossip parameters.** Gossiping is ruled by the following:

- *Buffer size.* We adopt a simple FIFO buffering strategy where each dispatcher caches only the events for which it is either the publisher or a subscriber. The buffer has a size of $\beta$ elements.
- *Gossip interval.* The frequency of gossiping is controlled by the gossip interval $T$ between two gossip rounds.
- *Combining pull approaches.* As mentioned, we combined the two pull approaches to improve performance. Which approach is used at a given moment is determined by the probability $P_{src}$.

**Simulation tool.** Our simulations are developed using OMNeT++ [18], an open source discrete event simulation tool.



(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).

**Figure 3. Event delivery.**

### 4.2. Event Delivery

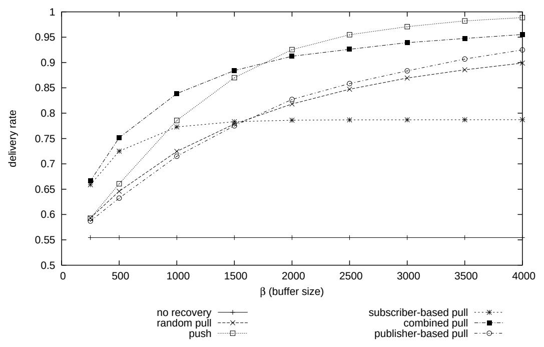
In this section we evaluate the effectiveness of our approach in improving event delivery. The left-hand side of Figure 3(a) compares the performance of the various solutions in the case of a stable system with lossy links, whose error rate[4] is $\epsilon = 0.05$. The performance metric we choose is the delivery rate, i.e., the ratio between the number of events correctly received by a dispatcher and those that would be received in a fully reliable scenario. The delivery rate in the chart is averaged, and shown in percentage. Simulation time is on the $x$-axis.

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—a publisher-based is more convenient in this case. Nevertheless, in a situation with many dispatchers subscribed to the same pattern a publisher-based approach is less appealing, since gossip involves a much smaller fraction of the dispatchers. Therefore, 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 concentrated around the time when the reconfiguration occurs. In the left-hand side of Figure 3(b) reconfigurations occur every $\rho = 0.2s$, leading to a sequence of non-overlapping reconfigurations, i.e., the system stabilizes with correct routes before a new link breaks. Depending on where disruption occurred, 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, push and combined pull "level" the delivery rate in proximity of

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.



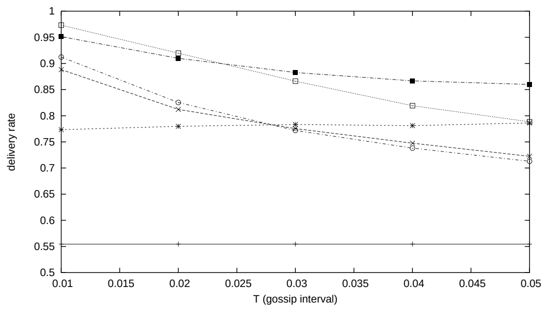


**Figure 4. Effect of buffer size (top) and gossip interval (bottom) on delivery.**

100%, by removing 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 are therefore overlapping. 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. 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.

## 4.3. 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 $\beta$ of the buffers where events are cached. Figure 4 shows how changes in these parameters affect event delivery. In the top chart, $\beta$ ranges 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, i.e., 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], push 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. Therefore, 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 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 $\beta$, to minimize bias. In particular, 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., [10]) 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. The interplay between the two parameters is shown in Figure 5, where we plot against $T$ the event delivery obtained with the combined pull approach, and vary $\beta$ with increments of 1000 elements, starting with

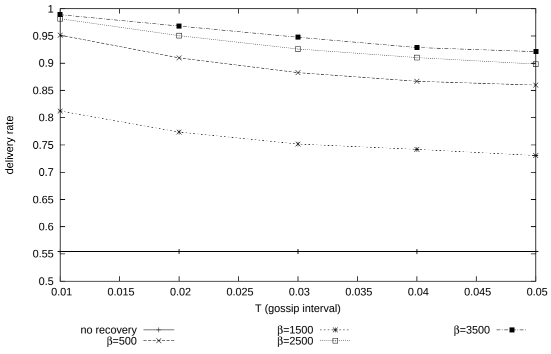


**Figure 5. Effect on delivery of simultaneous changes to $\beta$ and $T$ (combined pull).**

a buffer of 500. (Simulations of push show a similar behavior, and hence omitted.) 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 $\beta$ 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.

## 4.4. Scalability

The charts presented thus far are based on an overlay network of $N = 100$ dispatchers. An open question is how an increase of $N$ affects event delivery. The answer is in Figure 6. In each run we increased $N$ and, to compensate for the increased scale, we also increased $\beta$ 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 w.r.t. the number of dispatchers. This is not surprising, as this is a characteristic of epidemic algorithms and a motivation for our 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 one when $N$ is small. 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 dispatchers 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, which we evaluate by intervening on the maximum number $\pi_{max}$ 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 $\pi_{max}$ 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 20% 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].

The impact of $\pi_{max}$ on the delivery rate is then ana-

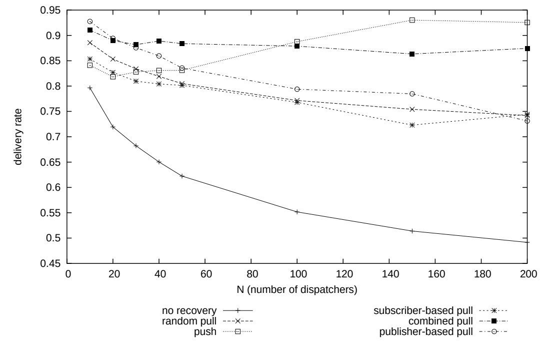


Figure 6. Delivery as the system size increases.

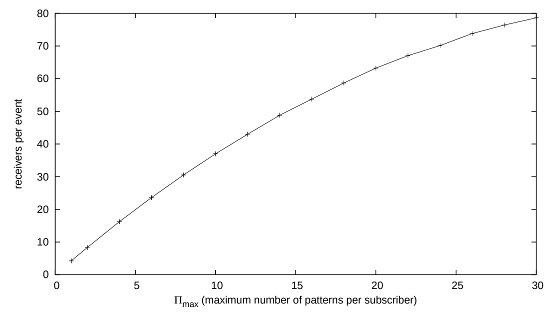


Figure 7. Number of dispatchers receiving an event as the number of subscriptions per dispatcher increases.

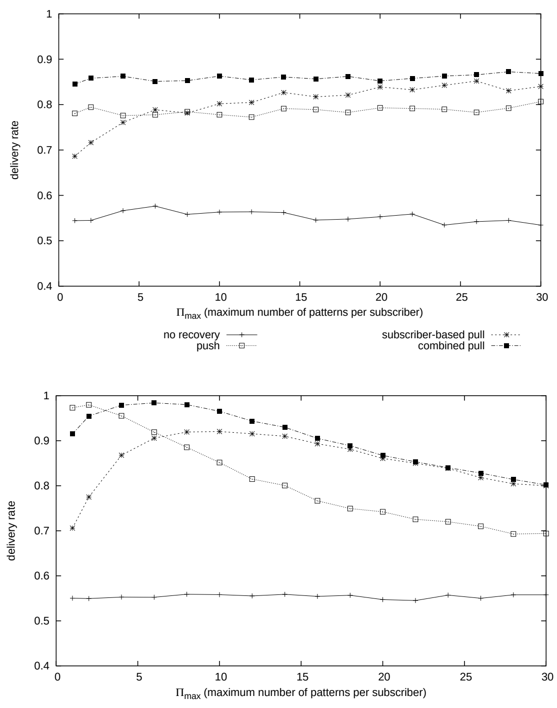
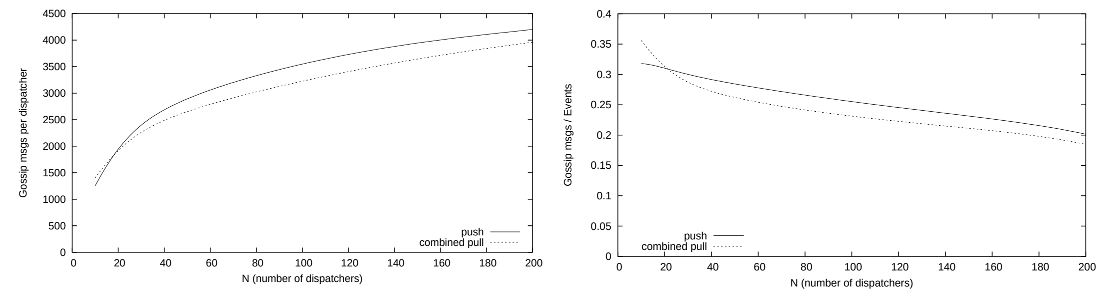
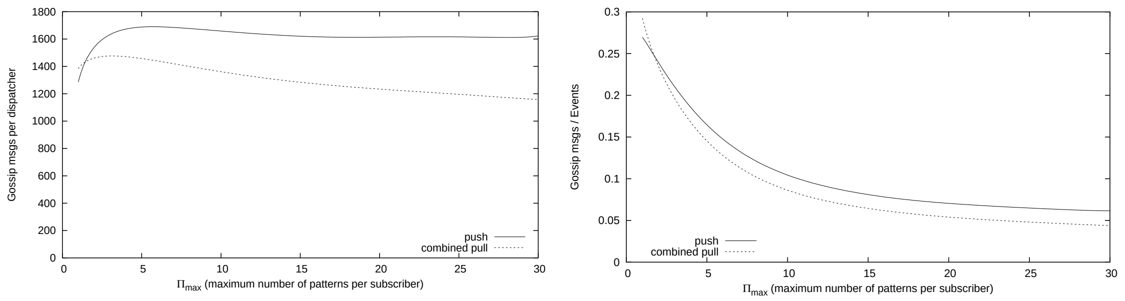


Figure 8. Delivery as the number of subscribers per pattern increases, under a low (top) and high (bottom) publish load.

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 $\pi_{max}$ of subscriptions per dispatcher.

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

lyzed in Figure 8, under different publishing loads. The top chart 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 $\pi_{max}$, with the former performing slightly better than the latter. Subscribed-based pull improves a little since more dispatchers are now caching an event. The bottom chart, 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 push evolves by gossiping about a pattern at a time: the higher the number of patterns, the higher the number of gossip rounds 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 actually cached the event. For $\pi_{max} > 10$ performance decreases significantly for all solutions. This is reasonable since 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 $\beta$ 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.

## 4.5. 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. Figure 9 contains the results of our evaluation. It considers the system size $N$ and the number $\pi_{max}$ of subscriptions per dispatcher as a measure of scalability, as in Section 4.4. 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 $N$ 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 holding the requested message.

Figure 9(b) shows the impact of $\pi_{max}$ on overhead. For

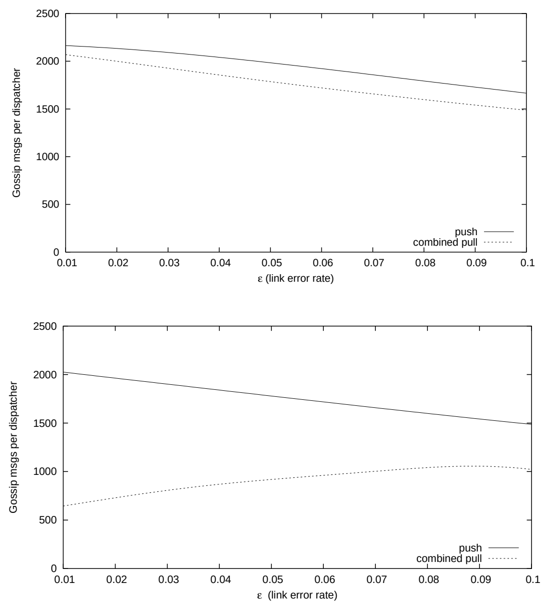


**Figure 10. Overhead under a high (top) and low (bottom) publishing load.**

a single dispatcher, overhead is only marginally affected, as shown in the left-hand side. It decreases a little for increasing values of $\pi_{max}$, 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 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 $\pi_{max}$, which is a desirable property. The reason can be grasped by looking back at Figure 7. An increase in $\pi_{max}$ determines an increase in the number of receivers, therefore 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, one could argue that the overhead vs. system scale ratio might look quite high. In Figure 9 it ranges from about 28% for $N = 40$, down to about 20% for $N = 200$. However, we remind the reader that our simulation scenarios are extremely challenging, as 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 in Figure 9 does not seem unreasonable. In any case, 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 we described in Section 4.3.

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 fact is shown in Figure 10, where the total number of gossip messages sent is plot against the error rate. The publish rate is 50 publish/s in the top chart, and 5 publish/s in the bottom one. In the latter case, the pull approach clearly wastes less bandwidth, especially when communication is more reliable: from the chart, when $\epsilon = 0.01$—corresponding to a baseline delivery rate of 95%—pull's 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 this potential source of inefficiency of the push 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 [6].

In general, in our simulations we assumed that the size of event and gossip messages is the same. Therefore, our results are only an upper bound for overhead: in reality, gossip messages are likely to be much shorter than event messages, bringing the relative overhead below the curves shown.

Finally, in our simulations we did not investigate computational overhead. 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 the matching process needed to route a message towards a neighbor usually stops as soon as the first matching pattern is found. We are currently investigating optimizations to limit this overhead. However, we also observe that only the publisher experiences it: the other dispatchers route events according to the normal processing.

## 5. Related Work

Several centralized publish-subscribe systems (e.g., all JMS [16] compliant ones) provide a reliable service. Also, several protocols exist for reliable multicast and group communication. Unfortunately, none of these results can be used for the systems we target here, due to the peculiarity of content-based routing and of the scenarios we consider.

Few works address reliability in content-based publish-subscribe systems. In [1], 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 exploits an acknowledgment-based scheme requiring stable storage at the publisher. This approach is not amenable to the dynamic scenarios motivating our work, where the solutions to deal with a publisher crash (e.g., shared and replicated logs) are impractical, and a topological change would trigger a high-overhead reconfiguration of many trees. Hermes [12] provides content-based routing based on constraints on type attributes and exploits Pastry [14] as the transport layer, thus inheriting the ability to deal with topological changes. Unfortunately, the authors do not address the recovery of events lost during these changes.

The closest match to our work is *hpcast* [9]. In this system, nodes are organized in a hierarchy where leaves represent event subscribers and publishers, and intermediate nodes represent *delegates*, i.e., nodes chosen to represent the aggregate interests of their children. Events are distributed through gossip push starting at the root, and moving downwards each time a delegate retrieves an event of interest for its children. The idea of using gossip for routing and recovery is simple and elegant, but suffers from several drawbacks. First, in absence of faults it *never* guarantees delivery, and it increases overhead since events are not routed only to interested nodes and can even be sent more than once to the same one. Second, it forces the adoption of a push approach where gossip messages include the entire event content instead of a simple digest, further increasing the network traffic. Finally, the nodes close to the root experience high traffic, and therefore must keep big event caches to increase the probability of event delivery.b

## 6. Conclusions

Modern distributed computing fosters scenarios that are large scale, unreliable, and highly dynamic. Distributed content-based publish-subscribe is emerging as an effective tool to tackle these challenges. However, reliable event delivery, a fundamental requirement in the new distributed scenarios, has largely been ignored thus far 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 do not rely on assumptions about the source of event loss, and therefore enjoy general applicability. Our ongoing work aims at complementing the results described here with those obtained for the reconfiguration of the dispatching infrastructure [11], and conveying them in a new generation of distributed content-based publish-subscribe systems able to tolerate topological reconfigurations and minimize event loss.

## References

[1] 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*, pages 7–16, 2002.

[2] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. on Computer Systems*, 17(2):41–88, 1999.

[3] 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)*, pages 87–94, May 2001.

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

[5] P. Costa, M. Migliavacca, G. 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`.

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

[7] 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*, 27(9):827–850, Sept. 2001.

[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*, 22(1):8–32, 1988.

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

[10] 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.

[11] 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)*, pages 234–243, 2003.

[12] 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)*, pages 611–618, July 2002.

[13] D. 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)*, LNCS 1301. Springer, Sept. 1997.

[14] 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)*, LNCS 2218, pages 329–350, Nov. 2001.

[15] 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.

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

[17] 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.

[18] A. Varga. OMNeT++ Web page. `www.omnetpp.org`.