Introducing Reliability in Content-Based Publish-Subscribe through Epidemic Algorithms

Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola
Dip. di Elettronica e Informazione, Politecnico di Milano
P.za Leonardo da Vinci 32, 20133 Milano, Italy
{costa, migliava, picco, cugola}@elet.polimi.it

ABSTRACT

Distributed content-based publish-subscribe middleware provides the necessary decoupling, flexibility, expressiveness, and scalability required by modern distributed applications. Unfortunately, this middleware usually does not provide reliability, especially in the presence of highly reconfigurable scenarios. Indeed, this problem has been thus far largely disregarded by the research community and solutions developed in other contexts are not immediately applicable.

In this paper, we tackle the problem of introducing reliability in content-based publish-subscribe in dynamic environments by exploiting epidemic algorithms, whose characteristics in terms of decentralization, scalability, and resilience to topological changes resonate with our problem.

1. INTRODUCTION

Publish-subscribe middleware has recently become popular because of its asynchronous, implicit, multi-point, and peer-to-peer style of communication. Components in a publish-subscribe system are strongly decoupled: they can be easily replaced, thus providing a high degree of flexibility both at the application and infrastructure level. A number of publish-subscribe systems have been proposed to date. In this paper we focus on those that seek increased scalability and flexibility by exploiting a distributed architecture for event dispatching, and that empower the programmer with maximum expressiveness by using a content-based scheme for determining the match between an event and a subscription. Representative examples are [4, 29, 27, 3, 10].

Although the publish-subscribe *model* enjoys a growing popularity, we observe that the characteristics of the available *systems* still fall short of expectations under many respects. For instance, this paper is motivated by the observation that the reliability of the distributed event dispatching infrastructure is rarely guaranteed by dedicated mechanisms: instead, it is typically delegated to the underlying transport protocol, e.g., by assuming the existence of TCP links. Unfortunately, this approach is overly restraining in several scenarios, including simple ones characterized by small scale and a static network topology. For instance, communication can be implemented on top of unreliable transport protocols like UDP for performance reasons; moreover, links and nodes of the dispatching infrastructure may fail altogether. Clearly, the situation is exacerbated in the more dynamic scenarios that are increasingly characterizing modern distributed computing, where publish-subscribe would find its natural use. As an example, mobile computing implies a continuously changing network topology, where reliable links are often difficult to maintain and where the event dispatching infrastructure is itself continuously reconfigured, providing an additional source of event loss.

In this paper, we present solutions for reliable publish-subscribe. The starting point of our research was the desire to combine our previous work on efficient reconfiguration of content-based routing in the presence of changes in the underlying network topology [11, 20] with a mechanism to minimize the loss of events caused by such reconfiguration. Nevertheless, the contribution we put forth here goes well beyond this original goal, in that it is not tied to a specific source of event loss and hence it enjoys general applicability. As such, it is useful also in more traditional scenarios, e.g., those characterized by a fixed topology and unreliable links.

The approach we investigate in this paper relies on *epidemic algorithms* [2, 17, 14], a breed of distributed algorithms that find inspiration in the theory of epidemics. These algorithms aim at providing a lightweight, scalable, and robust means of reliably disseminating information to a group of recipients, by providing guarantees only in probabilistic terms. Given their characteristics, epidemic algorithms are amenable to the unreliable and highly dynamic scenarios we target. At the same time, epidemic algorithms were never applied to content-based publish-subscribe, and previous results in other fields, e.g., multicast communication, cannot be easily adapted to such scenario.

The paper is structured as follows. Section 2 provides the reader with the necessary background information concerning content-based publish-subscribe systems and epidemic algorithms. Section 3 analyzes the challenges posed by the application of epidemic algorithms in the specific context of content-based routing. Section 4 presents three algorithms we designed to provide reliability in content-based publish-subscribe systems. Section 5 provides a discussion of the contribution put forth by this paper. Finally, Section 6 places our contribution in the context of related work, and Section 7 ends the paper with brief concluding remarks.
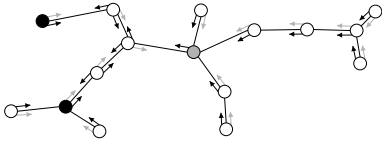
Figure 1: Subscription forwarding.

2. BACKGROUND

In this section we provide the reader with the background information about content-based publish-subscribe systems and epidemic algorithms necessary to grasp the contribution put forth by this paper.

2.1 Content-Based Publish-Subscribe

Several publish-subscribe middleware are available, 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 contain expressions (called *event patterns*) that provide 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. These 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 [4], subscriptions are delivered to every dispatcher along a single unrooted tree connecting all the dispatchers, and are used to establish the routes that are 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. 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 typically optimized by avoiding propagation of subscriptions for 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 a given 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. As a consequence of the subscription forwarding process we described, the routes for the two separate subscriptions are laid down on the single tree constituting the dispatching network. 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.

2.2 Epidemic Algorithms

Epidemic (or *gossip*) algorithms (e.g., [2, 17, 14]) recently became popular as a solution to address scalable and reliable multicast dissemination of information. These algorithms are inspired by the theory of epidemics, in that communication is achieved by trying to "infect" as many nodes as possible. In essence, gossip algorithms trade the strong reliability guarantees, typical of the traditional deterministic approaches, for better scalability, achieved at the price of weaker guarantees defined only in probabilistic terms.

Although epidemic algorithms have been originally developed to deal efficiently with the consistency management of replicated databases [12], they have been applied to a number of problems, including dissemination of news through NNTP and multicast in ad hoc mobile networks [6, 18].

Basic Concepts. The idea underlying this family of algorithms is for each process to communicate periodically its knowledge about the system "state" to a random subset of other processes. Hereafter, the state we consider is the set of messages appeared in the system so far. Missing messages are recovered through one or more "gossip rounds", during which other processes potentially holding a copy of the data are contacted. A gossip round consists of the following steps:

1. Process A chooses randomly another process to communicate with, say B.

2. A sends to B information that allows to determine the presence of inconsistencies in their view of the system's state (e.g., the identifiers of the messages A has received, or missed).

3. A and B reconcile their state by exchanging the actual messages that are not part of the history of both.

Epidemic algorithms differ along two dimensions. The first one is the mode of communication, which 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 a process solicits the transmission of information from other processes to compensate for local losses. Demers et al. showed in [12] that a pull approach converges faster than push, provided that a majority of the participants have the requested message. This can be explained intuitively by considering a scenario where a broadcast message reaches all the receivers but one. In this case, the pull strategy allows the receiver who missed the message to immediately recover it instead of waiting to be pushed, thus improving message delivery latency.

Another dimension along which gossip algorithms differ is the scheme used for disseminating the information about the process state, which can exploit either *positive* or *negative* gossip messages. In the former scheme, each gossip message sent by a process contains the state of communication as perceived by the process, e.g., the content of the process' event history. Hence, the gossip message lists all the events that the process has *received* lately. In a negative gossip scheme, instead, the gossip message contains the events that the process has *missed*.

While in principle the style of communication, push vs. pull, and the information dissemination scheme, positive vs. negative, are orthogonal, in practice pull/negative and push/positive are the most meaningful cases and those typically exploited. In the presence of a pull strategy, a negative scheme can be naturally used to react to a missing message by "pulling" it from other processes, while the positive scheme is best employed to proactively push a process' state to the rest of the system.

This last remark highlights a key difference between the two solutions we are considering, that impacts their performance and applicability. The pull/negative strategy is intrinsically reactive, in that it is triggered only when a process realizes it has lost a message; otherwise, no action is necessary. Instead, push/positive must be implemented according to a proactive scheme, where gossip takes place periodically. Typically, the high degree of reactivity provided by the first approach is preferable. Nevertheless, in some scenarios, e.g., those characterized by a low rate of communication, a process may not realize it missed a message until the next one is received. In this case, the other approach may be preferable. Clearly, the two are not mutually exclusive, and mixed approaches are also possible [12, 14].

Advantages and Motivation. The probabilistic and decentralized nature of these algorithms gives them a number of desirable properties. Gossip algorithms impose a constant, equally distributed load on the processes in the system, and are very resilient to changes in the system configuration (e.g., topological changes) since they do not rely on the existence of one or more processes. Moreover, these properties are preserved as the size of the system increases, thus leading to good scalability. Finally, these algorithms are very simple to implement and rather inexpensive to run.

Gossip algorithms are then a good match for highly distributed and dynamic scenarios. Nevertheless, their application to the case of content-based publish-subscribe system is not straightforward. Content-based systems pose peculiar challenges that have not been tackled thus far by the research community, which at best has concentrated on the simpler subject-based publish-subscribe systems. Still, the synergy between the two approaches is worth investigating, since a content-based approach enhances the underlying publish-subscribe middleware with unprecedented levels of flexibility, hence simplifying the programmer's task.

The challenges arising from the use of epidemic algorithms for content-based publish-subscribe are examined next.

3. CHALLENGES

Epidemic algorithms typically rely on some notion of *group* (or subject) that is exploited in at least two ways. On one hand, the group defines the set of nodes (the group members) that collectively define the scope of a gossip interaction. Hence, it provides a way to determine how to route gossip messages within the system. On the other hand, the group is used to tag the messages exchanged within it. Hence, it provides a clue for the recovery process when the message gets lost. These criteria have been applied successfully in systems that already provide a notion of group for the purpose of enabling communication, e.g., multicast protocols, group communication facilities, and subject-based publish-subscribe.

Unfortunately, content-based publish-subscribe systems do not provide an *explicit* group notion. One could argue that a subscription to an event pattern can be treated as an implicit group, but the analogy actually holds only to a given extent. In subject-based systems, each message is associated to one subject (the group), for which routing is performed independently. Instead, in content-based systems routing is entirely based on the message content: hence, a single message can match different patterns—i.e., different groups in the aforementioned analogy.

This observation is at the core of the challenge of applying epidemic algorithms to content-based publish-subscribe systems, which can be summarized by the following issues:

- *Detecting event loss.* In subject-based systems, a simple solution to detect event loss relies on tagging a published event at its source s not only with its subject p, as already done by s for routing purposes, but also with a sequence number associated to s and p. This sequence number gets increased each time an event for the subject p is published by s. Receivers can then easily detect an event loss by discovering missing sequence numbers in the set of events received by a given source and for a given subject. In content-based systems this technique must be somehow generalized, since an event may match many patterns instead of a single subject, and the source is not required to tag the published event in any way, since routing is entirely determined by the event content.

- *Routing gossip messages.* In subject-based systems, the subject defines the set of nodes it is useful to gossip with, since it contains the set of receivers for a given event associated to that subject. In content-based systems, however, this set of nodes cannot be determined as easily since the subject notion is missing. In principle, gossip interactions should rely on content-based routing as much as possible, since it is precisely the event content that determines the set of potential recipients. Nevertheless, it is generally not possible to simply route gossip messages as normal events. This is particularly evident when a pull approach with negative digests is applied. In this case, a node starts a gossip round to retrieve an event that has been lost and whose content is, by definition, not available.

4. RELIABLE CONTENT-BASED PUBLISH-SUBSCRIBE

In this section, we present three epidemic algorithms that overcome the challenges described in the previous section. The first algorithm, presented in Section 4.1, uses proactive gossip push with positive digests. Instead, in Section 4.2 we illustrate two alternatives using reactive pull with negative digests. All the solutions assume that a unicast transport layer (not necessarily reliable, e.g., UDP-based) is available, which is a reasonable assumption in most environments.

The behavior of each event dispatcher is formalized, for what concerns the processing of gossip messages, by using a very simple pseudo-code notation, as shown in Figure 2 and 3. The key steps of the algorithms are formalized as actions, reminiscent of subroutines, whose body is assumed to be executed atomically. The three algorithms we present share a common structure. Each dispatcher periodically initiates a new round of gossip by performing the operations described by an action startGossipRound, that is invoked externally upon expiration of a timeout determined by the *gossip interval*. We omitted the actions concerning the setting and triggering of this timeout as their semantics is trivial. Processing of gossip messages received by neighbor dispatchers is instead codified in the action handleGossipMsg. The variable *Cache* contains the event messages stored by the dispatcher. In the description and formalization of our algorithms we do not delve in the details of the policy employed to select which events to cache. A reasonable policy is to cache only the events for which a dispatcher is either a subscriber or a source. Nevertheless, other alternatives are meaningful depending on the deployment scenario, and we provide additional insights about this issue later in this paper. Similarly, we do not detail further the last step of a gossip interaction, i.e., the sending of the missing event. A reasonable implementation is to exploit an out of band channel, e.g., a unicast link.

4.1 Push

To provide an answer to the questions identified in Section 3 in the case of proactive push with positive digests we observe that, with this strategy, a gossip message sent by a dispatcher should include information about the set of events it cached. Moreover, this gossip message should be sent only to dispatchers subscribed to such events. As we already discussed, in content-based publish-subscribe systems this set of subscribers cannot be computed once and for all. Nevertheless, we can leverage off of the fact that every dispatcher that received and cached an event e knows, from its subscription table, all the patterns matching e. This means that each dispatcher is able to construct a gossip message which includes a digest of all the cached events matching a given pattern p. This gossip message can then be labelled with p and routed similarly to events matching p.

Figure 2 describes an algorithm based on the considerations above. When startGossipRound is invoked by the dispatcher acting as the gossip initiator—or *gossiper*—a pattern p is chosen according to some strategy (e.g., randomly) from the dispatcher's subscription table and a digest is constructed which includes the (globally unique) identifiers[3]

---

[3] A straightforward implementation of this identifier is the pair given by the source identifier and a monotonically increasing sequence number associated to the source.
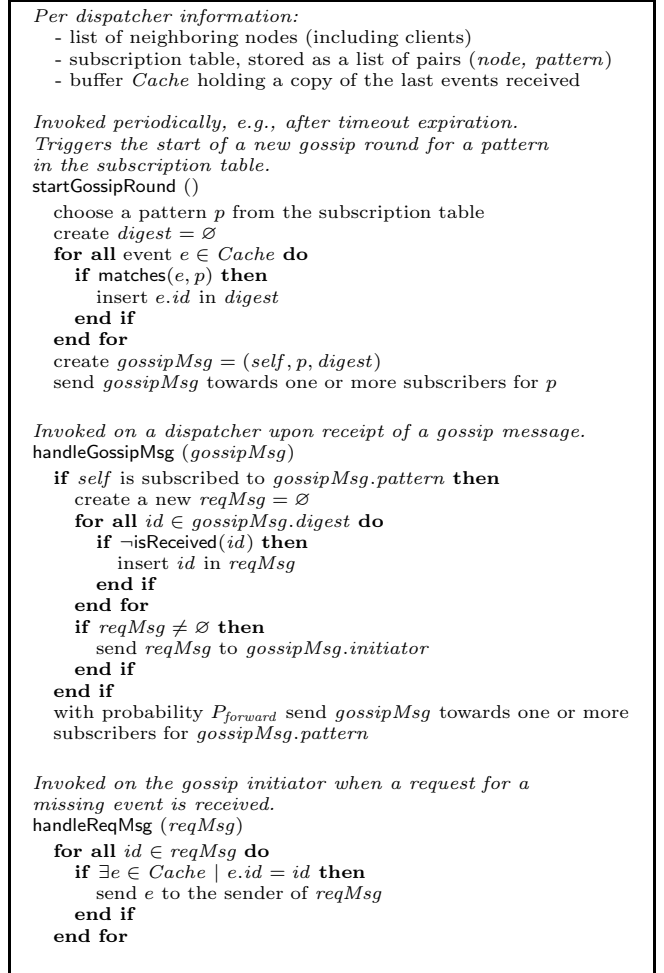
```
Per dispatcher information:
  - list of neighboring nodes (including clients)
  - subscription table, stored as a list of pairs (node, pattern)
  - buffer Cache holding a copy of the last events received

Invoked periodically, e.g., after timeout expiration.
Triggers the start of a new gossip round for a pattern
in the subscription table.
startGossipRound ()
  choose a pattern p from the subscription table
  create digest = ∅
  for all event e ∈ Cache do
    if matches(e, p) then
      insert e.id in digest
    end if
  end for
  create gossipMsg = (self, p, digest)
  send gossipMsg towards one or more subscribers for p

Invoked on a dispatcher upon receipt of a gossip message.
handleGossipMsg (gossipMsg)
  if self is subscribed to gossipMsg.pattern then
    create a new reqMsg = ∅
    for all id ∈ gossipMsg.digest do
      if ¬isReceived(id) then
        insert id in reqMsg
      end if
    end for
    if reqMsg ≠ ∅ then
      send reqMsg to gossipMsg.initiator
    end if
  end if
  with probability Pforward send gossipMsg towards one or more
  subscribers for gossipMsg.pattern

Invoked on the gossip initiator when a request for a
missing event is received.
handleReqMsg (reqMsg)
  for all id ∈ reqMsg do
    if ∃e ∈ Cache | e.id = id then
      send e to the sender of reqMsg
    end if
  end for
```

Figure 2: Push.

of all the cached events matching p. The gossip message *gossipMsg* is then labelled with the pattern p and propagated along the dispatching tree. Routing of *gossipMsg* message outside the gossiper and along the way towards a subscriber is determined as usual, by looking at the subscription table to find neighbors interested in the pattern p. Nevertheless, it is worth noting that the gossip message is not necessarily *duplicated* on all the outgoing routes towards subscribers, as in the normal operation of a publish-subscribe system. Instead, to limit the overhead, it is forwarded only to a subset of the neighbors on the dispatching tree, e.g., determined randomly. The extent of propagation is determined at each hop by the probability $P_{forward}$.

Moreover, in traditional push-based approaches every node gossips only with nodes sharing the same interests. A similar behavior could be obtained in a content-based publish-subscribe system by limiting the choice of p to those patterns in the subscription table that belong to subscriptions issued locally, i.e., by the clients attached to dispatcher. Nevertheless, in the system we consider a dispatcher receives—and stores—also subscriptions that are not issued locally, but that are nonetheless handled by the dispatcher because it is on the route toward a subscriber. Consequently, the above strategy would potentially limit the scope of each gossip

round. For this reason, in our solution p is selected by considering the whole subscription table, i.e., among all the patterns known to the dispatcher. This increases the chances of eventually finding all the dispatchers interested in the cached events, and speeds up convergence.

Upon receipt of *gossipMsg*, a dispatcher is expected to perform the operations represented by the action handle-GossipMsg. These consist of checking if the dispatcher is subscribed to the pattern p labelling *gossipMsg* and, if yes, of verifying if all the identifiers in the digest correspond to events already received. In our solution, the details of how this test is performed are glossed over, and encapsulated in a function isReceived (id), which returns *true* if the dispatcher received an event with the given id.

The identifiers of all the missed events, if any, are then included in a request message *reqMsg* which is sent back to the gossiper using an out of band channel. Upon receipt of this message, the gossiper invokes the action handleReqMsg, which selects the events with the corresponding identifier from the cache, and sends them back to the requester. This third and last phase concludes the interaction taking place in our push approach.

4.2 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, the problem of detecting an event loss in a content-based system is complicated by the fact that dispatchers do not receive all events, but only those matching the patterns they are subscribed to.

The technique we employ to overcome this problem is to tag events with identifiers carrying enough information to detect loss. Besides the event source, these identifiers contain information about the patterns[4] matched by the event, each associated with a sequence number incremented at the source each time an event is published for that pattern. For instance, let us consider a publisher with identifier S, which already published four events matching a pattern P_a, and other three matching P_b[5]. When this event source publishes a new event that matches both patterns, the identifier associated to the event message is $S{:}P_a{-}5{:}P_b{-}4$. Patterns are associated to an 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.

This scheme, which is a generalization of the one we described in Section 3 for subject-based systems, enables the detection of event loss. 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 the remainder of this section, we present two solutions that both rely on the this technique for detecting event loss, but differ in the way they attempt to retrieve the missing event. The solutions are complementary, in that the first one steers gossip messages towards the event receivers (the subscribers),

while the other steers them towards the event sender (the publisher). Both solutions are shown in Figure 3.

Subscriber-Based Pull. In the solution shown in Figure 3, as soon as a lost event is detected it is immediately inserted in the buffer *LostBuffer*, by an action that is not shown explicitly in the figure to keep the algorithm description concise. The elements of *LostBuffer* are the triples identifying an event in our encoding, i.e., source, pattern, and sequence number associated to the pattern and source. The action startGossipRound, that is invoked at regular intervals like in the push solution, first checks whether there are lost events. If yes, a gossip round is effectively triggered[6]. Note how in this case we do not use the whole subscription table like in the push approach, since here the focus is on retrieving events that are relevant to the gossiper, and not on disseminating events to as many dispatchers as possible. Consequently, the pattern p is drawn from the ones associated to subscriptions issued locally, i.e., by the clients attached to dispatcher. Pattern p is used to select the corresponding lost events from *LostBuffer*, which are inserted them in the digest attached to the gossip message *gossipMsg*, which is then labelled with p and routed in a way analogous to the push solution.

When a dispatcher receives a *gossipMsg*, it checks its event cache to see whether it holds some of the events requested by the gossiper. It does not matter whether the dispatcher at hand is a subscriber for the pattern p requested by the gossiper. For instance, following up on our earlier example, let us suppose that the gossiper is missing the event $S{:}P_a{-}5$, and that this information is included in a *gossipMsg*. Of course, there is no way for the gossiper to know that this event has been delivered also to dispatchers subscribed to P_b. Instead, a dispatcher that is subscribed to events matching P_b and has cached the event can easily determine that $S{:}P_b{-}4$ and $S{:}P_a{-}5$ are indeed the same event by looking at the event identifier ($S{:}P_a{-}5{:}P_b{-}4$). Hence, the dispatcher can retransmit the missed event to the gossiper.

Although events can be retransmitted by any dispatcher that has "seen" the event, it is very important for a gossip message to be steered towards a subscriber for the same pattern of the lost event (P_a in our case). In fact, subscribers act as "points of accumulation" for events, in that not only they might have the requested event in the cache, but they also actively try to recover lost events through gossip. Hence, when a *gossipMsg* reaches a subscriber rather than a middleman dispatcher, the likelihood of recovering the lost events is much higher.

Publisher-Based Pull. The right side of Figure 3 shows a source-routing scheme that recovers lost events by walking backwards towards the publisher. While in the other algorithms we are not sensitive about the policy used to cache events, here we assume that published events are cached at the source and possibly at all the dispatchers located on routes towards the subscribers for that event. Moreover, the address of each dispatcher encountered by the event during its travel towards a subscribers is appended to the event

---

[4]A hash signature of the pattern is actually enough.

[5]Content-based systems allow rather sophisticated expressions. For instance, P_a could be {`Software* OR Appl*s`} and P_b {`Distributed AND ?pplication?`}. Events containing `"Distributed Applications"` would then match both patterns.

[6]In principle, a gossip round could be triggered immediately upon detection of a lost event. Nevertheless, in scenarios characterized by frequent losses it is convenient to delay the triggering to the next gossip round, so that multiple lost events can potentially be retrieved during a single round.

```
Per dispatcher information:
   - list of neighboring nodes (including clients)
   - subscription table, stored as a list of pairs (node, pattern)
   - buffer LostBuffer holding a triple
     (source, pattern, sequence number) for each lost event
   - buffer Cache holding a copy of the last events received


Invoked periodically, e.g., after timeout expiration.
Triggers the start of a new gossip round for a pattern in the sub-
scription table.
startGossipRound ()
   if LostBuffer ≠ ∅ then
      choose a pattern p from the subscription table (considering only
      those coming from clients)
      create digest = ∅
      for all (s, p, c) ∈ LostBuffer do
         insert (s, c) in digest
      end for
      create gossipMsg = (self, p, digest)
      send gossipMsg towards one or more subscribers for p
   end if


Invoked on a dispatcher upon receipt of a gossip message.
handleGossipMsg (gossipMsg)
   for all (s, c) ∈ gossipMsg.digest do
      if ∃e ∈ Cache | e matches (s, gossipMsg.pattern, c) then
         send e to gossipMsg.initiator
      end if
   end for
   if self is not subscribed to gossipMsg.pattern then
      send gossipMsg to one or more subscribers for gossipMsg.pattern
   else
      with probability P_forward send gossipMsg to one or more sub-
      scribers for gossipMsg.pattern
   end if
```

```
Per dispatcher information:
   - list of neighboring nodes (including clients)
   - subscription table, stored as a list of pairs (node, pattern)
   - buffer LostBuffer holding a triple
     (source, pattern, sequence number) for each lost event
   - buffer Cache holding a copy of the last events received
   - buffer Routes holding a pair (source, route) for each event source

Invoked periodically, e.g., after timeout expiration.
Triggers the start of a new gossip round for a source.
startGossipRound ()
   if LostBuffer ≠ ∅ then
      choose a publisher s
      create digest = ∅
      for all (s, p, c) ∈ LostBuffer do
         insert (p, c) in digest
      end for
      create gossipMsg = (self, s, digest, r) with r|(s, r) ∈ Routes
      send gossipMsg to the first node in gossipMsg.route
   end if


Invoked on a dispatcher upon receipt of a gossip message.
handleGossipMsg (gossipMsg)
   for all (p, c) ∈ gossipMsg.digest do
      if ∃e ∈ Cache | e matches (gossipMsg.source, p, c) then
         send e to gossipMsg.initiator
      end if
   end for
   if self = gossipMsg.source then
      drop gossipMsg
   else
      send gossipMsg to the next node in gossipMsg.route
   end if
```

Figure 3: Subscriber-based (left) and publisher-based (right) pull.

message, thus recording a route from the publisher to the subscriber. Lost events are stored in *LostBuffer* as described earlier for the subscriber-based solution. In addition, a new buffer *Routes* is necessary to store the route towards a given publisher, e.g., based on the route information stored in the most recent event received from that source.

When a new gossip round is triggered, an event source is chosen among those known. The actions startGossipRound and handleGossipMsg essentially behave like their counterparts in the subscriber-based pull algorithm, except for the fact that the information distinctive of the gossip message is now the event source rather than pattern, and that *gossipMsg* is now augmented with the information necessary to be routed back to the publisher. It is interesting to note 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 source.

One reasonable question to ask is whether this solution suffers from the well-known "acknowledgment implosion" problem, which affects several reliable group communication schemes and occurs when several nodes missing a message request retransmission simultaneously to the same node. Our push and subscriber-based pull solutions, thanks to their distributed nature, are essentially free from this risk. For the publisher-based solution, the probability of such a phenomenon is rather low. In fact, it is unlikely that two subscribers holding different subscriptions (e.g., P_a and P_b) realize at the same time that they have missed an event.

Following our example, this would happen only if the next event published by S matches both patterns as well. Finally, differently from traditional NACK-based reliable multicast schemes, retransmission requests are handled by the first dispatcher holding the desired event found along the path, thus avoiding to overload the publisher.

5. DISCUSSION

As mentioned in Section 1, 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 improve reliability in any situation where an event loss may occur.

The three solutions we proposed exhibit very different characteristics, and are hence likely to perform best in different scenarios. To begin with, we observe that the effectiveness of pull-based solutions strongly depends on the availability of a reasonable number of subscribers for the same pattern used to "pull" messages. Hence, scenarios where the sets of subscriptions held by each dispatcher have a small intersection are very critical for pull, while they do not affect significantly the performance of push. One way to overcome this limitation of the pull approaches is to couple them with others. A natural solution is to combine subscriber-based with publisher-based pull, since the two mechanisms are dual and complementary. The rationale is that this combination improves the chances to recover an event, since the

recovery process proceeds not only along the routes towards receivers, but also towards the sender.

The number of subscribers does not affect significantly the push approach, whose performance is instead challenged by scenarios with a high number of patterns. The problem is that our push approach recovers messages by periodically gossiping the digest of received events for a given pattern. The more the patterns, the bigger the latency experienced in recovering an event, which meanwhile may get discarded from the buffer of dispatchers and eventually get lost forever. We observe, however, that what dominates the behavior of push is not the sheer number of patterns in the system, but only the number of patterns potentially matched by each event. In a scenario where a content-based system is being exploited for multiple applications at once (or by users with different roles) this number should not be very high, as it is unlikely that a single event matches all subscriptions, as the latter express very different application concerns.

The tradeoff between delivery and overhead can be tuned by intervening on a pair of parameters that are tightly related: the buffer size and the gossip interval. For instance, a bigger buffer size essentially allows events to persist longer in the cache, and hence enables less frequent gossiping, thus essentially trading storage consumption on dispatchers for communication overhead. The situation can be improved by exploiting more refined policies for discarding events from the cache. For instance, a dispatcher may discard first events for which it is neither a source nor a subscriber. Or, a probabilistic scheme can be employed to ensure that a given event does not get discarded at about the same time from all caches in the system.

One great advantage of pull-based solutions is their re-activity: a gossip round is triggered only when an event is lost. Reactive approaches perform better, in terms of generated traffic overhead, when there is a large variability in the frequency of event losses, e.g., in situations where bursts of errors are followed by periods in which the system looses very few events. A significant instance of this case is when the changes occur in the topology of the dispatching infrastructure, e.g., because of mobility. Instead, a proactive push approach is likely to result in wasted bandwidth when the system is stable. To remove this potential source of inefficiency, an adaptive approach can be exploited where the gossip interval is changed dynamically according to the current state of the system, as suggested for instance in [9].

Another issue is computational overhead. Pull-based solutions require that a dispatcher, when publishing an event e, performs a match of e against *all* the patterns in its subscription table. This introduces overhead 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 type overhead, we also observe that only the publisher experiences it: the event routing performed by the other dispatchers in the system follows the normal processing.

We conclude this section by pointing out that the considerations we just expressed are actually supported by a thorough evaluation we are carrying out through simulation. The results indeed demonstrate that our algorithms bring a remarkable improvement of event delivery, are scalable, and pay only a limited overhead. Nevertheless, space limitations forced us to present here only a concise discussion of our findings. The reader interested in the complete evaluation, comprehensive of simulation charts and a description of the simulated scenario, is redirected to [8].

6. RELATED WORK

Several centralized publish-subscribe systems offer a reliable service (e.g., all the JMS [28] compliant ones). Moreover, some of the existing distributed subject-based publish-subscribe systems provide a reliable service [30, 23, 3, 21, 31, 5]. Similarly, researchers working on reliable multicast [24, 19, 16] and group communication [13, 7] proposed several protocols for reliable multicast where routing is group or subject-based. Unfortunately, none can be used for the systems we target here, as discussed in Section 3.

Only a 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 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 the reconfiguration of several trees and generate high overhead.

The Hermes [22] system provides a form of content-based routing based on constraints on type attributes. Hermes exploits Pastry [26] as the basic transport layer, and hence inherits the ability to rearrange the overlay network and the routing information as a consequence of 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* [15]. 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 results in 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 are subject to a high traffic, and hence must keep their event caches very large to increase the probability of correctly delivering events.

7. CONCLUSIONS

Modern distributed computing is steering towards scenarios that are increasingly large scale, unreliable, and highly dynamic. Distributed content-based publish-subscribe embodies a communication model providing the necessary component decoupling, flexibility, expressiveness, and scalability to deal with these characteristics at the application level.

Nevertheless, the problem of reliable event delivery, which is exacerbated by the aforementioned scenarios, has not yet been tackled by researchers, and is hampering the exploitation of content-based publish-subscribe middleware in real-world applications.

In this paper, we presented three solutions that introduce reliability in content-based publish-subscribe systems by means of epidemic algorithms.

Our results are derived without making assumptions about the source of event loss, and hence enjoy general applicability. Our ongoing work, however, is aimed at complementing the results we described with those we already obtained for the reconfiguration of the dispatching infrastructure, and convey them in a new-generation distributed content-based publish-subscribe system able to tolerate arbitrary reconfigurations and minimizing the number of events lost.

8. 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 Transactions 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 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 87–94, Elmau, Germany, 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] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8), 2002.

[6] R. Chandra, V. Ramasubramanian, and K. Birman. Anonymous gossip: Improving multicast reliability in mobile ad-hoc networks. In *Proc. 21st Int. Conf. on Distributed Computing Systems*, pages 275–283, 2001.

[7] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[8] P. Costa, M. Migliavacca, G.P. Picco, and G. Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe. Technical report, Politecnico di Milano, March 2003. Available at www.elet.polimi.it/~picco.

[9] 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, November 2001.

[10] 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, September 2001.

[11] G. Cugola, G.P. Picco, and A.L. Murphy. Towards distributed publish-subscribe middleware for mobile systems. In *Proc. of the 3rd Int. Workshop on Software Engineering and Middleware*, LNCS 2596. Springer, May 2002.

[12] 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.

[13] C. Diot, W. Dabbous, and J. Crowcroft. Group communication. *IEEE Journal on Selected Areas in Communication*, May 1997.

[14] P. Eugster, , S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, pages 443–452, 2001.

[15] P. Eugster and R. Guerraoui. Hierarchical probabilistic multicast. Technical report, EPFL, Lausanne (Switzerland), 2001. Available at icwww.epfl.ch/publications.

[16] B. Levine and J. Garcia-Luna-Aceves. A comparison of known classes of reliable multicast protocols. In *Proc. of the IEEE Int. Conf. on Network Protocols*, October 1996.

[17] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. In *Proc. of the European Dependable Computing Conference*, pages 364–379, 1999.

[18] J. Luo, P. Eugster, and J.-P. Hubaux. Route Driven Gossip: Probabilistic Reliable Multicast in Ad Hoc Networking. Technical report, EPFL, Lausanne (Switzerland), 2002. Available at icwww.epfl.ch/publications.

[19] K. Obraczka. Multicast transport protocols: a survey and taxonomy. *IEEE Communications*, 36(1):94–102, January 1998.

[20] G.P. Picco, G. Cugola, and A.L. Murphy. Efficient Content-Based Event Dispatching in the Presence of Topological Reconfiguration. In *Proc. of the 23rd Int. Conf. on Distributed Computing Systems*, pages 234–243, 2003.

[21] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *Proc. of the Workshop on Distributed Event-Based Systems (DEBS), 2002.*, Vienna, Austria, July 2002.

[22] 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)*, pages 611–618, Vienna, Austria, July 2002.

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

[24] V. Roca, L. Costa, R. Vida, A. Dracinschi, and S. Fdida. A survey of multicast technologies. Technical report, Laboratoire d'Informatique de Paris 6 (LIP6), September 2000. www-rp.lip6.fr.

[25] D.S. Rosenblum and A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proc. of the 6th European Software Engineering Conf. held jointly with the 5th Symp. on the Foundations of Software Engineering (ESEC/FSE)*, LNCS 1301. Springer, September 1997.

[26] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[27] 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.

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

[29] 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.

[30] TIBCO Inc. *TIBCO Rendezvous.* www.rv.tibco.com.

[31] 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 11th Int. Workshop on Network and Operating System Support for Digital Audio and Video*, June 2001.