Publish/subscribe for Dynamic Systems

R. Baldoni and L. Querzoni and S. Tucci Piergiovanni and A. Virgillito and F. Morabito and F. Davide and G. Cortese
2005
Publish/subscribe for Dynamic Systems

R. Baldoni, L. Querzoni, S. Tucci Piergiovanni and A. Virgillito

Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”

Federico Morabito, Fabrizio Davide, Giovanni Cortese

Research and Development
Telecom Italia Learning Services
1 Introduction

The problem of selective information diffusion to a large number of dynamic targets is still an open issue and an hot topic for research. The publish/subscribe communication paradigm seems a promising approach in this sense, but currently we still lack convincing solutions able to give impulse to the development of working systems with high performance.

In a publish/subscribe communication system there are processes, called publishers, that produce information in the form of events, and processes, called subscribers, that consume that information. Each subscriber declares its interest in a subset of all the events produced by issuing subscriptions, i.e. filters on the events. A distributed system, made up of processes called event brokers, enforces the conditions expressed by subscriptions and works to deliver to each subscriber only the events that correctly match its subscriptions.

Publishers and subscribers are not required to communicate directly among themselves but they are rather decoupled: the interaction takes place through the event brokers. This decoupling is a desirable characteristic for a communication system because applications can be made more independent from the communication issues, avoiding to deal with aspects such as synchronization or direct addressing. In this way the communication model becomes data-driven, i.e. it is the content of the exchanged data that dictates the nodes subject of this interaction.

These properties makes the publish/subscribe communication paradigm the natural choice to realize selective data dissemination in the form of many-to-many communication in large settings. In the last few years systems based on this paradigm where implemented for static settings where clients of the service (i.e. publishers and subscribers), that act autonomously, are completely separated by the event broker infrastructure whose components are instead controlled and managed by administrators.

The advent of peer-to-peer (p2p) systems introduced a new model of distributed computation where:

- the scale of the system is very large, comprising up to million of users (peers);
- each peer acts independently from all the others, actually precluding any form of centralized network-wide administration or management;
- each peer acts as a client of the service and cooperates with other peers in the system to enable service for other participants;
- given the size of the system and the autonomy of each peer, the system is intrinsically dynamic as peers can leave at any time, due to a failure or an explicit action, while others can join in.

Such model poses new problems that actually inhibit a straightforward implementation of existing systems and architectures. More specifically, the intrinsic dynamic behavior of p2p systems is a great obstacle to the building and maintenance of any logical structure, as nodes continuously entering and leaving the system disrupt this structure thus increasing the effort needed to maintain it. This disruptive effect can be so strong that, in extremely dynamic settings,
even maintaining simple structures (like a connected graph) can become a non
trivial problem.

For this reason, before even thinking about how information can be diffused
in such a setting, the first problem to be solved is how a basic communication
infrastructure can be maintained.

From this point of view, some work \cite{2, 3, 4, 5, 6} in the last years tried to
address this problem devising new algorithms and protocols whose aim is to
build and maintain basic structures, mainly unstructured overlay networks in
the form of random graphs, that, besides their simplicity, offer some properties
(like small diameter, strong connectivity and load balance) that can be exploited
to implement complex applications. These protocols are built taking explicitly
into account the dynamic behavior of peers and are thus well suited for the cited
settings.

The implementation of a publish/subscribe communication system on top of
such unstructured overlay networks is currently an open issue. A few ideas have
been proposed \cite{7} recently, but are only a step ahead of a simple implementation
based on message broadcast.

Aim of this report is to show the research effort we are conducing on this
point. More specifically we present a novel idea for the implementation of pub-
lish/subscribe on an unstructured overlay network. The proposed system will
exploit the underlying overlay network maintenance protocol to enhance its per-
formance.

The report is structured as follows: in Section \ref{sec:2} we will introduce the pub-
lish/subscribe communication paradigm, and analyze the details of the various
architectural choices involved in the implementation of such systems; in Section
\ref{sec:3} we will introduce unstructured overlay networks, and explain how they can
be maintained in highly dynamic environments; Section \ref{sec:4} is devoted to the in-
troduction of a new idea for the implementation of a publish/subscribe system
in a dynamic p2p environment; finally Section \ref{sec:5} concludes this report.
2 Event Based Data Distribution

The publish/subscribe communication paradigm \[ \text{[1]} \] is a key technology to solve the problem of large scale information diffusion to dynamic targets, thanks to its capability of complete decoupling among participants. In this section we will analyze which architectures can be exploited to implement such paradigm. A more detailed version of this section comprising also mobile systems, and a complete survey of existing implementations, is available in \[ \text{[8]} \].

2.1 The Publish/Subscribe Communication Paradigm

A generic pub/sub communication system (often referred to in the literature as Event Service or Notification Service) is composed of a set of nodes distributed over a communication network. Clients to the systems are divided according to their role into publishers, which act as producers of information, and subscribers, which act as consumers of information. Clients are not required to communicate directly among themselves but they are rather decoupled: the interaction takes place through the nodes of the pub/sub system. This decoupling is a desirable characteristic for a communication system because applications can be made more independent from the communication issues, avoiding to deal with aspects such as synchronization or direct addressing of subscribers from publishers.\[ \text{[1]} \]

Figure 1: High-level interaction model of a publish/subscribe system with its clients (\( p \) and \( s \) indicate a generic publisher and a generic subscriber respectively).

Operationally, the interaction between client nodes and the pub/sub system takes place through a set of basic operations that can be executed by clients on the pub/sub system and vice versa (Figure 1). A publisher submits a piece of information \( e \) (i.e., an event) to the pub/sub system by executing the publish\((e)\) operation. Commonly, an event is structured as a set of attribute-value pairs. Each attribute has a name, a simple character string, and a type. The type is generally one of the common primitive data types defined in programming languages or query languages (e.g., integer, real, string, etc.). On the subscribers’ side, interest in specific events is expressed through subscriptions. A subscription \( \sigma \), is a filter over a portion of the event content (or the whole of it), expressed through a set of constraints that depend on the subscription language. A subscriber installs and removes a subscription \( \sigma \) from the pub/sub system by executing the subscribe\((\sigma)\) and unsubscribe\((\sigma)\) operations respectively.

---

\[ \text{[1]} \] The interested reader can refer to \[ \text{[1]} \] for a more deep discussion on the publish/subscribe paradigm and subscription models.
We say a notification $e$ matches a subscription $\sigma$ if it satisfies all the declared constraints on the corresponding attributes. The task of verifying whenever a notification $e$ matches a filter $f$ is called matching ($e \sqsubseteq f$).

2.2 Subscription Models

Various ways for specifying the events of interest have led to identifying distinct variants of the pub/sub paradigm. The subscription models that appeared in the literature are characterized by their expressive power: highly expressive models offer to subscribers the possibility to precisely match their interest, i.e., receiving only the events they are interested in. In this section we briefly review the most popular pub/sub subscription models.

2.2.1 Topic-based

Notifications are grouped in topics i.e., a subscriber declares its interest for a particular topic and will receive all events related to that topic. Each topic corresponds to a logical channel ideally connecting each possible publisher to all interested subscribers. For the sake of completeness, the difference between channel and topics is that topics are carried within an event as a special attribute. Thanks to this coarse grain correspondence, either network multicast facilities or diffusion trees, one for each topic, can be used to disseminate events to interested subscribers.

Topic-based model has been the solution adopted in all early pub/sub incarnations. Examples of systems that fall under this category are TIB/RV [9], iBus [10], SCRIBE [11], Bayeux [12] and the CORBA Notification Service [13].

Topics are equivalent to the notion of groups used for instance in the context of group communication [14] (e.g., for replication). This is not surprising, since the first systems to offer a form of publish/subscribe interaction were actually extensions of group communication toolkits [15] [16] and the subscription scheme was thus inherently based on groups [17]. Subsequently, subscribing to a topic can be viewed as becoming member of a group and publishing a notification for a topic translates accordingly to broadcasting that notification among the members of the corresponding group. Thanks to the topic-group equivalence the topic-based solution mechanism can drive to very efficient implementations, exploiting directly on one hand the large amount of research work in the multicast area, and on the other the network level multicast implementations for diffusing notifications.

The main drawback of the topic-based model is the very limited expressiveness it offers to subscribers. A subscriber interested in a subset of events related to a specific topic receives also all the other events that belong to the same topic. To address problems related to low expressiveness of topics, several solutions are exploited in pub/sub implementations. For example, the topic-based model is often extended to provide hierarchical organization of the topic space, instead of a simple flat structure (such as in [18] [9]). A topic $B$ can be then defined as a sub-topic of an existing topic $A$. Events matching $B$ will be received by all clients subscribed to both $A$ and $B$. Implementations also often include convenience operators, such as wildcard characters, for subscribing to more than one topic with a single subscription. For the sake of completeness, we point out that the word subject can be used to refer to hierarchical topics.
instead of being simply a synonymous for topic. Analogously, *channel-based* is sometimes used to refer to a flat topic model where the topic name is not explicitly included in the event.

### 2.2.2 Content-based

Subscribers express their interest by specifying conditions over the content of notifications they want to receive. In other words, a filter in a subscription is a query formed by a set of constraints over the values of attributes of the notification composed through disjunction or conjunction operators. Possible constraints depend on the attribute type and on the subscription language. Most subscription languages comprise equality and comparison operators as well as regular expressions. The complexity of the subscription language obviously influences the complexity of matching operation. Then it is not common to have subscription languages allowing queries more complex than those in conjunctive form (examples are [22, 23]). A complete specification of content-based subscription models can be found in [24]. Examples of systems that fall under the content-based category are Gryphon [25], SIENA [20], JEDI [27], LeSubscribe [28], Ready [29], Hermes [30], Elvin [31].

As an example of the content-based model, let us consider again notifications representing stock quotes. Differently from the topic-based scheme, a subscription can involve all the attributes of the notification, on which a subscriber can express a constraint with type-specific operators:

- `StockName = 'IBM' and change < -3`
- `StockName = 'M*' and change >= 1`

In content-based publish/subscribe, events are not classified according to some predefined criterion (i.e., topic name), but rather according to properties of the events themselves. As a consequence, the correspondence between publishers and subscribers is on an event basis. Then, the higher expressive power of content-based pub/sub comes at the price of the higher resource consumption needed to calculate for each event the set of interested subscribers [32, 33]. It is straightforward to see that a topic-based scheme can be emulated through a content-based one, simply considering filters comprising a single equality constraint.

### 2.2.3 Type-based

The type-based pub/sub variant events are actually objects belonging to a specific type, which can thus encapsulate attributes as well as methods. With respect to simple, unstructured models, Types represent a more robust data model for application developer, enforcing type-safety at the pub/sub system, rather than inside the application [36]. In a type-based subscription the declaration of a desired type is the main discriminating attribute. That is, with respect to the aforementioned models, type-based pub/sub sits itself somehow in the middle, by giving a coarse-grained structure on events (like in topic-based) on which fine-grained constraints can be expressed over attributes (like in content-based) or over methods (as a consequence of the object-oriented approach).
2.2.4 Concept-based

The underlying implicit assumptions within all the above-mentioned subscription models is that participants have to be aware of the structure of produced events, both under a syntactic (i.e., the number, name and type of attributes) and a semantic (i.e., the meaning of each attribute) point of view. Concept-based addressing \[37\] allows to describe event schema at a higher level of abstraction by using ontologies, that provide a knowledge base for an unambiguous interpretation of the event structure, by using metadata and mapping functions.

2.2.5 XML

Some research works \[38, 39, 40\] describe pub/sub systems supporting a semi-structured data model, typically based on XML documents. XML is not merely a matter of representation but differs in the fact that introduces the possibility of hierarchies in the language, thus differentiating from a flat content-based model in terms of an added flexibility. Moreover, it provides natural advantages such as interoperability, independence from implementation and extensibility. As a main drawback, matching algorithms for XML-based language requires heavier processing.

2.3 Architectural Model

In this section we describe the reference architectural model we use in our presentation. The architectural model is depicted in Figure 2, including four logical layer, namely Network Infrastructure, Overlay Infrastructure, Event Routing and Matching. We present in the following the functionality associated to each layer as well as the different possible solutions for its realization (also illustrated in the figure).

2.3.1 Network Protocols

Network protocols anchor a pub/sub system to the underlying network by allowing transmission of data among pub/sub system components. Due to the fact that a pub/sub system could span over heterogeneous networks (e.g., LANs, WANs, etc.), it could employ more than a single network protocol either to cope with different software/hardware condition that could be found in a given part of the network or to maximize performance. For example a pub/sub system deployed over a WAN could use MAC broadcast inside a LAN to reach in one shot all recipients of an events while sending events between two LANs using TCP connections.

Transport level Pub/sub systems are usually built exploiting the functionality of common transport-level protocols. That is, nodes in the overlay infrastructure communicate directly through TCP or UDP sockets or using specific TCP-based middleware protocols (like IIOP or SOAP). This choice allows the greater flexibility and ease of deployment, though in such situations, the deployment over a wide-area network can be limited by the presence of network firewalls or private networks, requiring the intervention of an administrator for configuration.
Network-level Multicast  Directly exploiting local-area or wide-area multicast and broadcast network primitives is an efficient way to realize many-to-many diffusion experiencing low latencies and high throughput, thanks to the small delays introduced by implementing the protocols exclusively involving routers and switches. For example, IP Multicasting can be directly used in wide-area topic-based systems, as each topic corresponds exactly to one multicast group. Using IP multicast for content-based systems is not as straightforward because subscribers cannot be directly mapped to multicast groups. This has inspired some research work targeting at organizing subscribers in clusters, where subscribers in the same cluster contain most of the subscriptions in common \[41, 42, 43, 44\]. The main drawback of IP multicast is in its lack of a widespread deployment \[45, 46\]. Hence, network-level multicasting cannot in general be considered as a feasible solution for applications deployed over a WAN (for example TIB/RV or the CORBA Notification Service uses network multicast only for diffusing notifications inside a local area network).

2.3.2 Overlay Infrastructure

A pub/sub system generally builds upon an application-level overlay network. In the following we discuss the possible pub/sub overlays, characterized by the organization of the nodes, the role of each node (pure server or also acting as client) and the overall functionality on which the event-routing algorithm rely on. The discussion includes the conditions under which each infrastructure is more feasible and the constraints it imposes.

Broker Overlay  The support for distributed applications spanning a wide-area, Internet-size network requires the pub/sub system to be implemented
as a set of independent, communicating servers. In this context, each single server is called a *broker*. Brokers form an application-level overlay and typically communicate through an underlying transport protocol. Clients can access the system through any broker and in general each broker stores only a subset of all the subscriptions in the system. The particular case of systems composed by a single broker (centralized architecture) is often considered in the literature [21], [47].

The broker network is implemented as an application-level overlay: connections are pure abstractions as links are not required to represent permanent, long-lived connections, so that the neighborhood in the network is determined purely by a knowledge relation. The topology is assumed to be managed by an administrator, based on technical or administrative constraints. For this reason, a broker overlay is inherently static: topology changes are considered to be rare, mainly to face events such as addition of new brokers or repairing after a failure.

The broker network is the most common choice in actual pub/sub implementations, being used by system such as TIB/RV [9], Gryphon [25], SIENA [26], JEDI [27] and REDS [18], as well as in several event routing algorithms proposed in the literature [49], [50]. Apart from the routing protocols, that we analyze in Section 2.4, the main aspect to be clarified in this type of infrastructure is the topology formed by the brokers themselves. There are basically two solutions, hierarchical or flat. In a hierarchical topology, brokers are organized in tree structures, where subscribers' access points lie at the bottom and publishers' access points are roots (or vice versa). Many contributions [51], [50] rely on this topology, thanks to the simplifications it can allow since notifications are diffused only in one direction. In a flat topology, a broker can be connected with any other broker, with no restrictions. [19] shows the more effective load-balance obtained with a flat topology with respect to a hierarchical one, due to the fact that brokers belonging to upper levels of the hierarchy experience a higher load than ones at lower levels.

**Peer-to-peer Structured Overlay** A peer-to-peer structured overlay infrastructure is a self-organized application-level network composed by a set of nodes forming a structured graph over a virtual key space where each key of the virtual space is mapped to a node. The structure imposed to the graph permits efficient discovery of data items and this, in turns, allows to realize efficient unicast or multicast communication facility among the nodes. A structured overlay infrastructure ensures that a correspondence always exist between any address and an active node in the system despite churn (the continuous process of arrivals and departures of nodes of the overlay) and node failures. Differently from a broker overlay infrastructure, a structured overlay allows to better handle dynamic aspects of the systems such as faults and node joins. Then, it is more suited in unmanaged environments (for example, large-scale decentralized networks) characterized by high dynamicity, where human administration interventions cannot be considered a feasible solution.

As a consequence of the popularity of structured overlays, many such systems have been developed: we cite among the others Pastry [52], Chord [53].

\[ ^2 \] Though centralized architectures are of practical interest being particularly suitable for small-scale deployments, they are evidently out of the focus of our work and will not be considered in the following.
Tapestry [54] (unicast diffusion) or CAN [55], I3 [56] and Astrolabe [57] (multicast diffusion). Structuring a pub/sub system over an overlay network infrastructure means leveraging the self-organization capabilities of the infrastructure, by building a pub/sub interface over it. The event routing algorithm is realized only exploiting the communication primitives provided by the underlying overlay. Examples of systems using this solution are Bayeux [12] and Scribe [58], for what concerns topic-based systems, and Meghdoot, Hermes [59] and Rebeca [60], for what concerns content-based systems. Finally, we cite Select-Cast [61], a multicast system built on top of Astrolabe providing a SQL-like syntax for expressing subscriptions.

Peer-to-peer Unstructured Overlays The overlay networks strive to organize nodes in one flat or hierarchical small diameter network (like a random graph) despite churn and node failures [2]. Differently from broker overlays, nodes in these overlays are not necessarily supposed to be dedicated server but can include workstations, laptops, mobile devices and so on, acting both as clients and as part of the pub/sub system. Moreover, the topology of the overlay is obviously unmanaged (that is, it does not rely on a human administrator).

Unstructured overlays use flooding, gossiping or random walks on the overlay graph to diffuse and to retrieve information associated with the nodes. This is due to the absence of a structure that facilitates event routing, which is difficult to maintain because of node dynamicity (see Section 2.4.2 for details). On the other hand, unstructured overlays are widely used for file sharing applications for their simplicity in handling joins and leaves of nodes (with respect to their structured counterparts) and for the fact that, in such applications, there is no need for precise searches. So unstructured overlay are probabilistic in nature as there is non-zero possibility that some item present in the network is not found during a search. Not many pub/sub systems have been proposed on the top of unstructured peer-to-peer overlay networks. Among them we cite [7], detailed in Section 2.4.3.

2.3.3 Event Routing

The core mechanism behind a distributed pub/sub system is event routing. Informally, event routing is the process of delivering an event to all the subscribers that issued a matching subscription before the publication. This involves a visit of the nodes in the Notification Service in order to find, for any published event, all the clients whose registered subscription is present in the system at publication time.

The impossibility of defining a global temporal ordering between a subscription and a publication that occurred at two different nodes makes this definition of routing rather ambiguous. A discussion on this point as well as formal specifications of the event routing problem can be found in [62].

The main issue with an event routing algorithm is scalability. That is, an increase of the number of brokers, subscriptions and publications should not cause a serious (e.g., exponential) degradation of performance. This requires on one hand controlling the publication process, in order to possibly involve in propagation of events only those brokers hosting matching subscriptions. On the other, reducing the amount of routing information to be maintained at brokers, in order to support and flexibly allow subscription changes. These two aspects
are evidently conflicting and reaching a balance between them is the main aim of a pub/sub system’s designer.

We have identified and classified the approaches presented in literature for event routing. Routing approaches are oblivious to the particular architectural solution in the sense that a same routing algorithm can be used in different infrastructures, though each approach can be more suitable for a specific architecture. Section 2.4 is entirely devoted to describing and comparing routing algorithms, as well as identifying which type of infrastructure is more suitable for each solution.

2.3.4 Matching

Matching is the process of checking an event against a subscription. Matching is performed by the pub/sub system in order to determine whether dispatching the event to a subscriber or not. As we show in the following section, also event routing algorithms often require a matching phase to support the routing choices. As the context of interest is that of large-scale systems, we expect on one side the overall number of subscriptions in the system to be very high, and on the other a high rate of events to be processed. Then, in general the matching operation has to be performed often and on massive data sizes. While obviously this poses no problems in a topic-based system, where matching reduces to a simple table lookup, it is a fundamental issue for the overall performance of a content-based system. The trivial solution of testing sequentially each subscription against the event to be matched often results in poor performance. Techniques for efficiently performing the matching operation are then one important research issue related in the pub/sub field. They can be grouped in two main categories [63], namely predicate indexing algorithms and testing network algorithms. Predicate indexing algorithms are structured in two phases: the first phase is used to decompose subscriptions into elementary constraints and determine which constraints are satisfied by the notification; in the second phase the results of the first phase are used to determine the filters in which all constraints match the event. Matching algorithms falling into the predicate indexing family are [64,65,21,66]. Testing network algorithms ([67,68,22]) are based on a pre-processing of the set of subscriptions that builds a data structure (a tree in [67] and [68] or a binary decision diagram in [22]) composed by nodes representing the constraints in each filter. The structure is traversed in a second phase of the algorithm, by matching the event against each constraint. An event matches a filter when the data structure is completely traversed by it. This quick overview is not intended to cover all the works proposed in the literature and was introduced here mainly for the sake of completeness, since the focus of our work is on distributed event routing. A formal complexity analysis and comparison of matching algorithms can be found in [69].

2.4 Event Routing

In this section, we investigate the general solutions for event routing to achieve scalable information dissemination. Three categories are identified, flooding algorithms (event flooding and subscription flooding), selective algorithms (rendezvous-based and filter-based) and event gossiping algorithms (basic gossiping and informed gossiping). Roughly, flooding algorithms are based on a complete deter-
ministic dissemination of event or subscriptions to the entire system. Selective algorithms aims at reducing this dissemination thanks to a deterministic routing structure built upon subscriptions, that aids in the routing process. Event gossiping are probabilistic algorithms with no routing structure, suitable for highly dynamic contexts. The general characteristics of the algorithms are summarized in Table 1, reporting for each algorithm the type of routing decisions (probabilistic or deterministic), the nodes that perform the filtering (producers, consumers or intermediate nodes on the path from publishers to subscribers) and the nodes to which events and subscriptions are sent (none, all or a subset).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Routing</th>
<th>Filtering</th>
<th>Nodes storing Subs</th>
<th>Nodes handling Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flooding</td>
<td>Det.</td>
<td>Subscribers</td>
<td>None</td>
<td>All</td>
</tr>
<tr>
<td>Subs Flooding</td>
<td>Det.</td>
<td>Publishers</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>Selective</td>
<td>Det.</td>
<td>Intermediaries</td>
<td>Subset</td>
<td>Subset</td>
</tr>
<tr>
<td>Rendezvous-based</td>
<td>Det.</td>
<td>Intermediaries</td>
<td>Subset</td>
<td>Subset</td>
</tr>
<tr>
<td>Gossiping</td>
<td>Prob.</td>
<td>Subscribers</td>
<td>None</td>
<td>All</td>
</tr>
<tr>
<td>Informed gossiping</td>
<td>Prob.</td>
<td>Intermediaries</td>
<td>Subset</td>
<td>Subset</td>
</tr>
</tbody>
</table>

Table 1: Classification of Event Routing Algorithms

In the remainder of this section we give a detailed description of all routing solutions, stating the relationship between each algorithm and the various overlay infrastructures and identifying their trade-offs in terms of the following dimensions:

- **Message overhead**: the overhead induced on the network by sending both publication and subscription messages. It is normally measured in terms of overlay hops, that is the number of nodes that are traversed by an event along propagation. Ideally, an event routing algorithm should reach all subscribers in a single hop. All the further messages besides these are considered as overhead.

- **Memory overhead**: the amount of information stored at each process. Related to subscription replication, which is the number of copies of each subscription that are present in the system.

- **Subscription language limitations**: the routing mechanism may induce limitations on the supported subscriptions, for example regarding the type of constraints.

Besides these aspects, one has finally to consider that event routing algorithm are subject to two types of dynamic changes: i) the behavior of users dynamically changing their subscriptions and ii) the changes in the composition of the system due to the process of arrival, departures and failure of nodes (that is, churn). While all event routing algorithms (except event flooding) are equally subject to the first type of dynamicity, some are more sensitive than others to the latter type, according to the overlay infrastructure they are deployed over. We also highlight this issue in the presentation of the algorithms and discuss it in detail in Section 2.4.2.

---

3In the following we refer as node to a generic node of the pub/sub system, let this be a broker in a broker overlay or a peer in a structured/unstructured overlay. Clients in broker-based architectures are not considered and their behavior is completely handled by nodes in the pub/sub system.

4A simulation study of event routing algorithms has recently appeared in [70].
2.4.1 Event/Subscription Flooding

The trivial solution for event routing consists in propagating each event from the publisher to all the nodes in the system (event flooding, Figure 3(a)). This algorithm can be simply implemented in all the architectures: a network-based solution consists in broadcasting each event in the whole network, while with any form of overlay it suffices for a node to forwards each event to all the known processes. The obvious drawback is that this routing mechanism does not scale in terms of message overhead. However, event flooding presents minimal memory overhead (no routing information needs to be stored at a node) and there are no language limitations.

On the other side of the spectrum of routing solutions with respect to the routing information stored at each node lies the subscription flooding approach: each subscription is sent to all the nodes together with the identifier of the subscriber. That is, each node has the complete knowledge of the entire system, thus recipients can be reached in a single hop (the ideal value) and non-interesting events can be immediately filtered out at producers (Figure 3(b)). However, both simulations studies ([71], [19]) and practical experiences ([20]) report that subscription flooding can rarely be considered a feasible solution if subscriptions change at a high rate, as each node has to send all the changes to all other nodes (in other words, the overlay is completely connected). For example, the complete flooding of subscriptions was a characterizing feature (referred to as “quenching”) of an older version of Elvin [31], which was removed in a successive version ([20]), since it proved to be very costly. A recent work presenting a subscription flooding approach is MEDYM [72], an algorithm part of the DADI framework.

Figure 3: Example of Event Routing Algorithms. Black boxes and arrows represent published and sent events, gray boxes and arrows represent stored and sent subscriptions and white boxes represent stored routing information.
2.4.2 Selective Event Routing

The principle behind Selective Event Routing algorithms is to reduce the message overhead of event flooding by letting only a subset of the nodes in the system store each subscription and a subset of the nodes be visited by each event (both subsets possibly spanning the whole system). Selective routing algorithms allow to save network resources particularly when an event has to be transmitted only to a restricted portion of subscribers. When most events are of interest for a large number of subscribers, flooding can be considered an option \[73,74\] since it avoids the overhead due to the storage and update of event routing information.

Let us point out that selective-based solutions are deterministic approaches to event routing, in the sense that they build event routing data structure to deterministically route event to its intended destinations. Nodes cooperate for letting these data structures do their best to timely track subscription changes. It is important to remark that deterministic event routing does not imply any deterministic guarantee on event delivery. There is indeed a non-zero delay between a change and the time in which the event routing data structure captures this change. During this delay deterministic approaches to event routing might become inefficient, in the sense that they can lead, on one hand, to event loss due to the fact that an event is routed to part of the overlay where there are no longer interested recipients (e.g., due to recent unsubscription) and, on the other hand, to not routing an event to an interested destination that just did the subscription \[62\]. Therefore, deterministic approaches to event routing are clearly best-effort in terms of delivery of events due to topology rearrangements.

Moreover, the effect of churn makes much more pronounced the discrepancy between the event routing data structures at a given time and the ones that would allow ideal deterministic event routing, amplifying, thus, the inefficiency of the event routing with respect to the delivery of events. Deterministic event routing approaches work therefore better over overlay infrastructures where the churn is mastered by some external entity. For example, in managed environments such as a broker overlay, the churn is very low and strictly under control of humans. In a peer-to-peer structured overlay, the churn effect is handled by the overlay infrastructure layer and then masked to the event routing level.

As the size and the dynamic of the system grow, the effect of churn can be disruptive in terms of delivery of events in deterministic event routing even in structured peer-to-peer networks \[75\]. This is why gossip-based (or epidemic) protocols have emerged as an important probabilistic event routing approach to cope with these large scale and dynamic settings \[76\].

Filtering-based Routing In Filtering-based routing \[19\] events are forwarded only to nodes that lie on an overlay path leading to interested subscribers. Message overhead is reduced by identifying as soon as possible events that are not interesting for any subscriber and arrest their forwarding. This approach has been largely studied and used in the literature \[74,70\].

The construction of diffusion paths requires routing information to be stored and maintained on the nodes. Routing information at a node is associated to each of its neighbors in the overlay and consists in the set of subscriptions that are reachable through that broker. This allows to build reverse paths to subscribers followed by events. In practice, copies of all the subscriptions have
to be diffused toward all possible publishers, and in the general case when all
nodes may act as publishers for any subscription, this means again flooding
all subscriptions. However, differently from the subscription flooding approach,
a node communicates directly only with its neighbors, thus reducing the local
message overhead due to subscription update. Subscription diffusion can also
be limited in this approach by exploiting subscription containment, as done in
SIENA and REBECA.

The pseudo-code of filtering-based routing at a broker is presented in Figure
Figure 4. A broker can handle publish or subscribe messages, respectively sent by a
client or by another broker. Each broker maintains three structures: a neighbors
list, a routing table and a subscription list. The routing table associates a
neighbor with an entry representing a set of subscriptions. The subscription
list associates a node to its subscription. The match function matches an event
against either the subscription list or a routing table entry and returns a list
with all the matching nodes. An example of Filtering-based routing is depicted
in Figure 3(c) where the dashed lines represent connections at overlay level.

Upon receive publish(event e) from node x
matchlist ← match(e, subscriptions);
send notify(e) to matchlist;
fwdlist ← match(e, routing);
send publish(e) to fwdlist − x;

Upon receive subscribe(subscription s) from node x
if x is client then
    add s to subscriptions;
else add (x, s) to routing
send s to neighbors − x;

Figure 4: Pseudo code of Filtering-based routing

The natural architecture for this kind of solution is the brokers’ network, usu-
ally structured in an acyclic topology (tree or graph). Actually, the presence of
cycles requires duplicate detection while diffusing both event and subscriptions
and thus is usually avoided in implemented systems. The addressing scheme
of a structured overlay does not represent a useful feature in this type of solu-
tion, except for the fact that it can keep the consistent association between a
node and its position in the overlay, allowing easy overlay repairing upon fail-
ure. However, the consistency of information in the routing tables has still to
be provided by specific event routing-level algorithms. This type of solution is
considered in Hermes [59] and in [60, 77]. The use of Filtering-based routing
over unstructured overlays suffers mainly from the dynamicity of the network,
that requires frequent updates of the routing information. Moreover, it is not
possible to assume an acyclic topology.

The performance of filtering-based routing is obviously influenced by the
topology of the overlay network. In particular, the diameter of the topology
is related to the length of the overlay paths traveled by events, thus affecting
notifications latency. Obviously, increasing the number of neighbors of a node
lowers the diameter of the network, but also the amount of routing information
kept by nodes (memory overhead) increases. This is the reason why the
efficiency of the matching algorithm also impacts on delivery latency.
Finally, filtering-based routing does not impose any limitation on the subscription language. Indeed, the only point in the algorithm dependent on the language is the match() function, that can be implemented easily for any data type.

**Rendezvous-based Routing** Rendezvous-based event routing is based on two functions, namely SN and EN, used to associate respectively subscriptions and events to nodes in the pub/sub system. In particular, given a subscription σ, SN(σ) returns a set of nodes, named rendezvous nodes of σ, which are responsible for storing σ and forwarding events matching σ to all the subscribers of σ. EN(e) complements SN by returning the rendezvous nodes of e, which are the nodes responsible for matching e against subscriptions registered in the system. Upon issuing a subscription σ, a subscriber sends σ to the nodes in SN(σ), which store σ and the subscribers’ identifier.

Then, rendezvous-based event routing is a two phase process: a publisher sends their events to nodes in EN(e), which match e against the corresponding subscriber. For each subscription matched by e, e is forwarded to the corresponding subscriber. In order for the matching scheme to work and forward e to the consumers, it is necessary that the rendezvous nodes of e collectively store all the subscriptions matched by e, i.e., if e ∈ σ for any subscription σ, then \( SN(e) \cap SN(\sigma) \neq \emptyset \). We refer to this property as the mapping intersection rule [78]. The pseudo-code of rendezvous-based routing is presented in Figure 5 (subscriptions list is defined as in Filtering-based routing), while an example is depicted in Figure 3(d), where we assume SN/EN functions that assign subscription \( x < 1 \) and event \( x = 0 \) to node \( n_1 \).

```
upon receive publish(event e) from node x at node i
rvlist ← EN(e);
if i ∈ rvlist then
    matchlist ← match(e,subscriptions);
    send notify(e) to matchlist;
else
    send(e) to rvlist;
upon receive subscribe(subscription s) from node x at node i
rvlist ← SN(s);
if i ∈ rvlist then
    add s to subscriptions;
else
    send(s) to rvlist;
```

Figure 5: Rendezvous-based routing

Rendezvous-based routing has been introduced in [50], and recently many systems appeared following such a scheme (Scribe [11], Bayeux [12], Hermes [30], Meghdoot [79] and [78]). This approach is motivated by the fact that a controlled subscription distribution allows to better load balance subscription storage and management: all subscriptions matching the same events will be hosted by the same node, avoiding a redundant matching to be performed in several different nodes. Also delivery of events is simplified, consisting in the creation of single-rooted diffusion trees starting from target brokers and spanning all subscribers.
However, it is clear that defining the couple of $EN(e)$ and $SN(\sigma)$ functions so that they satisfy the mapping intersection rule is a non-trivial task. This implies defining a clustering of the subscription space, such that each cluster is assigned to a node that becomes the rendezvous for the subscriptions and events that fall into that cluster.

A rendezvous-based algorithm over a broker-based architecture does not handle well dynamicity: when a new node $n$ joins the system, the whole partitioning criteria has to be rearranged among nodes. Moreover, subscriptions that map to $n$'s partition have to be moved to $n$ from the node that was previously in charge. Similarly, when a node leaves or crashes, the subscriptions that it stores should be relocated to another node. Unstructured networks are even less suitable in this sense because the system is highly dynamic and its size not known. On the contrary, the powerful abstraction realized by structured overlay networks greatly helps in the definition of the mapping functions, thanks to the fact that the fixed-size address space can be used as a target of the functions rather than the set of nodes. This allows the mapping to be independent from the actual system composition and not be influenced by changes in it.

Maybe the biggest drawback of rendezvous-based solutions is the restrictions it may impose to the subscription language. In general, mapping a multi-dimensional, multi-typed content-based subscription to the uni-dimensional or bi-dimensional numerical-only address space of structured overlays is not straightforward. While numerical range constraints can be intuitively handled, constraints over string attributes, like substrings, prefixes or suffixes, that are an important part of a content-based language, can be hardly reduced to numerical ranges, then they may be excluded from the subscription language.

As for performance, memory overhead depends on the mapping function used. In general, the mapping function should map a subscription to the lower number of nodes possible in order to satisfy the mapping intersection rule. It is natural though that “larger” subscriptions (i.e. matching more events) will be mapped to more nodes with respect to “smallest” ones. This allows also to share the load due to matching. Moreover, routing information should be preserved at a node to reach the rendezvous nodes.

### 2.4.3 Gossip-based

In basic gossip-based protocols, each node contacts one or a few nodes in each round (usually chosen at random), and exchanges information with these nodes. The dynamics of information spread resembles the spread of an epidemic [80] and lead to high robustness, reliability and self-stabilization [81]. Being randomized, rather than deterministic, these protocols are simple and do not require to maintain any event routing data structure at each node trying to timely track churn and subscriptions changes. The drawback is a moderate redundancy in message overhead compared to deterministic solution. Gossiping is therefore a probabilistic and fully distributed approach to event routing and the basic algorithm achieves high stability under high network dynamics, and scales gracefully to a huge number of nodes[81]. Specific gossip algorithms for pub/sub systems have been proposed in [83, 82, 18, 7].

---

5Gossiping has been also used to improve delivery guarantee of a filtering-based event routing protocol in [82].
In gossip protocols, the random choice of the nodes to contact can be sometimes driven by local information, acquired by a node during its execution, describing the state either of the network or of the subscription distribution or both. In this case, we are in the presence of an informed gossip protocol. The algorithm presented by Eugster and Guerraoui in [83] (pmcast) is an example of informed gossip specifically targeted to pub/sub system. It follows a principle similar to that of filter-based routing: avoiding to gossip a message to not-interested subscribers. pmcast organizes processes in a hierarchy of groups. Groups are built and organized in hierarchies according to the physical proximity of nodes. Each process maintains in its view the identities and the subscriptions of its neighbors in a group. Special members in a group, namely delegates, maintain an aggregation of the subscriptions within a group and have access to the delegates view of nodes at adjacent levels of the tree. Events are gossiped throughout the tree. The membership information allows to exclude from gossipping the nodes that are not interested in an event.

Finally let us remark that Costa and Picco in [7] proposed a hybrid approach that mixes deterministic and probabilistic event routing. Subscriptions are propagated only in the immediate vicinity of a subscriber. Deterministic event routing leverages of this subscription information, whenever available, by deterministically routing an event along the link a matching subscription was received from. If no subscription information exists at a given node, events are forwarded along a randomly chosen subset of the available links over the peer-to-peer overlay.

\footnote{To help this process of acquiring information at each node some limited horizon advertising mechanism can be employed.}
3 Unstructured Peer-to-peer Systems

In this section we will show a study on unstructured peer-to-peer systems. Aim of this study, based on simulations on an experimental setup, is to better understand if and how these systems can be used as an overlay network in extremely dynamic scenarios. Before analyzing the results, we introduce these systems through a general background and a more detailed analysis of the tested protocols. The complete study, comprising other simulations along with their analysis, is available in [84].

3.1 Background

With the term peer-to-peer we usually refer to a class of systems that employ distributed resources to perform some task such as distributed computing, data sharing, information distribution, etc. in a distributed manner [85, 86]. Decentralization may apply to algorithms, data, or both. These systems are often implemented as overlay networks which provide mechanisms to search and discover data stored by overlay nodes.

P2P systems can actually be classified in three main groups [87]:

Centralized. Systems (like Napster) that maintain a constantly updated directory of available resources at central locations. Peers issue queries to these nodes in order to obtain the list of peers holding the desired resources. Obviously such systems scale poorly when the number of peers grows and also have single points of failure in directory nodes.

Structured. Systems that exploit a highly-constrained overlay network topology and specific resource placement to provide clients with high performance resource-discovery primitives. Such systems can be further grouped in loosely structured, where the network topology is almost random but resource placement is determined by specific algorithms, and highly structured, where both the network topology and resources are organized in a specific manner. Currently great interest is focused in the latter type and specifically in systems implementing distributed hash tables (DHTs) [52, 54, 53, 55].

Unstructured. These are systems where the network topology is mostly random or built using loose rules. Moreover resource placement is not controlled at all. These architectural choices make overlay network maintenance less cumbersome due to the low overhead generated by join and leave operations, but also render resource discovery expensive; in fact unstructured P2P systems often implement algorithms based on flooding or random walk techniques for resource discovery primitives [88, 89].

Unstructured P2P systems [90, 91, 92, 93] organize overlay nodes in a random graph characterized by some desired properties like low-diameter, high connectivity, constant-node degree. Each node in the overlay network maintains a partial view of the whole system containing network addresses of other nodes participating to the same network. The union of all the nodes together with their partial views, constitutes the whole overlay. Due to the large scale of such systems, partial views are usually limited in their size to a constant degree or to a size that varies only logarithmically with the overlay size.
Joining nodes use some randomized algorithm that starts at a bootstrap node that is considered randomly chosen among those already present in the graph. This bootstrap node is used to build an initial partial view through which a new node effectively becomes part of the overlay. Node leaves can be treated with two different approaches:

**Active leave** Each node is required to execute some protocol before it can actually leave the overlay. This protocol is used to ensure the connectivity of the whole overlay network despite node leaves.

**Passive leave** Each node can simply “disappear” from the system at any time without executing any action.

With the latter approach leaving nodes are actually treated by the system as failed nodes, while with the former one different approaches are required to let the overlay safely treat node failures.

### 3.2 Dynamic Overlay Networks

Since their introduction, peer-to-peer (p2p) technologies were thought as possible solution to introduce new forms of communication in wide, heterogeneous and dynamic communities, but the intrinsic dynamic behaviour of such systems is a great obstacle to the building and maintenance of any logical structure on top of it, as nodes continuously entering and leaving the system disrupt this structure thus increasing the effort needed to maintain it. This disruptive effect can be so strong that, in extremely dynamic settings, even maintaining simple structures (like a connected graph) can become a non trivial problem.

From this point of view unstructured p2p systems, like [3, 5, 6] that are based on random graphs, are considered more suited to such environments than structured ones like [53, 55] or [52]. On the other side such unstructured p2p systems offer only weak communication primitives to the developer, mainly based on broadcasting techniques. Such unstructured systems are built trying to approximate as much as possible a random graph, as this can offer good qualities for broadcast-based communications, like low diameter and high connectivity. Some strategy must then be exploited to maintain such randomness in presence of churn and faults.

From a theoretical point of view unstructured p2p systems are built adding edges to a graph containing only vertices. Each edge is added with probability $p$.

Building such a graph in a real system is a hard challenge. Nodes joining the system must build links to other nodes yet in the system, and these links must be chosen randomly among all those that are possible. Realizing this without resorting to some centralized service can lead to non truly-random topologies, where links don’t have all the same probability of existence.

To mitigate this problem current unstructured p2p systems employ a Overlay Maintenance Protocol (OMP) whose aim is to build and maintain a graph trying at the same time to approximate the desired randomness. OMPs can act in various ways: usually part of the protocol is executed when a node join the system; but in a dynamic setting also nodes leaving the system can affect its properties, thus part of the OMP is also executed periodically (for algorithms that treat leaves passively) or when a node leaves the system (for algorithms that treat leaves actively).
3.3 Protocols Description

The common characteristic of all OMPs is that each node maintains a limited number of links to other nodes in the system. We call this set of link the view of the node. The views should be such that the graph resulting by interpreting links in the view as arcs and nodes as vertexes is connected. OMPs differentiate among themselves with respect to the techniques they use for building and maintaining the views. We consider decentralized OMPs in which such protocols does not require a central coordination. In this Section we describe in detail the two protocols that are subject of our study.

3.3.1 Scamp

SCAMP [5] is a gossip-based protocol whose main innovative feature is that the size of the view is adaptive w.r.t. a-priori unknown size of the whole system. More precisely, size views in SCAMP is logarithmic of the whole system size. The protocol consists of mechanisms for nodes to join and leave, and to detect and recover from isolation and to rebalance the size of partial views across overlay nodes. The following is a brief description of these mechanisms.

Data Structures. Each node maintains two lists, a PartialView of nodes it sends gossip messages to, and an InView of nodes that it receives gossip messages from, namely nodes that contain its node-id in their partial views.

Join Algorithm. New nodes join the overlay by sending a join request to an arbitrary member, called a contact. They start with a PartialView consisting of just their contact. When a node receives a new join request, it forwards the new node-id to all members of its own PartialView. It also creates $c$ additional copies of the new join request ($c$ is a design parameter that determines the proportion of failures tolerated) and forwards them to randomly chosen nodes in its PartialView. When a node receives a forwarded join request, provided the subscription is not already present in its PartialView, it integrates the new node in its PartialView with a probability $p = 1/(1 + \text{sizeofPartialView}_n)$. If it decides not to keep the new node, it forwards the join request to a node randomly chosen from its PartialView. If a node $i$ decides to keep the join request of node $j$, it places the id of node $j$ in its PartialView. It also sends a message to node $j$ telling it to keep the node-id of $i$ in its InView.

Leave Algorithm. The leaving node has ordered the id’s in its PartialView as $i(1), i(2), ..., i(l)$ and the id’s in InView as $j(1), j(2), ..., j(l)$. The leaving node will inform nodes $j(1), j(2), ..., j(l-c-1)$ to replace its id with $i(1), i(2), ..., i(l-c-1)$ respectively (wrapping around if $(l-c-1) > l$). It will inform nodes $j(l-c), ..., j(l)$ to remove it from their lists without replacing it by any id.

Recovery from isolation. A node becomes isolated when all nodes containing its identifier in their PartialViews have either failed or left. In order to reconnect such nodes, a heartbeat mechanism is used. Each node periodically sends heartbeat messages to the nodes in its PartialView. A node that has not received any heartbeat message in a long time re-joins through an arbitrary node in its PartialView.

Re-balancing partial views. Each join request has been given a finite lifetime
called its *lease*. When a join request expires, every node holding it in its PartialView removes it from the PartialView. Each node re-joins at the time that its join request expires. Nodes re-join to a node chosen randomly from their PartialView. Re-joins differ from ordinary joins in that the partial view of a re-joining node is not modified.

### 3.3.2 Cyclon

Cyclon [6] follows a proactive approach, where nodes perform a continuous periodical gossiping activity with their neighbors in the overlay and does not react to failures or departure of other nodes. Only joins are obviously managed in a reactive manner. The periodical gossiping phase (named “shuffle”) has the aim of randomly mixing the views between neighbor nodes. This provokes the long term effect of an overlay whose topology approximates a random graph.

**Data Structures.** Each node maintains only a single view of nodes it can gossip with (i.e., it corresponds to SCAMP’s PartialView). The size of the view is fixed and it can be set arbitrarily. Each node in the view is associated to a age, indicating the number of rounds of the algorithm during which the node was present in the view.

**Join Algorithm.** A node $A$ joins by choosing one node at random among the ones already present in the network. $A$ starts then a set of independent random walks from the contacted node. The number of random walks is equal to the view size, while the number of steps per each random walk is a parameter of the algorithm. When each random walk terminates, the last visited node, say $B$, adds $A$ to its view. If $B$’s view is full, it replaces one node, say $C$, which is added to $A$’s view.

**Shuffle Algorithm.** The shuffle algorithm is executed periodically at each node. A round of the shuffle is composed of three phases. In the first phase a node $A$, after increasing the age of all the nodes in its view, chooses its shuffle target, $B$, as the one with higher age among those in its view. Then, $A$ pushes to $B$ the links to $l - 1$ randomly chosen nodes in its view, plus the link to itself. In the second phase, $B$, once received the shuffle message from $A$, replaces $l - 1$ nodes in its view (chosen at random) with the $l$ links received from $A$ and send them back to $A$. In the final phase $A$ replaces the links sent to $B$ with those received from it (if $A$’s view was not full empty slots are filled instead). In overall, the result of the shuffle is an exchange of $l$ links between $A$ and $B$. The link initially present from $A$ to $B$ is also reversed after the shuffle.

In the specifications given in [6], no action was defined when a node receives a shuffle request while it has a shuffle in progress. If concurrency is considered, the nodes sent by $A$ to $B$ can be modified by concurrent shuffle $A$ is involved in as a shuffle partner. In our implementation, we extended the original specification in order to address this situation: in case nodes to be replaced by $A$ are no longer in its cache, it replaces some nodes chosen at random.
3.4 Evaluation

In this Section we present the details of a simulation study we did on the two presented protocols. We point out that this work is not intended to be a comparison between the two protocols, since they were designed for different purposes: SCAMP was originally targeted at the construction of overlays for large-scale information dissemination, for which the reactive nature of the protocol is more appropriate, while Cyclon is in general suited for applications requiring a constant sampling of nodes in the network (for example searching, monitoring and so on). For this reason, SCAMP has been tested under moderate dynamics while Cyclon was subject to heavier churn.

3.4.1 Experimental Setting

The simulation study was carried out by developing the two algorithms in Peersim. The event-driven mode of Peersim was used for both algorithms. Event-driven simulations in Peersim are based on a logical clock. At each time unit of the clock one or more events (join and leave operations of nodes, send and receive operations of messages) can be scheduled. Differently from the cycle-driven mode of Peersim, the event-driven mode allows to introduce concurrency in the following senses: i) nodes are not synchronized in the execution of joins, leave and shuffle operations; ii) a random delay occurs between the send and receive of a message. The consequence of these two assumptions is that actions of a protocol are not atomic, have non-finite duration and can interleave.

Both protocols implementations were validated comparing to the ones presented in and respectively. We made our implementations run with the same parameters under which original results were obtained. Our experiments returned exactly the same distribution of nodes degree as the original experiments, as shown in Appendix, meaning that our implementation is consistent with the protocols’ original specifications.

Simulations for both protocols were carried out as follows. A run of a protocol is divided into three periods: creation, churn and stability. During the creation period, nodes join until reaching a given value \( N \). Neither leaves nor interleaving of joins occur along this phase. During the churn period, nodes continuously join and leave the network at the same rate (i.e., the number of nodes oscillates around \( N \)). The cumulative number of join/leave operations that occur at each time unit during the churn period is denoted as the churn rate. For example, a churn rate of 10 means that at each time unit, 10 nodes join and 10 nodes leave the system. The churn period terminates after 3000 operations have been issued, then the stability period follows during which no operation is issued. Message delay varies uniformly at random between 1 and 10 time units. \( N \) was set to 1000 in all experiments except where indicated. 10 independent runs were made for each experiment.

The metric we focus on is the average percentage of reached nodes \( (R) \), defined as the average number of nodes that can be reached from any node in the system, with respect to the total number of nodes. This metric is obviously related to the connectivity of the overlay graph, as any value lower than 100% indicates that at least one node cannot be reached by at least one other node. Moreover, we also show the size of the largest connected cluster in the system \( (LCC) \).
3.4.2 Evaluation of SCAMP

The results of experiments for SCAMP is presented in figures 6(a), 6(b) and 6(c). In all experiments the lease mechanism was not used, as it did not prove to be effective in conditions of high dynamicity. In the first experiment we tested the effect of the variation of the churn rate $C$ on $R$. Figure 6(a) shows $R$ against the execution time, with each curve corresponding to a different value of $C$ (respectively 0.02, 0.2, 2, 10 and 100 operations per time unit). Nodes leaving the system were chosen uniformly at random among those present in the system. In order to facilitate the comparison between curves, execution time is expressed as a normalized time ($\bar{t}$), where a same value of normalized time in different experiments correspond a same number of overall issued operations. At value $\bar{t} = 100$, 3000 operations have been issued and the churn period ends. At the end of the churn period, the set of initial 1000 nodes are almost completely replaced (in average, the 1% of the initial nodes remains during the entire simulation).

The plot clearly illustrates the dependence of $R$ from $C$, showing how churn rate can permanently disrupt the overlay connectivity, leading to nodes that become permanently isolated, i.e. nodes that continue their activity though partitioned from the main cluster. At the end of each run, $R$ is close to 0%, meaning that the overlay is entirely fragmented into small-sized partitions. We remark the great difference with the results showed in [5], in which $R$ starts to deviate from 100% only after 50% of the nodes have been removed from the network. In our scenario, $R$ starts to drop when only 5% of nodes have been substituted.

Another surprising result is that the main effect of disconnection is not due to the churn rate but rather to the number of overall nodes that changes (i.e., number of overall join/leave operations). This can be observed by noting that for all values of $C$ up to 10, the behavior of $R$ is almost the same, dropping to 0 when 3000 operations have been issued.$^7$

The main reason behind this behavior is that new nodes that join during the churn period in SCAMP are not integrated into the LCC, because nodes are added to the PartialView only when a node is chosen as a contact. Then, connectivity of “recently” added nodes is inherently weaker than that of nodes which stay in the system for longer. Hence, when a node inside the LCC leaves the system, the connectivity of all the nodes that used it as a contact decreases. The more the nodes that leave, the higher the probability of partitioning. The result is that despite the fact that the overall number of nodes remains the same during the run, the overlay becomes fragmented into small disconnected clusters that separates from the LCC as long as churn goes. The churn rate becomes a dominating effect over the number of overall operations only for $C = 100$. That is, the further negative impact of concurrency can be perceived only for very high values.

It is clear that, under these conditions, it becomes critical for the protocol the presence of a well-connected cluster of nodes not subject to churn. For testing this effect, in the second experiment we consider a variable percentage of nodes to be “permanent”, that is they join during the creation phase and never leave the system during the churn phase. Figure 6(b) shows the results of the experiment when changing the percentage of permanent nodes. Values

---

$^7$The fact that complete disconnection occurs exactly after 3000 operation is purely incidental, as discussed in the following
chosen were 0%, 10%, 50% and 100%. In the “Random” curve, nodes leaving the system were chosen at random, as in the previous experiment. \( C \) was set to 2 in all cases. The plot shows the positive effect of the permanent nodes over \( R \). The percentage of reached nodes during the stability period is always higher than the number of permanent nodes, meaning that the presence of a fixed connected cluster facilitates new joining nodes to be absorbed in the main cluster. Random performs better than 0% because some permanent nodes (in average the 4%) are present.

Finally, Figure 6(c) shows the dependency of \( R \) from the initial size, when the same number of operations are applied. The plot shows that continuous churn leads always to a progressive decrease of \( R \) over time, ending in complete fragmentation. Though apparently a larger network appears to be more resilient to churn, a larger initial cluster determines only a slower degradation. Please note that curve corresponding to an initial size of 3000 does not end to \( R = 0 \) only because the churn stops at \( l = 1000 \).

### 3.4.3 Evaluation of Cyclon

All experiments with Cyclon use a view size set to 7, being the logarithm of the system size. The shuffle length \( l \) is 2, while the length of random walks in the join is 5. Also for Cyclon, in the first experiment we tested the effect of the variation of the churn rate \( C \) on \( R \). The shuffle period is set to 20 time units. Differently from Scamp, we considered values of \( C \) starting from 2, being the protocol insensitive to lower values with the given shuffle period.

Results for \( R \) are plotted in Figure 6(d), while Figure 6(e) shows the size of the \( LCC \) in all cases. Again, a severe churn rate permanently disrupts the overlay connectivity, with nodes getting isolated from the \( LCC \). As a comparison with the results presented in [6], where \( R \) starts to decrease when 75% of nodes are removed, in our experiments \( R \) is lower than 100% starting from the first point (5% of substituted nodes).

There is an important difference with SCAMP: the churn rate affects significantly the trend of \( R \). With \( C = 6 \), \( R \) remains almost 100% despite the number of overall, while increasing \( C \) nodes cannot be reintegrated into the \( LCC \) resulting in a progressive decrease of \( R \). The fact that \( R \) is always much lower than the \( LCC \) size indicates clearly the presence of several small clusters of isolated nodes, that have a significant impact on \( R \) (for example, when \( R \) is 60%, the 80% of nodes belong to the \( LCC \)).

Interestingly, after the churn stops (\( l = 100 \)) there is a small raise in \( R \), while the size of \( LCC \) remains constant. This can be motivated as follows: recently joined nodes have their view not completely filled and, furthermore, they are not pointed by any other node in the \( LCC \). So, thought they are actually part of the \( LCC \), they do not contribute to \( R \). The execution of the shuffle allows new nodes to be integrated quickly into the \( LCC \), being reachable from all other nodes when the overlay converges. In overall, the continuous shuffle activity slows down the degradation of the \( LCC \), by allowing to integrate quickly the newly added nodes into it. Clearly, if the shuffle is less frequent, the overlay is more sensitive to high churn, because new nodes are more prone to isolation.

In the second experiment, we test the effect of varying the shuffling period. The churn rate is fixed to \( C = 6 \) and the shuffling timer varies from 20 to 320 time units. The results of this experiment are depicted in Figure 6(e). As
Figure 6: Experimental Results
expected, $R$ decreases faster with higher shuffling periods. Also the convergence in the stability period is slower. Obviously, increasing the shuffle frequency produces more overhead. Moreover, a continuous shuffling provokes a constant instability of the views that make them not usable for application use.
4 Publish/Subscribe for Unstructured Overlay Networks

In Section 2.3.2 we introduced various overlay architectures that can be used to implement publish/subscribe. We also cited peer-to-peer unstructured overlays as a possible substrate for event-based data dissemination, but actually not many systems have been proposed that employ such architecture. In this section we propose a novel algorithm for the implementation of publish/subscribe on top of a dynamic distributed system, exploiting the characteristics of a shuffle-based group membership protocol (namely Cyclon) previously introduced and evaluated in 3.3.2. Note that the system we propose can be used to implement any subscription model, and can thus be considered as a general approach.

4.1 Event Distribution Lists

The main problem in a publish/subscribe system is how to correctly identify the set of subscribers target for each specific event. From an abstract point of view, this problem is solved building for each event a list containing pointers (e.g. the IP address or other form of identification) to all the target processes. In the following we will refer to this list as the event distribution list (EDL).

Each EDL is characterized by the two following properties:

Completeness : the EDL contains all the subscribers target for the event;

Accuracy : the EDL does not contain any subscriber that is not interested in the event.

Note that the notion of EDL is only logical, but it is actually present in some form in any publish/subscribe system, and can be identified as the set of subscribers notified for a specific event.

Event distribution through EDLs is based on a two-steps process: first data constituting EDLs is stored in some way in the publish/subscribe system, then, when a new event is published, its EDL is retrieved to proceed with notifications.

EDLs can be stored with a centralized approach (each EDL is stored as a single piece on a process) or a decentralized one (each EDL is made up of various pieces stored on different processes).

EDLs can be retrieved exploiting a cooperative mechanism, where nodes not directly interested in the event can collaborate to the EDL retrieval, or a non cooperative one, where only the subscribers and the publisher collaborate to reach this goal.

Varying these two aspects we obtain the different approaches for event routing that we explored in 2.4. We summarized them in table 7.

<table>
<thead>
<tr>
<th>Non cooperative retrieval</th>
<th>Cooperative retrieval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscription flooding</td>
<td>Rendezvous based</td>
</tr>
<tr>
<td>Event flooding</td>
<td>Filtering based</td>
</tr>
</tbody>
</table>

Figure 7: Event distribution strategies

Varying these two aspects we obtain the different approaches for event routing that we explored in 2.4. We summarized them in table 7.

\(^8\)Gossip-based event routing here is not considered as it can be seen as a variant of Event flooding
4.2 Distribution List Accuracy

In Section 4.1 we showed how EDLs must first be stored and then retrieved when an event must be notified, and how these two parts of the process can be realized exploiting different approaches, and thus obtaining different possible implementations of the publish/subscribe system. Here we want to study how these different approaches are suited to for a dynamic peer-to-peer environment.

First of all, let us note that any cooperative approach for EDLs retrieval requires imposing some sort of structure on the overlay network, which is able to provide the minimum information needed to realize collaboration among nodes. But as we pointed out in Section 3.2 the considered dynamic model is a great obstacle to the building and maintenance of any logical structure on top of the overlay. The implementation of rendezvous-based or filtering-based event distribution algorithms on dynamic networks would thus require an enormous overhead for the simple maintenance of EDLs in order to obtain good completeness and accuracy. For this reason we think that cooperative EDL retrieval approaches are not a good choice for dynamic systems.

Non cooperative algorithms can instead be implemented with success both with centralized and decentralized EDL storage strategies. A correct implementation can be based on event flooding (EF) or on the dual subscription flooding (SF) approach [8]. With EF each event is broadcasted by the publisher in the network and matched on subscribers' side; this solution leads to a small memory footprint (because every subscription is hosted only on the original subscriber), but a large overhead as every event is sent to every node in the network regardless of its actual interest in it. From a logical point of view this approach leads to the retrieval of correct and accurate EDLs.

On the other side with SF every subscription change is broadcasted by the subscriber in the network; events are matched on the publishers' side and sent directly only to interested subscribers; this solution dramatically reduces the overhead generated for event diffusion at the price of an higher memory footprint (as every node in the system must keep track of every subscription currently active) and some overhead due to SF. Note that this overhead is exactly equal to the one generated with the EF approach only in an unrealistic scenario where each event matches exactly and only one subscription, whereas in more realistic scenarios each subscription can be matched by thousands or millions of distinct events. Thus the overhead generated by the SF approach is usually various order of magnitude lower, and this is the reason why this approach is often preferable.

The SF approach assume that each subscription update is sent to all nodes in the system. This behaviour is needed to keep subscriptions updated on the whole system, but it becomes a problem when the system itself is dynamic. In fact in a dynamic system nodes can exit the system without any warning due to an explicit leave or a fault, thus leaving on other nodes their active subscriptions that occupy useless space in memory. From this point of view this method cannot provide completely accurate EDLs in a dynamic system. Moreover our system model considers a potentially unbounded number of faulty nodes, and thus the potential quantity of wasted memory space is unbounded as well while the accuracy of the EDLs tends to zero as time passes by (i.e. the fraction of

\[ \text{The update procedure can be simplified considering it as a couple of consecutive operations where the node first proceeds with an unsubscription and then issues a new subscription (the updated one).} \]
non valid subscriptions in each EDL grows with time and tends to one). For this reason managing explicitly unsubscriptions is actually useless as the system anyway needs a mechanism to purge those subscriptions that pertain to nodes that abruptly left the network.

4.3 Improving Accuracy in Subscription-flooding based systems

Starting from the considerations done in the previous Section, we designed a new publish/subscribe system based on subscription flooding, able to address the issues present in a dynamic environment.

4.4 Basic Idea

A first simple implementation of this system can be obtained straightforward by the original protocol used for subscription flooding [8] just removing the parts related to unsubscriptions and forcing a periodic broadcast of all the active subscriptions. Each node collects subscriptions received by other nodes in a list whose size can be kept limited just purging oldest entries. The correct size of the list must be carefully estimated to effectively limit the waste of space while avoiding the exclusion of valid subscriptions. A wrong estimation of its size can rapidly bring to the retrieval of non correct EDLs.

To further limit the size of such lists (further on referred to as distribution lists) we advise the usage of advertisements to enable a form of pre-filtering. When a publisher receives a subscription from another node, it put it in the distribution list iff at least one of its advertisements matches against it. This mechanism can, depending on the specific applicative scenario, effectively reduce the size of every distribution list purging all the subscriptions that will never be matched my any event published on the corresponding node.

To further improve the performance of our approach we exploited the view shuffle mechanism introduced in [6] to realize in a different way the subscription diffusion previously obtained with plain broadcast. Subscription diffusion is realized through this simple protocol:

- each node sends its subscriptions and its advertisements to all the nodes in its view every time a view shuffle operation has completed;
- when a node receives a subscription from another node it puts it in a specific list called subscription history;
- when a node receives an advertisement from another node it matches this advertisement against its subscriptions and those contained in its subscription history. For each positive match it sends the address of the corresponding node to the node it received the advertisement from.

Aim of this protocol is to correctly keep distribution lists updated incurring less overhead than with a standard periodic broadcast. In fact our protocol requires communication only among neighbour nodes and can reach the same results obtained with periodic broadcast thanks to the fact that views change randomly and periodically.
4.4.1 Algorithm

In this section we show the detailed algorithm that implements the ideas introduced in section 4.4.

Before showing the algorithm itself, let us introduce some data structures that will be used in the following:

- $V_i$ is the partial view at node $i$
- $S_i$ is the set of subscriptions maintained at node $i$
- $A_i$ is the set of advertisements of events produced at node $i$
- $H_i$ is the subscription history at node $i$. Each element in $H_i$ is a tuple $H=<S, n, ts>$ where:
  - $S$ is a subscription;
  - $n$ is a node identifier;
  - $ts$ is a timestamp.
- $D_i$ is the event diffusion view at node $i$. Each element in $D_i$ is a tuple $D=<n, A, ts>$ where:
  - $n$ is a node identifier;
  - $A$ is an advertisement of events produced at $i$.
  - $ts$ is a timestamp.

Figures 8 and 9 report the pseudocode of the event handlers and functions respectively that constitute the algorithm.

Each time a node updates its view, it sends two messages containing its sets of subscriptions and advertisements to all nodes in the updated view.

As soon as a node $i$ receives a subscription set from another node $x$ it adds all the subscriptions in the set $S_x$ to its subscription history $H_i$, along with the timestamp of the moment they were added and a reference to $x$. To complete this operation some technique must be employed to ensure that the length of the history remains bounded to a predefined constant; in this sense, a simple policy would just throw away those entries with the oldest timestamp, as they are with higher probability related to subscriptions that are no longer present in the system.

The history is used by each node to match advertisements against known subscriptions. As soon as a node $i$ receives an advertisement set from another node $x$ it tries to match the advertisements against its own subscriptions, and those contained in $H_i$. For each satisfied subscription, it sends a message MATCH to $x$ containing the matching advertisement $A$, the address of the node owning the matched subscription, and the timestamp of the corresponding entry in $H_i$ (note that when one advertisement matches one of the subscriptions managed by $i$, the timestamp sent to $x$ is equal to current time).

The MATCH messages are used by each node to build their event diffusion views. $D_i$ actually represents an approximate view of the set of nodes potentially interested in events produced at $i$, thus it can be used when publishing events to decide where to forward the event itself. Actual event filtering happens on
on viewchange do
    for each $x \in V_i$ do
        send SUBS[$S_i$] to $x$
        send ADVR[$A_i$] to $x$
    done

on receive SUBS[$S_i$] do
    for each $S \in S_i$ do
        $H_i \leftarrow < S, x, \text{getTimestamp}() >$
    done

on receive ADVR[$A_i$] do
    for each $A \in A_i$ do
        for each $S \in S_i$ do
            if match($A, S$) then
                send MATCH[$i, A, 0$] to $x$
            done
        for each $H \in H_i$ do
            if match($A, H.S$) then
                send MATCH[$H.n, A, H.ts$] to $x$
            done
    done

on receive MATCH[$n, A, ts$] do
    $D_i \leftarrow < n, A, ts >$
    done

on receive PUBLISH[$e$] do
    for each $S \in S_i$ do
        if match($e, S$) then
            notify($e$)
    done

function publish($e$)
    for each $D \in D_i$ do
        if $e \in H.A$ then
            send PUBLISH[$e$] to $H.n$
    done
end function

Figure 8: Event handlers for the Publish/Subscribe algorithm
function publish(e)
    for each $D \in D_i$ do
        if $e \in H.A$ then
            send PUBLISH[e] to $H.n$
        end if
    done
end function

function subscribe(S)
    $S_i \leftarrow S$
end function

function advertise(A)
    $A_i \leftarrow A$
end function

Figure 9: Functions for the Publish/Subscribe algorithm

the subscriber side as soon as it receives a PUBLISH message from some other node. Note that, as it happens with $H_i$, also for $D_i$ a mechanism is needed to avoid its unbounded growth; in this case too we suggest a simple strategy that aims at deleting from the event diffusion views those entries with the oldest timestamp.
5 Conclusion

In this report we showed the research effort we are conducting on the development of a new idea for the implementation of a publish/subscribe system for dynamic peer-to-peer systems.

We first introduced the publish/subscribe communication paradigm with an in-depth analysis of all the issues involved in the definition of a correct architecture for its implementation.

Then we introduced dynamic peer-to-peer systems and showed how we can address their dynamic behaviour, and the related problems, using unstructured overlay networks.

At last we presented a novel idea for the implementation of publish/subscribe on an unstructured overlay network for dynamic systems. The proposed system exploits the underlying overlay network maintenance protocol to enhance its performance.

Currently we are working on the definition of an architecture in order to implement such new algorithm and then conduct some tests to evaluate its performance.

References


38


40


