Reconfiguration Primitives for Self-adapting Overlays in Distributed Publish-Subscribe Systems

Elisabetta Di Nitto, Daniel J. Dubois, Alessandro Margara
Politecnico di Milano, Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32, Milano, Italy
{*dinitto,dubois,margara*}*@elet.polimi.it*

***Abstract*—Most distributed applications involve some form of event-based interaction, often implemented using a *publish-subscribe (pub-sub) infrastructure*. To improve scalability, the acts of matching events against subscriptions and delivery them are performed collaboratively by a set of *brokers* connected into an overlay network. Recent research has proposed several approaches to support the self-adaptation of such overlay network to adapt it to changes in application traffic. However these approaches focus on the *monitor, analyze, plan* parts of the self-adaptation loop, without considering the issues that arise in the *execution* part. This paper proposes a set of primitives that fills the gap in the execution phase. Compared to existing work, our approach: (*i*) is transparent w.r.t. the routing policies of the middleware, (*ii*) preserves existing properties and guarantees of the middleware, such as no duplication of events, causal ordering, and minimal delays for the events delivered during a reconfiguration. We discuss the correctness of our primitives and implement them in a simulated environment to measure their cost in terms of network overhead.**

***Keywords*-Pub-sub systems; topology management; self-adaptation actions;**

I. Introduction

The last decade has seen the development of a large number of pub-sub infrastructures [20], [2], [10], [19], [8], [5], coming from academia and industry. These infrastructures enable distributed components to *subscribe* to the *event notifications* (or simply "events") they are interested to receive, and to *publish* those they want to spread around [17]. To improve scalability, many of them provide distributed solutions [6], [4], in which the processes of matching events against subscriptions and delivering them to interested recipients are performed collaboratively by a set of *brokers* connected into an overlay network. Researchers and practitioners working on pub-sub infrastructures have mainly focused on the issues of providing efficient protocols to route information and split the processing load among available brokers [6], [4].

This work focuses on dynamically self-adapting the topology of the overlay network to the application traffic. This aspect becomes of primary importance today, as more and more messaging infrastructures are offered as a service in the Cloud. For instance, Amazon EC2 provides a Simple Notification Service (SNS) to deliver notifications; VMware CloudFoundry includes messaging API; Microsoft Azure offers a Service Bus supporting a complete pub-sub model. In this context, the cloud provider has a complete control over the network of brokers, and frequent overlay reconfigurations become one of the core mechanisms to achieve elasticity, which allows to rapidly scale-in to tolerate sudden network spikes, and scale-out to minimize the usage of processing resources. Moreover, systems like EC2 offer computational resources from different, geographically separate, zones. This is completely transparent from the point of view of the application, but not from the point of view of the infrastructure because traffic between different zones is expensive and has a higher latency. In this context, a reconfiguration mechanism that optimizes the routes taking zones into account becomes even more important.

While some self-adaptive solutions have been proposed in the literature to reconfigure the overlay network and optimize the information flow [21], [18], [9], [24], they are all hampered by the fact that during the reconfiguration phase, events may be lost, duplicated, received out of order, and with increased delay. This lack of guarantees reduces the applicability of overlay reconfiguration algorithms in contexts where strong guarantees are required, e.g., financial applications for online trading.

This paper introduces three novel *generic* and *transparent* overlay reconfiguration primitives, which realize the execution phase of the self-adaptation loop of a pub-sub system and are transparent w.r.t. the other phases of the self-adaptation logic (monitor, analysis, plan). Our primitives are generic: they work with every pub-sub routing protocol that relies on a tree topology [7], or that builds multiple tree topologies on top of a generic topology [6]. Moreover they are also transparent: they do not impact on the guarantees offered by the underlying pub-sub system. Most importantly, they preserve guaranteed delivery with no duplication of events, and causal ordering.

In this paper we (*i.*) describe our reconfiguration primitives, (*ii.*) discuss their correctness and properties, (*iii.*) show how our they can be adopted to support existing self-adaptive reconfiguration strategies, and (*iv.*) evaluate them in a simulated environment, measuring the reconfiguration cost in terms of network traffic under different scenarios.

Consistently, Section II introduces the terminology we use in the rest of the paper; Section III motivates the work and provides a classification of existing reconfiguration approaches; Sections IV and V describe our primitives and highlight their properties; Section VI provides an evaluation of our primitives in a simulated environment. Finally, Sec-

tion VII provides some conclusive remarks.

II. System Model

We consider a distributed pub-sub infrastructure, composed of a set of brokers B, connected into an *overlay network*, serving a set of clients C, which act as producers, *publishing* events, and consumers, *subscribing* to events.

The overlay network. We model the overlay network as a tree $T = (B, L)$ where B is the set of brokers and L is the set of bidirectional overlay links connecting brokers together. We represent a link between two brokers $b_1, b_2 \in B$ as $\langle b_1 b_2 \rangle$. Moreover, we represent a path (composed of one or more adjacent links) used to send information from broker $b_1 \in B$ to broker $b_2 \in B$ as $b_1 \rightarrow b_2$. We use the notation $b_1 \leftrightarrow b_2$ when we do not need to specify the direction of the communication. We write $b_1 \rightarrow b_3 \rightarrow b_2$ to specify that the path between b_1 and b_2 includes broker b_3. Since we are considering a tree topology, we always have a unique simple path (i.e., a path that does not have repeated brokers) between every two brokers $b_1, b_2 \in B$. We define the set N_b of neighbors of a broker $b \in B$ as the set of brokers directly connected to b through a link $l \in L$. Each client $c \in C$ is connected to a single broker $b \in B$ at any point in time. We call C_b the set of clients connected to b.

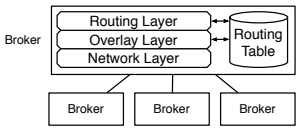


Figure 1: Architecture of a pub-sub system

The architecture of brokers. Figure 1 shows the abstract architecture of a broker $b \in B$. We consider three layers: from the bottom, the `Network Layer` handles the network connections with other brokers and clients and manages the low level communication details. It is used by the `Overlay Layer`, which keeps information about the overlay network, i.e., about links and neighbors. The `Overlay Layer` hides the `Network Layer` to the `Routing Layer`, which is responsible for processing and forwarding events and subscriptions. The `Routing Layer` uses the `Routing Table` to store information about the subscriptions received from other brokers and local clients. In some pub-sub models the information stored in the `Routing Table` is the same for all brokers. In others each broker stores in its `Routing Table` only those subscriptions that concern itself and its direct neighbors. Our approach does not impose a specific format for events and subscriptions: it works both with a *topic-based* and with a *content-based* model [10]. We call S the set of all possible subscriptions and E the set of all possible events. Each client $c \in C$ specifies its interests by providing a (possibly empty) set of subscriptions S_c. We assume a *matching function* is defined as $M : E \times S \rightarrow \{1, 0\}$. Given an event $e \in E$ and a subscription $s \in S$, M returns 1 if e matches s and 0 otherwise. An event e has to be delivered to every client $c \in C$ such that $\exists s \in S_c$ — $M(e, s) = 1$. Our reconfiguration primitives operate at the `Overlay Layer`, by modifying the links in L. They are designed to be generic and work with different routing protocols. Moreover, they are completely transparent for the `Routing Layer`, which is not aware of the changes in the overlay. To achieve this transparency, our primitives require to access and modify the information stored in the `Routing Table`. We model this behavior in Figure 1, by allowing the `Overlay Layer` to read and write from the `Routing Table`. As we will better show in the following, depending on the specific routing protocol adopted, it may be possible to exploit the information stored in the `Routing Table` to optimize some reconfiguration primitives.

III. Background and Motivation

This section revises related approaches, motivating the need for our contribution.

Existing solutions. Based on the analysis performed on the literature, we present some classification criteria for reconfiguration existing strategies.

Assumption on Topology. This criterion specifies the assumptions made by the reconfiguration algorithm on the structure of the topology. It can assume a specific structure that can be *cyclic* (a generic graph) or *acyclic* (a *tree*), or it can make no assumption on such structure. Algorithms assuming a specific structure can rely on it to support the reconfiguration procedure; however, they have to guarantee that the structure is kept also after each reconfiguration. On the contrary, algorithms that make no assumptions have to rely on redundant approaches for transmitting reconfiguration messages (e.g., gossiping) but do not have to worry about causing possible disruptions to the topology structure.

Locking Strategy. A reconfiguration process may *lock* a subset of the brokers, preventing them to participate in other reconfigurations. This locking strategy specifies which brokers are locked by a reconfiguration process, and so the maximum number of reconfigurations that can occur concurrently. We distinguish among four kinds of strategies: (*i*) *Global*: the reconfiguration locks all the brokers in B. (*ii*) *Path*: the reconfiguration locks two brokers p and q in the system, plus all the brokers included in the path $p \leftrightarrow q$. (*iii*) *Local*: the reconfiguration only locks one broker, plus at most a subset of its neighbors that are involved in the topological changes. (*iv*) *None*: the reconfiguration does not lock any broker.

Impact on fault tolerance. The use of reconfiguration may provide the pub-sub infrastructure with a level of fault tolerance that is different w.r.t. the one offered in the absence of reconfiguration. We classify this aspect in the following way: (*i*) *Weak*, in which the level of fault tolerance in terms of damage to the topology and its traffic is lower during the reconfiguration. Weak fault tolerance means that during a reconfiguration the whole system becomes more vulnerable to faults. (*ii*) *Standard*, in which the level of fault tolerance in terms of damage to the topology and its traffic during the reconfiguration is equivalent to the one the middleware has when it is not performing a reconfiguration.

Standard fault tolerance means that during a reconfiguration the whole system does not become more vulnerable to faults. (*iii*) *Strong*, in which the reconfiguration increases the level of fault tolerance of the system.

Guaranteed Delivery. For each event $e \in E$, we define the set of clients interested in e as $C_e \subseteq C = \{c \in C \mid \exists s \in S_c, M(e,s) = 1\}$. A reconfiguration provides guaranteed delivery iff e is eventually received by all clients in C_e.

No Duplicated Delivery. Given an event $e \in E$ and the set C_e of clients interested in e, a reconfiguration exhibits no duplicated delivery iff each client $c \in C_e$ receives e at most once. If a system provides both guaranteed and no duplicated delivery, we say that it provides *exactly once delivery*.

Ordering Guarantees. Given two events $e_1, e_2 \in E$ produced by a publisher client $p \in C$ one after the other (e_1 first and then e_2), and the set of clients interested in both events $C_{e_1,e_2} = C_{e_1} \cap C_{e_2}$, a reconfiguration exhibits *FIFO ordering guarantee* iff $\nexists c \in C_{e_1,e_2}$ such that c receives event e_2 before event e_1. A stronger guarantee is called *causal ordering guarantee* and is defined as follows: consider two events $e_1, e_2 \in E$ the set of clients C_{e_1,e_2} interested in both e_1 and e_2. A reconfiguration exhibits causal ordering guarantee iff, when e_1 happens-before. e_2, then $\nexists c \in C_{e_1,e_2}$ such that c receives e_2 before e_1.

Minimal delay. Consider two brokers $p, q \in B$. Let us define Lat_{pq}, called *average latency of the path* $p \leftrightarrow q$, as the average time for sending publications or subscriptions from p to q. If we denote with Lat_{pq} the latency of the path before the reconfiguration, with Lat^*_{pq} the maximum latency of the path during the reconfiguration, and with $\overline{Lat_{pq}}$ the latency of the path after the reconfiguration, then the reconfiguration algorithm satisfies the *Minimal delay* property iff $Lat^*_{pq} \leq \max(Lat_{pq}, \overline{Lat_{pq}})$ for all possible $p, q \in B$ under the following two assumptions: (*i*) the latency added by operations that are not controlled by the reconfiguration algorithm (e.g., the updates to the `Routing Tables`) are not considered in the calculation of the latencies; (*ii*) the bandwidth of the path $p \leftrightarrow q$ is not significantly reduced by the overhead added by the reconfiguration protocol. We will measure this overhead for our approach in Section VI, showing that it is negligible. Intuitively, the *Minimal delay* property is satisfied by algorithms that do not block the forwarding of publications and subscriptions during the reconfiguration.

Table I: Classification of Reconfiguration Algorithms

				Guarantees			
Algorithm	**Top**	**Lock Strat**	**Fault Tol**	**Guar Del**	**No Dup Del**	**Order Guar**	**Min Del**
Picco [22]	T	Path	Stand.	✓	✓	Causal	
Baldoni [1]	T	Path	Stand.	✓	✓	Causal	
Parzyjegla [21]	T	Local	Stand.	✓	✓	Causal	
Dubois [9]	T	Local	Stand.	✓	✓	Causal	
Yoon [24]	T	Local	Stand.	✓	✓	Causal	
Bhola [3]	C	Global	Stand.	✓	✓	None	
Kim [15]	U	Global	Stand.			None	✓
Loulou [16]	C	Local	Stand.	✓		None	✓
Guo [12]	U	None	Strong		✓	None	✓

Why a New Approach?. Table I summarizes all relevant approaches we are aware of that implement a solution to the execution phase of our self-adaptation loop. Each approach is named by the last name of its first author. We immediately observe that none of the existing proposals is able to satisfy all the four guarantees we have identified (last four columns).

The works in [22], [1], [21], [9], [24] fulfill all the guarantees under classification, except for the minimal delay property. All these works use a tree-based topology with different reconfiguration strategies and standard fault tolerance. The reason why the minimal delay property is violated is because their locking mechanisms are implemented in such a way that the traffic generated during a reconfiguration is stored in a queue until the reconfiguration is completed. Locking the traffic will then result in a delay increase for delivering events. The works in [3], [15], [16], [12] make less strict assumptions on the topologies, therefore they may not need to lock the traffic on reconfiguring brokers because they can use alternative paths for forwarding events. However, the cost for such flexibility is the loss of ordering guarantee in the delivery of events. Indeed, unstructured overlays fail to provide guarantees on the delivery and on the ordering without proper more complex end-to-end mechanisms.

The approach we are proposing in this paper relies on a structured graph topology on top of which trees are used to deliver events, with a standard level of fault tolerance, and a local reconfiguration strategy. In the following we show that our work is currently the only one that, under a proper set of assumptions discussed in the next section, ensures at the same time guaranteed and no duplicated delivery, causal ordering, and minimal delay.

IV. Reconfiguration Primitives

This section presents our reconfiguration primitives in detail. They consider a tree topology, and transform it into a new tree topology[1]. The primitives are: (*i*) `SwapLink`(a,b,c) Given three brokers a, b, and c, such that $a, c \in N(b)$, it removes the link connecting a and b, and creates a new link connecting a and c. (*ii*) `AddBroker`(a,b) Adds a new broker b to the overlay network, connecting it to broker a; (*iii*) `RemoveBroker`(b) Removes a broker b from the overlay network.

For ease of exposition, we focus on the brokers involved in each reconfiguration and model all the others connected to any of them, let us say b, as a virtual broker B_b connected through a link $\langle bB_b \rangle$. We use the term *reconf message* to indicate the messages used by our reconfiguration primitives that work at the `Overlay Layer`. Conversely, we call *pub-sub messages* the messages (i.e., events and subscriptions) handled at the `Routing Layer`.

Assumptions. Our primitives rely on the following assumptions: (*i*) brokers and links do not fail; (*ii*) links are FIFO; (*iii*) brokers process and forward reconf messages in the order in which they are received. The first two properties

[1]Our approach can be trivially extended to generic structured topologies that use multiple trees for the delivery of events, by applying the primitives independently on each of these trees in a compositional way.

are inherited from tree-based pub-sub systems [10], while the last one ensures FIFO ordering of reconf messages along arbitrary paths (composed of one or more links); moreover, it ensures that FIFO ordering is preserved between a reconf message and pub-sub messages flowing over the same path. On the contrary, we do not impose any limitation in the order in which pub-sub messages are processed. For example, brokers may process different events in parallel to increase performance: in this case they may not preserve FIFO ordering of events over multi-hop paths.

A. SwapLink

As shown in Figure 2, the `SwapLink` (a, b, c) primitive considers three brokers, $a, b, c \in B$, such that broker b is initially connected to both a and c, i.e. $a, c \in N(b)$. Since we consider tree topologies, there is no link connecting a and c, i.e., $c \notin N(a)$. The primitive removes the link $\langle ab \rangle$ and adds a new link $\langle ac \rangle$. In other words, it removes a from the neighbors of b and adds it to the neighbors of c.

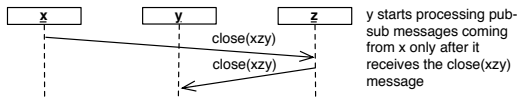
After creating the new link, `SwapLink` has to introduce 4 changes in the paths used to route pub-sub messages: (*i*) pub-sub messages forwarded from a to c use the new link $\langle ac \rangle$; (*ii*) the same holds for pub-sub messages moving from c to a; (*iii*) pub-sub messages from a to b are now forwarded through c using the path $a \rightarrow c \rightarrow b$; (*iv*) similarly, pub-sub messages from b to a use the path $b \rightarrow c \rightarrow a$. Before describing `SwapLink` in detail, we introduce the `ChangePath` procedure. It ensures that events are received by the reconfiguring brokers in a FIFO order. This is important to ensure that the guarantees offered by the `Routing Layer` are not altered by the reconfiguration.

(a) Before reconfiguration (b) After reconfiguration

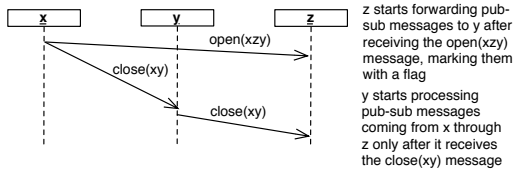
Figure 2: SwapLink: nodes involved

(a) Using path $x \rightarrow z \rightarrow y$ (b) Using link $\langle xy \rangle$

Figure 3: ChangePath - Possible paths between x and y



(a) From $x \rightarrow z \rightarrow y$ to $\langle xy \rangle$



(b) From $\langle xy \rangle$ to $x \rightarrow z \rightarrow y$

Figure 4: ChangePath - Reconf messages exchanged

The ChangePath procedure. All the changes of paths adopted in our primitives involve three brokers. In this specific situation, there are two ways for changing a path $x \rightarrow y$ between two brokers $x, y \in B$: moving from the use of the link $\langle xy \rangle$ (see Figure 3b) to a path $x \rightarrow z \rightarrow y$ that involves a third broker $z \in B$ (see Figure 3a), and viceversa. Our `ChangePath` procedure addresses both cases.

Figure 4a shows the reconf messages exchanged for moving from $x \rightarrow z \rightarrow y$ to $\langle xy \rangle$: before start using the new path, broker x sends a `close(xzy)` message on the old path, to notify y that it will not receive other pub-sub messages from that path. This is guaranteed by the assumptions on FIFO links and ordered processing of reconf messages w.r.t. pub-sub messages. When x starts sending pub-sub messages using the new channel $\langle xy \rangle$, y may receive pub-sub messages from the new channel while other (previous) pub-sub messages are still flowing through the old channel. y stores these pub-sub messages in a temporary queue until it is sure that no more pub-sub messages are coming from the old channel, i.e., until it receives the `close(xzy)`. This ensures that the `Routing Layer` receives all the pub-sub messages from the old path before starting to receive those coming from the new one. If the pub-sub system ensures FIFO ordering of pub-sub messages over arbitrary paths, this property continues to hold during the reconfiguration.

Figure 4b shows the exchange of reconf messages for changing the communication channel from $\langle xy \rangle$ to $x \rightarrow z \rightarrow y$. In this case, we need to notify z about the change, asking it to route pub-sub messages from x to y. To do so, x sends an `open(xzy)` message to z. As in the previous case, x also sends a `close(xy)` message to y, which is the last one to travel through the old channel (i.e., $\langle xy \rangle$). Also in this case, the `Overlay Layer` of broker y starts delivering pub-sub messages coming from the new channel to the `Routing Layer` only after it receives the `close(xy)`, thus ensuring that the procedure does not violate the ordering assumptions done by the pub-sub system. Notice that y may receive from z both pub-sub messages that are coming from x and messages that are coming from other sources, e.g., the ones generated by the clients of z. These last ones do not need to wait for the `close(xy)` message to be processed, and can be delivered to the `Routing Layer` immediately. Thus, to let y distinguish between the two cases, broker z marks the messages from x with a special flag. The last `close(xy)` message notifies broker z that it can safely stop to flag pub-sub messages forwarded from x, since broker y has already received the `close(xy)` message.

The SwapLink primitive. We are now ready to describe the `SwapLink` (a, b, c) primitive. Figure 5 shows the reconf messages exchanged among a, b, and c. We assume the process always starts at broker a, reserving a `start` message to show the triggering of the procedure from other nodes.

Each broker has an associated `State`, which defines its behavior. A broker in `State 0` is not involved in

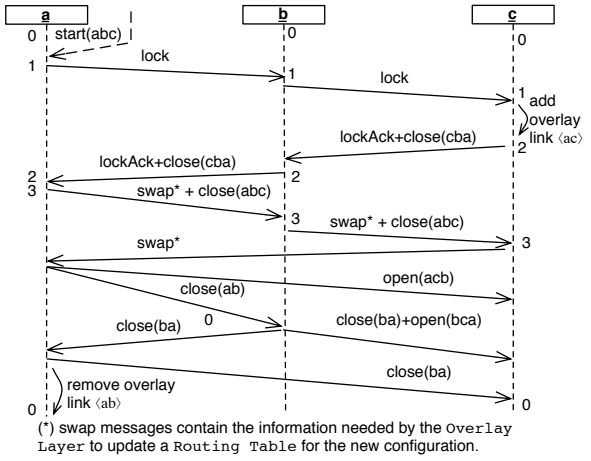


Figure 5: SwapLink - Reconf messages exchanged

a reconfiguration, it adopts the normal routing policy as defined in the `Routing Layer`. The other states, from 1 to 3, identify different steps of the reconfiguration process. When a broker is in one of these states the routing policies can be altered by the `Overlay Layer`. Figure 5 shows the exact point in time in which a broker enters a new state (labels 0, 1, 2, and 3). Intuitively, the `SwapLink` primitive performs the following steps:

- It locks the three brokers involved in the reconfiguration, making sure that they are not already participating in another reconfiguration process.
- It creates the new link $\langle ac \rangle$. Each reconfiguring broker forwards pub-sub messages to the other two: duplicates are avoided by marking pub-sub messages with a special flag.
- It forwards the information required to update the `Routing Table` to correctly forward pub-sub messages after the reconfiguration.
- It removes the link $\langle ab \rangle$ and starts adopting the new routing policies. Finally, it unlocks the brokers, which become ready for other reconfigurations.

In the following we provide details on the above steps.

State 0. As a result of the `start` message, broker a sends a `lock` message to b, asking it to lock for a reconfiguration involving c. If b is already locked in another reconfiguration, it replies with a `lockNack` that immediately stops the process. Otherwise, it forwards the `lock` to c. Depending from the fact that it is free or already locked, c replies with a `lockAck` or `lockNack` that b forwards to a.

State 1. When a (non locked) broker sends or receives a `lock` message, it moves to `State 1`. From this state, it continues to use the previous routing policies, but is not able to accept other requests for reconfiguration. When c moves to `State 1`, it also creates the new overlay link $\langle ac \rangle$.

State 2. When broker c has finished creating the overlay link $\langle ac \rangle$, it moves to `State 2`. When the other two brokers a and b receive a `lockAck` message, they move also to `State 2`. When in `State 2` any broker $x \in \{a, b, c\}$ considers the new link $\langle ac \rangle$ as open. Thus, upon receiving a pub-sub message e from a sender s, it behaves as follows:

- If s is not one of the brokers under reconfiguration, i.e. $s \notin \{a, b, c\}$, x forwards e to all *non* reconfiguring brokers using the previous routing policies. Moreover, it sends e to both reconfiguring brokers, marking it with a special *flood* header.
- If $s \in \{a, b, c\}$ and e does not have the *flood* header, then the sender was not yet in `State 2` when it delivered e. Broker x processes e using the previous routing policies.
- If $s \in \{a, b, c\}$ and e has the *flood* header, then the sender was in `State 2` when it delivered e. Broker x removes the header from e and processes it using the previous routing policies, but it does *not* forward e to the other brokers involved in the same reconfiguration.

Intuitively, following this procedure, old pub-sub messages (from brokers in `State 1`) are forwarded using the old routing policies; new pub-sub messages (from brokers in `State 2`) are forwarded to both reconfiguring neighbors only once, thus avoiding duplicates. There are a few things to note. A reconfiguring broker in `State 2` forwards pub-sub messages coming from external brokers to both reconfiguring neighbors. This does not change the communication paths of pub-sub messages coming from b, but it changes the paths $a \rightarrow c$ and $c \rightarrow a$, using the new link $\langle ac \rangle$. To realize these two changes of path, we make use of the `ChangePath` as described in Figure 4a. The `ChangePath` (from $c \rightarrow b \rightarrow a$ to $\langle ca \rangle$) is realized with a `close(cba)` message sent in $\langle cb \rangle$ and in $\langle ba \rangle$. The `ChangePath` (from $a \rightarrow b \rightarrow c$ to $\langle ac \rangle$) is realized with a `close(abc)` message sent in $\langle ab \rangle$ and in $\langle bc \rangle$.

State 3. When a broker sends or receives a `swap` message, it moves to `State 3`. Routing is performed as in `State 2`; the `swap` messages are used to ensure the reconfiguring brokers that the transition to `State 2` has ended everywhere. Accordingly, a broker in `State 3` cannot receive pub-sub messages from brokers in `State 1`. Depending on the specific routing protocol adopted by the system, brokers may need to exchange information about subscriptions to correctly populate the `Routing Table` and support the new topology after the reconfiguration. If needed, we embed this information inside `swap` messages. The information stored in the `Routing Table` of each node has another implication: in protocols that provide every broker with a global knowledge about the subscriptions of the network, we can exploit this information to optimize the delivery of pub-sub messages in `State 2` and `State 3`. In particular, in the case of publications, reconfiguring brokers can filter events instead of flooding them, thus avoiding the propagation of unneeded events among reconfiguring brokers.

State 0 (new configuration). After receiving a `swap` message, a broker has all the information required to adopt the new routing policies. This already happens for the communication between a and c, which uses the new link

$\langle ac \rangle$. We now need to close the link $\langle ab \rangle$ (done by broker a) and start using the new paths $a \rightarrow c \rightarrow b$ and $b \rightarrow c \rightarrow a$. This is implemented through two `ChangePath` procedures, as shown in Figure 5. The `ChangePath` (from $\langle ab \rangle$ to $a \rightarrow c \rightarrow b$) is realized with an `open(acb)` sent in $\langle ac \rangle$, and a `close(ab)` sent in $\langle ab \rangle$ and in $\langle bc \rangle$. The `ChangePath` (from $\langle ba \rangle$ to $b \rightarrow c \rightarrow a$) is realized with an `open(bca)` message sent in $\langle bc \rangle$, and a `close(ba)` message sent in $\langle ba \rangle$ and in $\langle ac \rangle$. We save a reconf message by combining the `close(ab)` and the `open(bca)` messages sent from b to c. When a broker enters `State 0`, it completely adopts the new routing policies; from this moment on, it is unlocked and free to participate in other reconfigurations.

B. AddBroker

As shown in Figure 6, the `AddBroker`(a, b) primitive increases the size of the overlay network by adding a new broker b and connecting it to an existing broker a through a link $\langle ab \rangle$. The exchange of reconf messages defined by the `AddBroker` primitive is shown in Figure 7: broker b sends a `connect` message to a and receives a `connectAck` back. Only after receiving this message, b considers the link $\langle ab \rangle$ as open. The primitive does not change the original connection between a and the remaining part of the overlay (i.e., B_a). Moreover, it does not impact on the routing protocols being used: indeed, b is connected through a single link $\langle ab \rangle$ and does not have local clients ($C_b = \emptyset$). Accordingly, there is no traffic flowing along the link $\langle ab \rangle$ until some client is connected to b and starts to generate traffic. The `AddBroker` primitive does not lock broker a: in other words, multiple brokers may start connecting to a concurrently. On the contrary, broker b becomes available for another reconfiguration only after the process of connecting it with broker a is complete, i.e., after receiving the `connectAck` message.

There are two reasons for adding new brokers: to serve future clients or to perform some form of load balancing. Protocols for moving clients have been already addressed in the literature [13] and are outside the scope of this paper.

(a) Before reconfiguration (b) After reconfiguration

Figure 6: AddBroker: nodes involved

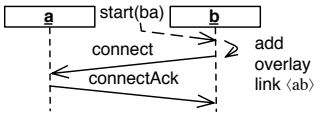


Figure 7: AddBroker - Reconf messages exchanged

C. RemoveBroker

The `RemoveBroker`(b) primitive removes a broker $b \in B$ from the overlay. As shown in Figure 8, the primitive requires b to have a single neighbor, represented as a. It removes the link $\langle ab \rangle$, preserving all the other

(a) Before reconfiguration (b) After reconfiguration

Figure 8: RemoveBroker: nodes involved

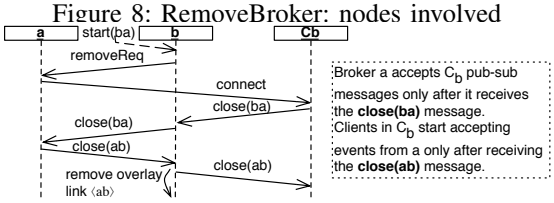


Figure 9: RemoveBroker - Reconf messages exchanged

links in the overlay network, modeled through B_a. The `RemoveBroker` primitive works even when $C_b \neq \emptyset$ and moves every client $c \in C_b$ to broker a. As for the other primitives, it preserves guaranteed delivery, no duplicated delivery and causal order of pub-sub messages. To do so, however, it requires some logic to be deployed inside the clients.

Figure 9 shows the exchange of reconf messages generated by the `RemoveBroker` primitive. Without loss of generality, we assume the process starts at b: as for the `SwapLink` primitive, we reserve a `start` to force the process from other brokers. Broker b first sends a `removeReq` message to a, which contains the identifier (e.g., the network address) of all the clients in C_b. If $C_b = \emptyset$, b is removed from the overlay network and the process terminates. In the case $C_b \neq \emptyset$, a uses the identifiers of the clients in C_b to contact them, sending a `connect` message. From this moment, the clients stop sending pub-sub messages to b and start sending them directly to a. The subsequent reconf messages realize two `ChangePath` procedures, the first one changing the communication path from the clients in C_b to a, and the second one changing the communication path from a to the clients in C_b, without loss of pub-sub messages (see Section V, property 3).

V. Analysis of the Primitives

This section provides a detailed analysis of our primitives. In particular we give proofs or intuitive explanations for the capability of the primitives to have the following properties.

Property (1). "The primitives transform tree topologies into tree topologies."

`SwapLink` **case.** Consider the brokers involved in the `SwapLink` primitive, reported in Figure 10a. In this figure the overlay is represented divided into three interconnected subtrees ($\{a, B_a\}$, $\{b, B_b\}$, and $\{c, B_c\}$), where a, b, c are the reconfiguring brokers and B_a, B_b, B_c represent external brokers that are (directly or indirectly) connected to them.

(a) Before reconfiguration (b) After reconfiguration

Figure 10: Brokers involved in a SwapLink process

If we remove the link $\langle ab \rangle$, we are partitioning the original tree into two subtrees. The first one includes nodes a and B_a; let us call it T_a. The second one includes nodes b, B_b, c, and B_c; let us call it T_{bc}. If we merge again T_a and T_{bc} using the new link $\langle ac \rangle$ (see Figure 10b), we obtain a tree topology again. Indeed, we know from the graph theory that joining two trees through a single link produces another tree.

`AddBroker` **case.** From the graph theory we know that a graph $G = (B, L)$ is a tree if and only if it is connected and $|L| = |B| - 1$. Now assume we have a topology with n brokers ($|B| = n$) and $n - 1$ links ($|L| = n - 1$). The `AddBroker` primitive adds one broker b and one link, obtaining $n + 1$ brokers and n links. Moreover, the new graph is connected since the new link connects b to the (connected) graph G.

`RemoveBroker` **case.** Similarly, the `RemoveBroker` primitive removes one broker b and one link. Accordingly, it generates a new graph $G = (B, L)$ where $|B| = |L| - 1$. G is connected, since b was a leaf in the original graph. Hence, G is a tree.

Property (2). "The primitives transform a tree topology to every other tree topology (after a finite number of reconfigurations)." A formal demonstration of this property can be found in [24], where the authors provide primitives for adding and removing brokers and for swapping links that result in topology transformations that are equivalent to the ones we propose. Even if the cited work enables the same transformations as our primitives, it is not able to exhibit the other guarantees we offer.

Property (3). "The primitives preserve guaranteed delivery, if it is offered by the pub-sub system." If the pub-sub middleware ensures guaranteed delivery through end-to-end mechanisms (e.g., through retransmission), then the property is maintained. Indeed, these mechanisms are built on top of the `Routing Layer` and are independent from the way in which pub-sub messages are routed and processed. On the contrary, some pub-sub system ensures guaranteed delivery by enforcing it at every link of the overlay network. We now analyze this case, showing that our primitives do not dirsupt guaranteed delivery.

`SwapLink` **case.** In the `SwapLink` primitive, all the brokers in `State 0` and `State 1` follow the existing routing policies; if the system offers guaranteed delivery, it still does during this reconfiguration. Brokers in `State 2` and `State 3` send all pub-sub messages to both reconfiguring neighbors; therefore, no pub-sub message can be lost.

`AddBroker` **case.** The `AddBroker` primitive consists in the insertion of a single broker. Until the new link is finally activated, there is not traffic of pub-sub messages flowing through it. Existing clients are served by the brokers already present in the overlay network, adopting existing routing policies. Therefore, no pub-sub message can be lost.

`RemoveBroker` **case.** The `RemoveBroker` primitive consists in the removal of a broker b. In our approach every client $c \in C_b$, initially connected to b, moves to a broker a before the link $\langle bc \rangle$ is invalidated. Therefore, c can continue to use the previous path for sending pub-sub messages until it is available. c also receives all pub-sub messages, because they all flow through a, which stops forwarding them to b only when a connection with each client in C_b is established.

Property (4). "The primitives preserve no duplicated delivery, if it is offered by the pub-sub system." Similarly to Property (3), if the middleware offers this property using end-to-end mechanisms that do not make any assumptions at the `Overlay Layer`, our primitives do not interfere with them, since they work at a higher layer. We now show that our approach does not disrupt no duplicated delivery by avoiding duplicates at the level of links or paths.

`SwapLink` **case.** In the `SwapLink` primitive, all the brokers in `State 0` and `State 1` follow the existing system routing policies; therefore, if the system offers no duplicated delivery, it still does during the reconfiguration. All brokers in `State 2` and `State 3` send all pub-sub messages to both reconfiguring neighbors, but with a special *flood* header that inhibits the reforwarding to other reconfiguring neighbors; therefore, no pub-sub message can be duplicated.

`AddBroker` **case.** There is no traffic of pub-sub messages during the reconfiguration because the new broker has no clients, therefore no pub-sub message can be duplicated.

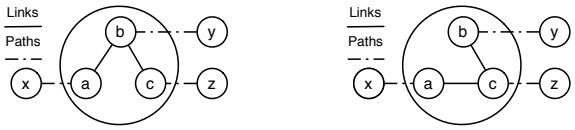
`RemoveBroker` **case.** In the `RemoveBroker` primitive (using the client migration approach proposed in Section IV), a client $c \in C_b$ is moved from a broker b to a broker a. It sends pub-sub messages only once, either to b, before the reconfiguration, or to a, after the reconfiguration. Thus, no pub-sub messages coming from c can be duplicated. All the pub-sub messages that have to reach c are routed by broker a both before and after the reconfiguration takes place. It is broker a that decides, atomically, when to start serving c using the link $\langle ac \rangle$. a never routes a pub-sub message twice, so no pub-sub message can be duplicated.

Property (5). "The primitives preserve FIFO and causal ordering guarantees, if they are offered by the pub-sub system." As in Property (3) and (4), we preserve this property if it is offered using end-to-end mechanisms. On the contrary, it is possible that the pub-sub system ensures ordering among pub-sub messages by enforcing them at the level of paths. As demonstrated in [23], [14], the use of a tree-topology and a FIFO order over paths is sufficient to provide both FIFO and causal order for end-to-end pub-sub message delivery. Starting from this premise, we show that, if the pub-sub system offers FIFO order among paths, our primitive does not disrupt it.

FIFO order over paths is preserved during reconfigurations.

`SwapLink` **case.** Consider the `SwapLink` (a, b, c) primitive. In Figure 11, we consider the three brokers $a, b, c \in B$ under reconfiguration, and three external brokers $x, y, z \in B$, connected (directly or indirectly) with a, b, and c, respectively. We want to show that the paths $x \leftrightarrow y$, $y \leftrightarrow z$, and $z \leftrightarrow x$ preserve the FIFO ordering also during the reconfiguration process. This is trivially true for the path

$y \leftrightarrow z$, since it does not change during the reconfiguration (it continues to use the link $\langle bc \rangle$).



(a) Before reconfiguration (b) After reconfiguration

Figure 11: Causality is preserved during the swapLink

On the contrary, the paths $x \leftrightarrow y$ and $x \leftrightarrow z$ stop using the link $\langle ab \rangle$ and start using the link $\langle ac \rangle$. These two steps are performed sequentially, one after the other by the `SwapLink` primitive: it first adds the link $\langle ac \rangle$ and uses it to route the communication between a and c. To ensure that previous pub-sub messages flowing through b are delivered in order, this process is implemented by two `ChangePath` procedures, one realized through the `lockAck` messages and one realized through the first two `swap` messages. After that, the `SwapLink` primitive removes the link $\langle ab \rangle$, previously used for the communication between a and b. Also in this case the process is implemented through two `ChangePath` procedures, which preserve FIFO ordering of pub-sub messages during the reconfiguration.

`AddBroker` **case.** In this case there is no traffic of pub-sub messages during the reconfiguration, therefore any assumption on the order of pub-sub messages is preserved.

`RemoveBroker` **case.** In the case of removing a broker we preserve ordering by using the `ChangePath` procedure, which guarantees FIFO ordering during the reconfiguration.

Property (6). "The primitives do not introduce delays in the delivery of pub-sub messages, if we consider negligible the bandwidth used to transfer reconf messages." Let us denote Lat, $\overline{Lat}$, and Lat^* the latencies of an pub-sub message e that traverses a link connecting two reconfiguring brokers respectively before, after, and during a reconfiguration. We want to show that Lat^* has a value between Lat and $\overline{Lat}$, assuming that the link bandwidth is not affected by the traffic of reconf messages.

We first show that the property holds when the `ChangePath` procedure is applied. The `ChangePath` procedure changes the communication path between two brokers $a, b \in B$. Let us call $a \rightarrow b$ the path before the procedure takes place and $\overline{a \rightarrow b}$ the path when the procedure ends. The latency for sending a pub-sub message e over $a \rightarrow b$ is Lat, while the the latency for sending e over $\overline{a \rightarrow b}$ is $\overline{Lat}$. Broker b always processes pub-sub messages as soon as it receives them from one of the two paths. The only exception is the first pub-sub messages received from $a \rightarrow b$, which are stored until a `close` message is received from $a \rightarrow b$. The `close` message, however, is sent before a starts using the new channel $\overline{a \rightarrow b}$, and takes Lat to be received by b. Accordingly, stored pub-sub messages start to be processed at most Lat after the time they are sent by broker a. This shows that the `ChangePath` procedure does not add any overhead: the latency Lat^* is always bounded by Lat and $\overline{Lat}$. We can now show that the property holds for our primitives.

`SwapLink` **case.** The `SwapLink` primitive does not introduce any mechanism that prevents the traffic from being routed. The only exception is represented by the application of the `ChangePath` procedure, which does not introduce any overhead, as shown above. Moreover, during the reconfiguration, the traffic uses either the paths adopted before the reconfiguration, or the paths adopted after the reconfiguration. It follows that the latency Lat^*, under our assumptions, is always bounded by the latencies before and after the reconfiguration, i.e., Lat and $\overline{Lat}$.

`AddBroker` **case.** The property trivially holds since all pub-sub messages are routed over the same paths before, after, and during the reconfiguration.

`RemoveLink` **case.** Similarly to the `SwapLink` primitive, there is no mechanism that prevents the traffic from being routed. In the case of client migration, there is just the adoption of the `ChangePath` procedure, which, as shown above, does not introduce any overhead. Therefore also in this case the latency Lat^* is always bounded by the latencies before and after the reconfiguration.

Remarks on the negligible bandwidth assumption. To show that our approach does not introduce additional delay w.r.t. a non-reconfiguring system we are assuming that reconf messages consume a negligible quantity of bandwidth. We will see from the experiment in the next Section that this tends to be true also in the reality. However, depending on the routing protocol adopted and on the actual load of the system, there may be situations in which the cost for exchanging the information needed to complete the reconfiguration (e.g., portions of the `Routing Tables` of brokers) may reduce the bandwidth available for pub-sub messages. In this case the effect of a bandwidth reduction can result in an additional time for the delivery of events, and therefore an additional delay experienced by the clients. However, these modifications depend on the specific `Routing Layer` adopted, therefore it is out of the control of any reconfiguration approach (including ours). In all the cases in which brokers have only local knowledge about subscriptions (e.g., [7]), an update operation on their subscription information cannot be avoided.

VI. Evaluation

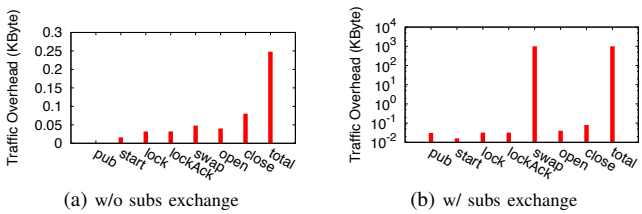
The aim of this evaluation is to measure the overhead of our primitives. Other performance metrics (e.g., scalability) depend on the specific routing protocol and self-adaptive logic adopted. We implemented and tested our primitives into the ProtoPeer simulator [11]. Since several parameters of the pub-sub system may impact the performance of our solution, we designed a default scenario and then changed one parameter at a time. Our default scenario includes 100 brokers and 100 clients. We kept the ratio of clients and brokers fixed in all the scenarios we tested. We always consider 50% of *pure forwarders*, i.e., brokers without any client attached. Clients are uniformly distributed over

the remaining brokers. The latency of links is uniformly distributed between 1ms and 5ms. Each client issues 10 different subscriptions, each of them capturing 0.1% of published events, on the average. Moreover, each client publishes one event every 750ms. We consider a fully content-based pub-sub systems, in which events are represented as attribute-value couples and subscriptions define constraints on the value of attributes. With an average size of 5 attributes per event and 5 constraints per subscription, the average size of the packets flowing in the network is 0.76KB for events and 0.77KB for subscriptions.

The values of the parameters for the default scenario have been chosen in such a way to represent a medium-scale system. We explore the impact of parameter variation w.r.t such scenario and report here the most significant findings. For each test, we perform 100 reconfigurations and we measure the average traffic overhead generated by a single reconfiguration. The reconfigurations have been generated both using a random algorithm and using our self-adaptive algorithm described in [9]. The results are identical in the two cases, which further demonstrates the transparency of our primitives w.r.t. the specific self-adaptation policy.

SwapLink. We first analyze the `SwapLink` primitive. We consider two separate cases: (*i*) every broker has a complete knowledge about all the subscriptions of the network. In this case there is no need to propagate information about subscriptions during the reconfiguration; (*ii*) every broker has only a partial (aggregated) view of subscriptions, as in the subscription forwarding protocol [7]. In this case, the reconfiguring brokers need to exchange some information about the subscriptions they hold.

Default scenario. We first measure the overhead of the `SwapLink` primitive in the default scenario. Figure 12a shows the traffic overhead when subscriptions do not need to be propagated. First of all, we notice that the total overhead introduced by a single reconfiguration is very low (below 0.25 KBytes). All the kinds of reconf messages participate with a similar cost; in particular, since `swap` messages do not have to carry information on subscriptions, their total weight is comparable to those of the other messages. Finally, the complete knowledge of subscriptions allows brokers to avoid useless forwarding of publications (`pub` in Figure 12a) during the reconfiguration phase.



(a) w/o subs exchange (b) w/ subs exchange

Figure 12: Default scenario

Figure 12b shows the traffic overhead including the propagation of subscriptions. Notice the use of the logarithmic scale to represent the large differences between the different types of messages. In particular, the `swap` messages, which include information on subscriptions, are the main sources of traffic, contributing for more than 99% of the total weight. In our default scenario, the total cost of a reconfiguration is slightly above 1MB. Notice that the amount of information exchanged within the `swap` messages depends on the specific `Routing Layer` adopted and is out of the control of our primitives. Accordingly, they would be required by every other reconfiguration approach performing the same topological change.

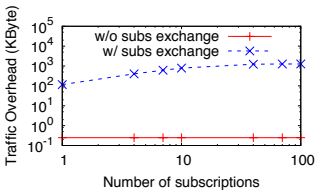


Figure 13: Number of subscriptions per client

Number of subscriptions. Figure 13 shows the behavior of the `SwapLink` primitive when the number of subscriptions per client increases. As expected, the number of subscriptions only influences the performance of the primitive when the `Routing Layer` requires to exchange information on subscriptions. In this case, the overhead increases linearly with the number of subscriptions. Interestingly, this happens up to a certain (small) threshold (about 100 subscriptions per client with our default parameters), while after this limit the overhead remains almost constant. In this case, indeed, subscriptions start to overlap with each other, so that the amount of information moved from broker to broker does not increase further.

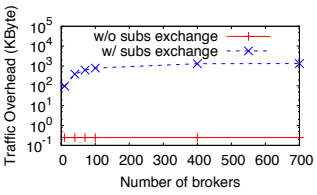


Figure 14: Number of brokers in the network

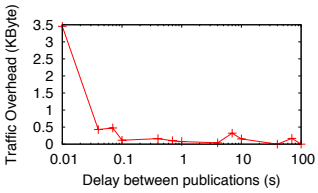


Figure 15: Period between publications

Number of brokers. Figure 14 shows the traffic overhead of the `SwapLink` primitive when increasing the number of brokers in the overlay network. When increasing the size of the network, we also increase the number of clients accordingly. As a consequence, also the overall number of subscriptions in the network increases: as Figure 14 shows, this impacts on the traffic overhead, which increases with the number of brokers and exhibits a behavior that is similar to the one already observed in Figure 13.

Delay between publications. We now consider how the traffic overhead generated by the `SwapLink` primitive changes with the average delay between two subsequent publications of a client. Without changing other parameters, the delay between publications only influences the number of additional flooded events exchanged during `State 2` and `State 3`. If the routing protocol provides each broker with a complete knowledge about the subscriptions in the network, there is sufficient information to

correctly filter events without generating additional traffic. Figure 15 shows the overhead generated by publications when the routing protocol does not provide each broker with the information required to correctly filter all the events received during the reconfiguration phase. First of all, we notice how this traffic significantly drops when the delay between publications increases. Moreover, this traffic is negligible if compared with the overall overhead, which is around 1MB with our default parameters (see Figure 12b).

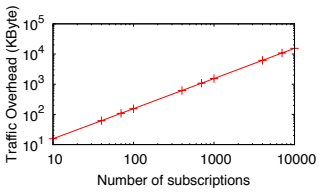


Figure 16: RemoveBroker

AddBroker and RemoveBroker. The `AddBroker` primitive only involves the creation of a new connection, and does not produce any traffic overhead in the existing network. On the contrary, the `RemoveBroker` requires a transfer of subscriptions between the removed broker and its connected neighbor. Figure 16 shows the overall traffic overhead generated by the `RemoveBroker` when changing the number of subscriptions of each client. All the remaining parameters are specified as in the default scenario. Without considering the cost of moving subscriptions, the overhead of the primitive is constant and equal to 0.11KB. As expected, the cost increases linearly with the number of subscriptions of each client, reaching more than 10MB when each client provides 10000 subscriptions. Once again, we transfer the minimum amount of information required for the routing of events, so this overhead is common to all reconfiguring approaches performing the same operation.

VII. Conclusions

This paper presents three generic and transparent overlay reconfiguration primitives for pub-sub systems that enable modifications in the overlay topology that are transparent to clients. These primitives can be used to realize the execution phase of a self-adaptation loop in a self-organizing middleware that is independent (*i*) from the monitoring/analysis/plan phases of the self-adaptation logic (*ii*) from the routing protocol adopted by the system.

In the future, we plan to extend these primitives to a wider class of self-adaptive pub-sub systems such as the ones that use unstructured topologies or the ones that require a stronger fault-tolerance pub-sub middleware.

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

[1] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to siena. *Comput. J.*, 50(4), 2007.

[2] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. *Middleware for Network Eccentric and Mobile Applications. State of the Art.*, 2008.

[3] S. Bhola. Topology changes in a reliable publish/subscribe system. Technical report, IBM T.J. Watson, 2004.

[4] A. Carzaniga, A. J. Rembert, and A. L. Wolf. Understanding content-based routing schemes. Technical Report 2006/05, University of Lugano, 2006.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *PODC '00*. ACM, 2000.

[6] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *INFOCOM '04*, 2004.

[7] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *TSE*, 27, 2001.

[8] G. Cugola and G. Picco. REDS: A Reconfigurable Dispatching System. In *SEM '06*. ACM Press, 2006.

[9] D. J. Dubois. *Self-organizing Methods and Models for Software Development*. PhD in Information Engineering, Politecnico di Milano, Milan, Italy, 2011.

[10] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35, 2003.

[11] W. Galuba, K. Aberer, Z. Despotovic, and W. Kellerer. Protopeer: a p2p toolkit bridging the gap between simulation and live deployement. In *Simutools '09*. ICST, 2009.

[12] S. Guo, K. Karenos, M. Kim, H. Lei, and J. Reason. Delay-cognizant reliable delivery for publish/subscribe overlay networks. In *ICDCS '11*. IEEE CS, 2011.

[13] S. Hu, V. Muthusamy, G. Li, and H.-A. Jacobsen. Transactional mobility in distributed content-based publish/subscribe systems. In *ICDCS '09*. IEEE CS, 2009.

[14] X. Jia. A total ordering multicast protocol using propagation trees. *TPDS*, 6(6), 1995.

[15] M. Kim, K. Karenos, F. Ye, J. Reason, H. Lei, and K. Shagin. Efficacy of techniques for responsiveness in a wide-area publish/subscribe system. In *Middleware '10*. ACM, 2010.

[16] I. Loulou, M. Jmaiel, K. Drira, and A. H. Kacem. P/s-com: Building correct by design publish/subscribe architectural styles with safe reconfiguration. *J. Syst. Softw.*, 83, 2010.

[17] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001.

[18] M. Migliavacca and G. Cugola. Adapting publish-subscribe routing to traffic demands. In *DEBS '07*. ACM, 2007.

[19] G. Mühl, L. Fiege, F. Gartner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *MASCOTS '02*, 2002.

[20] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[21] H. Parzyjegla, G. Gero Muhl, and M. A. Jaeger. Reconfiguring publish/subscribe overlay topologies. In *ICDCSW '06*. IEEE CS, 2006.

[22] G. P. Picco, G. Cugola, and A. L. Murphy. Efficient content-based event dispatching in the presence of topological reconfiguration. In *ICDCS '03*. IEEE CS, 2003.

[23] L. Xie, X. Jia, and K. Zhou. Causal ordering group communication for cognitive radio ad hoc networks. In *INFOCOM Workshops*, 2011.

[24] Y. Yoon, V. Muthusamy, and H. A. Jacobsen. Foundations for highly available content-based publish/subscribe overlays. In *ICDCS '11*. IEEE CS, 2011.