REDS: A Reconfigurable Dispatching System

Gianpaolo Cugola[1]
[1]Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
cugola@elet.polimi.it

Gian Pietro Picco[1,2]
[2]Dept. of Informatica e Telecomunicazioni
Università di Trento, Italy
picco@dit.unitn.it

ABSTRACT

We present a new publish-subscribe middleware called REDS (REconfigurable Dispatching System) designed to tolerate *dynamic reconfigurations* of the dispatching infrastructure, like those occurring in scenarios characterized by fluid topologies as in mobile and peer-to-peer networks. We illustrate the modular architecture of REDS, which enables programmers to change the internal configuration of the middleware to suit the deployment scenario, focusing on the aspects concerned with the dynamic reconfiguration of the dispatching network.

Keywords. Publish-subscribe, content-based routing, dynamic reconfiguration, mobile and peer-to-peer computing.

1. INTRODUCTION

The publish-subscribe model of interaction has recently emerged as a promising approach to tackle the requirements of modern distributed applications in terms of flexibility and decoupling among components. Several publish-subscribe middleware exist, developed both by industry and academia. In most cases, their focus is on scalability (e.g., in distributing the dispatching infrastructure and implementing efficient algorithms for matching messages against subscriptions) and are designed with fairly stable networks in mind. Consequently, they do not provide any explicit mechanism to rearrange their dispatching infrastructure in response to changes in the network topology, as those inevitably occurring in many scenarios including peer-to-peer and mobile networking. This situation is unfortunate, since it hampers the use of publish-subscribe precisely in those scenarios where its flexibility provides most of its benefits.

In the last few years, we worked with colleagues to remove this limitation by developing several protocols dealing with various aspects of reconfiguration, with the goal of making publish-subscribe suitable for the aforementioned scenarios [13]. In experimenting with these protocols we felt the need for a *framework* enabling us to sharply decouple reconfiguration issues from the other aspects characterizing a publish-subscribe system, to simplify the evaluation of our solutions without the need to rebuild the whole system.

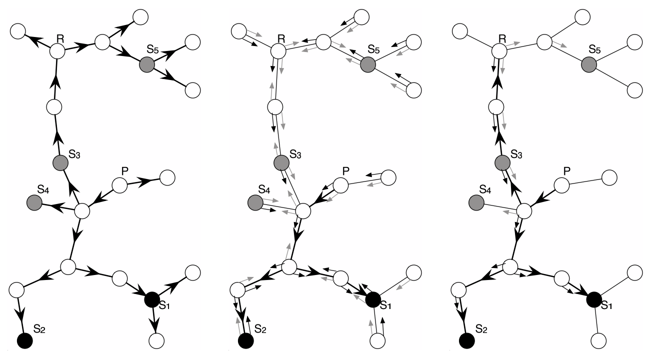
In this paper we present REDS (REconfigurable Dispatching System), the publish-subscribe middleware we developed to satisfy these needs. It provides a modular architecture whose components can be easily changed to adapt to different deployment scenarios. In particular, the innards of REDS are easily accessible, enabling one to select (or integrate) the most appropriate protocols to deal with topological reconfiguration. Moreover, our system is customizable also along dimensions of publish-subscribe not immediately related with reconfiguration (e.g., the routing strategy or the format of messages and filters). REDS is publicly available as open source at `zeus.elet.polimi.it/reds`.

Next, we introduce publish-subscribe middleware and motivate the need for a new system. In Section 3 we present the application programming interface (API) that REDS provides to developers of distributed applications, describe its internal architecture, and discuss the mechanisms offered for customization and extension. Section 4 illustrates the versatility of our approach by using the built-in components included in the REDS distribution. Section 5 compares REDS with related work. Finally, Section 6 ends the paper with brief concluding remarks.

2. WHY A NEW SYSTEM?

Distributed applications exploiting publish-subscribe middleware are organized as a collection of peers (hereafter called *clients*) that interact by *publishing* messages and by *subscribing* to the classes of messages they are interested in [21]. The core component of the middleware, the *dispatcher*, is responsible for collecting subscriptions and forwarding messages from publishers to subscribers. In doing so it achieves a high degree of decoupling among the clients. In principle, it is possible to add or remove one of them without affecting the others—only the dispatcher needs to be aware of the change. Clearly, this form of decoupling is particularly desirable in scenarios where the set of clients undergoes continuous change as in the peer-to-peer and mobile ones.

Publish-Subscribe on Dynamic Topologies. Unfortunately, much of the potential of the publish-subscribe *model* still remains to be unleashed by most of currently available



(a) Message forwarding (b) Subscription forwarding (c) Hierarchical forwarding

Figure 1: Tree-based routing strategies. Broker S_1 and S_2 subscribed (through their clients, not shown in the figure) to the same "black" filter, while S_3 subscribed to "gray". Arrows represent the content of subscription tables for the filters of the same color. Broker P published a message matching the black filter but not the gray one. The path followed by this message is shown by thick arrows. Broker R is the root for hierarchical forwarding.

publish-subscribe *systems*. Indeed, most of the research in the field focuses on scalability and realizes the dispatcher as a distributed set of *brokers* interconnected in an overlay network, cooperatively routing the messages and subscriptions issued by the clients connected to them. In this context, the main design decisions concern the topology of interconnection and the routing strategy. As an example, Figure 1 illustrates the simplest and most common tree-based routing strategies found in the literature [4,6,11]. These distributed solutions, however, are meant to be deployed in broker networks that are essentially stable: their use in the dynamic topologies characteristics of the aforementioned scenarios is inefficient or even impossible. Therefore, paradoxically, many systems are unusable precisely in the deployment scenarios where the decoupling fostered by the publish-subscribe model is clearly an asset.

Publish-subscribe over dynamic topologies is an open research issue, where our group has made a number of contributions (e.g., [9,12,14,23]). However, no established and general solution has yet appeared, in part because the tradeoffs strongly depend on the assumptions made on the deployment scenario (e.g., in terms of the frequency of topological changes or the nature of the underlying network). In this situation, we quickly faced the need to experiment with different solutions for different scenarios, without having to restart from scratch every time, and with the ability to easily factor out the portion common to multiple approaches.

REDS fulfills this goal with a modular architecture that encapsulates the basic mechanisms to support dynamic reconfiguration of the dispatching infrastructure (i.e., to change the interconnection topology of brokers and to reconcile subscription tables) into a small set of components. These interact with the rest of the system through well-defined interfaces, and can be easily changed at deployment time.

Publish-Subscribe *à la carte*. Once we made the leap towards a high degree of modularization, we easily recognized that other dimensions, not necessarily related with dynamic topologies, could be left open-ended and easily customizable according to the application needs.

Indeed, existing publish-subscribe systems interpret the publish-subscribe paradigm in many different ways [15,21]. A first point of differentiation are the very notions of message and subscription. Depending on the system at hand, messages can be bare strings, untyped sequences of values (tuples), untyped name-value pairs, XML code, typed C-like structures, and even full-fledged typed objects, complete of attributes and methods. In turn, the message format constrains the subscription language. For instance, regular expressions are the typical subscription language associated to string messages. Also, the expressive power of the subscription language impacts the inner working of the middleware. For instance, as described in [6], under some conditions it is possible to detect that a filter is subsumed by an already existing one, or even to merge two filters into a more general one, thus reducing the matching space. The actual mechanisms used in the implementation of a publish-subscribe system exhibit an even wider variety. We already touched upon the routing strategies enabling a distributed dispatcher. Moreover, several solutions (e.g., [1,7,16]) exist to efficiently match messages against subscriptions and determine the intended recipients. This process usually relies on *subscription tables*: dedicated data structures that hold information about the subscriptions (e.g., filter and source). Their layout, often designed along with the subscription language, is key to the alternative efficient matching and forwarding strategies found in the literature.

REDS encompasses this space of alternatives by encapsulating each concern in a separate module. Thus, the format of message and filters, the mechanisms for managing subscription tables and the related matching, the routing strategy, the underlying network transport protocol, and the management of the dispatching overlay network are all separate components in REDS. In some cases (e.g., routing strategy, network transport, overlay network maintenance) it is possible to replace one of these components without affecting the rest of the architecture. In the cases where specific design choices or the very semantics of publish-subscribe prevents this desirable orthogonality (e.g., when designing a subscription table that relies on a specific subscription language), the REDS architecture provides a reference for understanding the relationship among components and the extent of their interactions.

Target users. We believe that the open framework provided by REDS can be useful to both developers and researchers interested in publish-subscribe systems. To the former, REDS provides flexibility to deploy the components most suited for a given application scenario, as well as the basic building blocks for customizing such components when faced with new application challenges. To the latter, REDS provides the ability to experiment with and evaluate novel algorithms and mechanisms, without the need to reinvent the wheel each time.

3. ARCHITECTURE

REDS is a framework (in the object-oriented sense) of Java interfaces and classes, which define:

1. a client API, enabling access to the publish-subscribe services;
2. a broker API, enabling access to the components inside the broker.

Moreover, the REDS distribution provides several ready-to-use implementations of the various components.

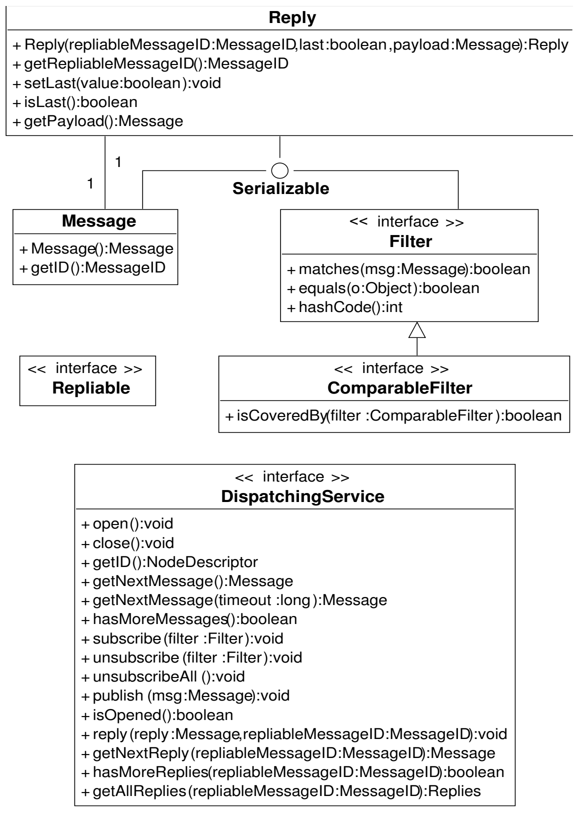
In the rest of this section, we describe the client (Section 3.1) and broker (Section 3.2) APIs, alongside with a brief discussion of available components. Their actual use is exemplified in Section 4.

3.1 Client API

Application components access the services provided by the REDS distributed dispatcher through the `DispatchingService` interface, shown in Figure 2. The current release provides two implementations that differ in the transport protocol used to connect with brokers (UDP and TCP) and ultimately hide all the details about how clients are connected and communicate through the network.

Moreover, REDS is independent from the specific format of messages and filters. This empowers developers with great flexibility: custom messages and filters with very different semantics can be easily developed (e.g., using XML messages and XPath filters) without affecting the dispatching infrastructure. To achieve this goal, REDS defines one abstract class for messages and two interfaces for filters, which include the minimal set of methods required by a publish-subscribe broker to operate. The `Message` abstract class embeds the identifier of the message, computed at publish time. Instead, the `Filter` interface defines a method `matches` to be redefined by the implementing class with the desired filtering logic. In addition, the methods `hashcode` and `equals` are used internally by REDS brokers to store and compare filters, e.g., to avoid propagating them multiple times along the same link. Finally, the `ComparableFilter` interface represents filters that can be "covered" by others according to some notion of partial ordering. As in the case of matching, the method `isCoveredBy` encodes the coverage semantics and is used by brokers to limit the propagation of subscriptions, e.g., as described in [6].

REDS also supports a paradigm where clients are enabled to send *replies* to messages, thus naturally providing bidirectional communication within the framework of content-based publish-subscribe. Brokers keep track of the transit of messages tagged as `Repliable` and store routes followed back by the reply messages, issued using the `reply` method of `DispatchingService`. By supporting replies natively, REDS brokers are able to track the number of expected replies for each message and check if and when all of them have been received—a feature that cannot be obtained by implementing replies at the application level.



Reply
+ Reply(repliableMessageID:MessageID,last:boolean,payload:Message):Reply
+ getRepliableMessageID():MessageID
+ setLast(value:boolean):void
+ isLast():boolean
+ getPayload():Message
1
1
Serializable
Message
+ Message():Message
+ getID():MessageID
<< interface >>
Filter
+ matches(msg:Message):boolean
+ equals(o:Object):boolean
+ hashCode():int
<< interface >>
Repliable
<< interface >>
ComparableFilter
+ isCoveredBy(filter :ComparableFilter):boolean
<< interface >>
DispatchingService
+ open():void
+ close():void
+ getID():NodeDescriptor
+ getNextMessage():Message
+ getNextMessage(timeout :long):Message
+ hasMoreMessages():boolean
+ subscribe(filter :Filter):void
+ unsubscribe(filter :Filter):void
+ unsubscribeAll():void
+ publish(msg:Message):void
+ isOpened():boolean
+ reply(reply:Message,repliableMessageID:MessageID):void
+ getNextReply(repliableMessageID:MessageID):Message
+ hasMoreReplies(repliableMessageID:MessageID):boolean
+ getAllReplies(repliableMessageID:MessageID):Replies

Figure 2: The client API.

3.2 Broker API

While the previous section focused on the API offered to the application programmers, here we focus on the internal architecture of the broker and consequently on the API offered to system integrators and middleware programmers.

Similarly to most publish-subscribe systems, REDS adopts a distributed architecture for its message dispatcher, which is organized as a set of brokers connected in an overlay dispatching network. This choice is usually motivated by its enhanced scalability, and REDS benefits from it as well. However, we chose to adopt such approach for a different reason: the fact that the dynamic scenarios we target inherently require the adoption of a distributed dispatcher. Actually, in most cases these scenarios even require a broker per host, as it is not possible to rely on permanently available brokers in, say, mobile ad hoc networks (MANETs) or fully decentralized peer-to-peer networks.

To support dynamic reconfiguration of the dispatching network, REDS brokers are structured in two layers: *overlay* and *routing*. The two are independent, i.e., the particular choice of components building one of them generally does not influence the other.

Overlay. The overlay layer (see Figure 3) is in charge of managing the topology of the overlay dispatching network. It offers services to build the overlay network and rearrange it based on input from upper layers (e.g., for load balanc-

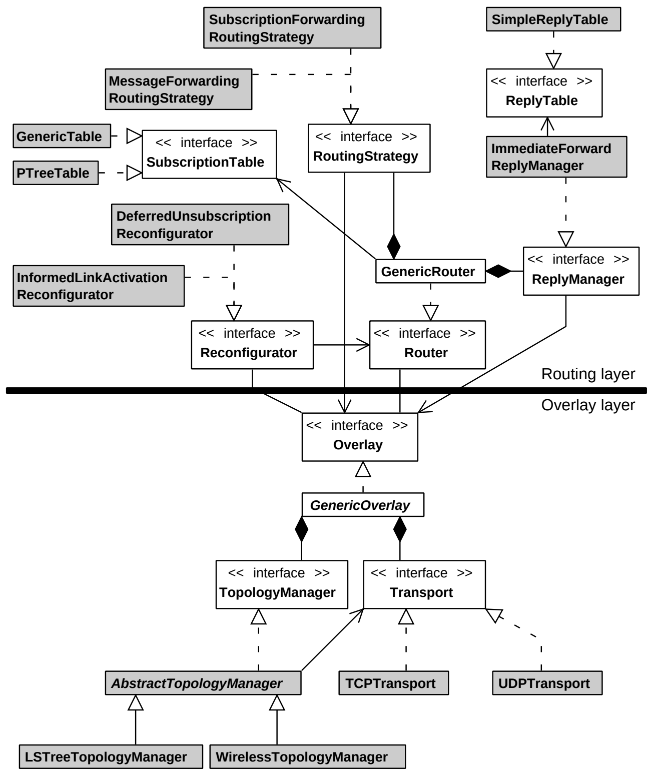

SubscriptionForwarding RoutingStrategy
MessageForwarding RoutingStrategy
SimpleReplyTable
<< interface >> ReplyTable
GenericTable
PTreeTable
<< interface >> SubscriptionTable
<< interface >> RoutingStrategy
ImmediateForward ReplyManager
DeferredUnsubscription Reconfigurator
InformedLinkActivation Reconfigurator
GenericRouter
<< interface >> ReplyManager
<< interface >> Reconfigurator
<< interface >> Router
Routing layer
Overlay layer
<< interface >> Overlay
GenericOverlay
<< interface >> TopologyManager
<< interface >> Transport
AbstractTopologyManager
TCPTransport
UDPTransport
LSTreeTopologyManager
WirelessTopologyManager

Figure 3: The broker API. The dark components are those available in the current REDS distribution.

ing). Moreover, and most crucial for our goals, it embeds the protocols that maintain the overlay network connected when the topology of the underlying physical network changes autonomously (e.g., because some peers leave the peer-to-peer network, or as a consequence of mobility).

The `Overlay` interface defines the aforementioned services. Although in principle a single component could offer all the services specified by this interface, our development greatly benefited from a design that decouples the details of how data is transported among brokers from those concerned with building and maintaining the overlay network itself. This results in a cleaner design, enabling one to change the two aspects separately. Therefore, the REDS distribution provides a `GenericOverlay` class, which implements the `Overlay` interface by delegating its services to the `Transport` and `TopologyManager` components.

The `Transport` provides methods to open and close a link toward another broker, and to send and receive data among directly connected brokers. To provide an entry point for the `TopologyManager`, the `Transport` offers methods to register listener components to be notified when new links open or existing links close (or break). The current release includes two implementations of the `Transport` interface, using TCP and UDP. In both cases, messages are encoded using the standard Java serialization.

The `TopologyManager` implements the protocol that maintains the overlay connected, and does so by guaranteeing that the topology constraints required by the chosen routing protocol are satisfied (e.g., that no loops exists in a tree overlay). This component provides methods called by the routing layer to explicitly manipulate the overlay (e.g., by adding or removing brokers), and to access the list of neighboring brokers. It also implements the methods offered by the `Overlay` interface to register the listeners to be notified when the neighbor set changes. These methods are useful to enable components in the routing layer (e.g., the `Reconfigurator` we describe next) to react to changes in the overlay topology.

As shown in Figure 3, the current release includes two topology managers, sharing a common set of functionalities implemented into the `AbstractTopologyManager` class. Both build and maintain an overlay organized as an unrooted tree of brokers. The `LSTreeTopologyManager` targets peer-to-peer, wired scenarios, and implements the protocol described in [17]. It assumes the availability of a low-level point-to-point routing protocol (i.e., IP) and provides mechanisms enabling brokers to freely join and leave the network (including crashes) guaranteeing that the overlay network remains connected and operational. A radically different approach is adopted by the `WirelessTopologyManager`, which implements the protocol described in [20] to guarantee connectivity among brokers in a MANET. Given the peculiarity of the scenario, it does not rely on any lower-level network protocol and assumes only that a local broadcast facility is available—a fundamental and always satisfied assumption in MANETs.

Routing. As shown in Figure 3, the central component of the routing layer is the `Router`, which implements the main routing process by registering with the `Overlay` component to be notified when subscriptions, messages, and replies arrive from neighbors (i.e., other brokers or clients). As with `Overlay`, although in principle a single component could implement the entire routing process, our experience suggests that it is better decoupled into independent concerns. Therefore, REDS provides a `GenericRouter` class, which implements the `Router` interface and delegates functionality to other components:

- `SubscriptionTable` is in charge of efficiently storing subscriptions coming from neighbors. When a new message arrives, the `SubscriptionTable` is responsible for matching it against stored subscriptions to determine subscribed neighbors, i.e., it implements the "matching strategy" [21]. As shown in Figure 3, the current release includes a very simple subscription table, `GenericTable`, which maintains a list of all the filters received for each neighbor and matches incoming messages by repeatedly invoking the `Filter.match` method. Although not very efficient, this approach is simple and does not make any assumption about the format of messages and filters. A radically different approach is taken by the `PTreeTable`, which implements the more efficient algorithm described in [1]. However, this component requires specialized messages and filters, with the former organized as a set of attributes (i.e., key-value pairs) and the latter as a conjunction of predicates on message attributes.

- As its name suggests, `RoutingStrategy` implements the specific routing strategy adopted, which ultimately determines how subscriptions and messages are routed along the dispatching network. REDS currently includes the two most commonly used in publish-subscribe systems: subscription forwarding and message forwarding [6].
- `ReplyManager` is in charge of routing replies towards their corresponding publisher, based on the information it stores in the `ReplyTable`. REDS currently provides only an `ImmediateForwardReplyManager` that, as its name suggests, forwards replies as soon as they arrive[1].

The `Router` component entirely governs the "standard" routing behavior on a static topology. However, it is the `Reconfigurator` component that holds the critical role of guaranteeing that routing information, and in particular the content of subscription tables, remains consistent when the topology of the dispatching network changes. This is achieved by embedding appropriate protocols to reconcile such information. In a sense, the `Reconfigurator` complements the `TopologyManager`: the latter guarantees connectivity among brokers without caring about publish-subscribe routing, which is instead responsibility of the former. The current release provides two reconfiguration strategies, both extending subscription forwarding with reconfiguration capabilities. The `DeferredUnsubscriptionReconfigurator` component implements the protocol we described in [23] and, as the name suggests, delays the unsubscriptions necessary to account for a broker leaving the network. This way, the `TopologyManager` has the time to restore connectivity and allow new subscriptions to spread in the network before the unsubscription process starts, therefore reducing the overhead, as shown quantitatively in [23]. Moreover, an `InformedLinkActivationReconfigurator` component is also provided, which bring significant additional overhead reduction at the expense of more complicate processing and more stringent assumptions.

4. REDS IN PRACTICE

A thorough illustration of how REDS addresses scenarios characterized by topological reconfiguration of the dispatching network is outside the scope of this paper. In fact, a fundamental constituent of this capability are the distributed protocols that enable REDS brokers to coordinate and cooperate to change the topology of the overlay network in response to topological changes occurring at the networking level, e.g., hosts moving in a wireless network or joining and leaving in a peer-to-peer one. However, the characteristics of the various alternatives we provide for overlay or routing are described in full detail in other papers (see [13] for an overview). Here, instead, we want to present the reader with a practical illustration of the benefits of the decoupling of reconfiguration achieved by the REDS architecture—the focus of this paper—and in particular that:

1. the ability to assemble the broker's inner components enables simple customization of the *infrastructure*, by leaving the *application* unmodified;
2. the effort to implement such a broker customization is minimal.

In publish-subscribe systems, application development consists of implementing the client code exploiting the publish-subscribe operations. This is the case also for REDS, where the `DispatchingService` provides applications with access to the dispatching infrastructure. Nevertheless, while in other systems the dispatching infrastructure is usually a run-time component outside the control of the programmer, REDS provides the ability to adapt it to the application scenario at hand. This task is performed by the system integrator (in general different from the application programmer) and may consist of "assembling" a broker based on the elementary components provided by REDS, and/or developing new ones. Interestingly, dealing with reconfiguration in REDS is relegated entirely to the customization and deployment of the dispatching network and does not affect at all the client code. This decoupling is a highly desirable property, as it allows one to run the *same* application code (that can be developed by a third party, and therefore unchangeable) on different dispatching networks and deployment scenarios.

Decoupling makes broker customization a straightforward chore. Of all the components involved in the REDS architecture, only two need to change to address reconfiguration—`TopologyManager` and `Reconfigurator`—and, as illustrated in Section 3.2, each is largely independent from the other. Therefore, the provision of custom components that redefine the built-in strategies is simplified as the aspects dealing with reconfiguration are highly decoupled and independent from each other and w.r.t. the system at large: the system integrator performing the customization needs to care about only one concern at a time.

Instead, if the system integrator decides to use the components already available in REDS, their selection is trivial, as shown in Figure 4. On the left-hand side[2] is a broker definition targeted at a large-scale, wired, peer-to-peer scenario. Here, we selected the `LSTreeTopologyManager` overlay, optimized for this setting, and used an `InformedLinkActivationReconfigurator` to govern the reconciliation of routing information upon a topological change. On the right-hand side is instead a broker using our overlay for MANETs, along with a `DeferredUnsubscriptionReconfigurator`. Incidentally, note how we also easily configure the transport layer to be TCP in the wired setting and UDP in the mobile one. As the reader can appreciate by visually comparing the code, a change in a few lines of the broker's code bears a big impact in the broker's behavior, making it suited for scenarios that have radically different characteristics. In this case, not only we switch from a socket-based `TopologyManager` to a broadcast-based one, but also use reconfiguration strategies that provide radically different tradeoffs between performance and applicability. To provide the reader with

[1]Other strategies could be used, like aggregating replies at each hop. We are currently investigating the tradeoffs of several alternatives.

[2]The figure shows the full code for the sake of argument. In practice, the selection can be performed through configuration files.

```java
public class WiredBroker {
  ...
  Transport trans;
  TopologyManager topolMgr;
  Overlay overlay;
  Router router;
  RoutingStrategy rs;
  NeighborManager neighMrg;
  SubscriptionTable subTbl;
  ReplyManager replyMgr;
  ReplyTable replyTbl;
  // create the overlay layer
  trans   =new TCPTransport(port);
  topolMgr=new LSTreeTopologyManager(trans);
  overlay =new GenericOverlay(topolMgr, trans);
  // create the routing layer
  router  =new GenericRouter(overlay);
  rs      =new SubscriptionForwardingRoutingStrategy();
  neighMrg=new InformedLinkActivationReconfigurator();
  subTbl  =new GenericTable();
  // provide support for replies
  replyMgr=new ImmediateForwardReplyManager();
  replyTbl=new HashReplyTable();
  // bind routing components to each other
  router.setSubscriptionTable(subTbl);
  router.setRoutingStrategy(rs);
  router.setNeighborManager(neighMrg);
  router.setReplyManager(replyMgr);
  router.setReplyTable(replyTbl);
  replyMgr.setReplyTable(replyTbl);
  // bind routing and overlay components
  rs.setOverlay(overlay);
  neighMrg.setOverlay(overlay);
  replyMgr.setOverlay(overlay);
  // start the overlay
  overlay.start();
}
```

```java
public class WirelessBroker {
  ...
  Transport trans;
  TopologyManager topolMgr;
  Overlay overlay;
  Router router;
  RoutingStrategy rs;
  NeighborManager neighMrg;
  SubscriptionTable subTbl;
  ReplyManager replyMgr;
  ReplyTable replyTbl;
  // create the overlay layer
  trans   =new UDPTransport(localPort);
  topolMgr=new WirelessTopologyManager(trans, port);
  overlay =new GenericOverlay(topolMgr, trans);
  // create the routing layer
  router  =new GenericRouter(overlay);
  rs      =new SubscriptionForwardingRoutingStrategy();
  neighMrg=new DeferredUnsubscriptionReconfigurator();
  subTbl  =new GenericTable();
  // provide support for replies
  replyMgr=new ImmediateForwardReplyManager();
  replyTbl=new HashReplyTable();
  // bind routing components to each other
  router.setSubscriptionTable(subTbl);
  router.setRoutingStrategy(rs);
  router.setNeighborManager(neighMrg);
  router.setReplyManager(replyMgr);
  router.setReplyTable(replyTbl);
  replyMgr.setReplyTable(replyTbl);
  // bind routing and overlay components
  rs.setOverlay(overlay);
  neighMrg.setOverlay(overlay);
  replyMgr.setOverlay(overlay);
  // start the overlay
  overlay.start();
}
```

Figure 4: Two different broker incarnations: For large-scale, wired, peer-to-peer networks (left) and small-scale, wireless mobile ad hoc networks (right).

a feel about the entity of changes, from results we derived recently the `Reconfigurator` we use in the wired broker appears to generate 50% less overhead than the one used in the wireless case. Nevertheless, the former is hard to apply in a wireless scenario as it uses out-of-band messages, which would need a separate unicast channel not provided by our wireless overlay. This shows that there is no one-size-fits-all approach, and that our architecture is effective in enabling big configuration changes with minimal programming effort.

5. RELATED WORK

After the first experiences focusing on centralized, subject-based systems, recent years witnessed the development of many distributed, content-based, publish-subscribe middleware. They differ in several aspects [15,21] but most share two characteristics: *i)* they do not explicitly take reconfiguration of the dispatching network into consideration and *ii)* most of their features are hard-wired and unchangeable. REDS is innovative in both aspects: dynamic reconfiguration of the dispatching topology is the main motivating requirement, and modularity enables the selection and integration of several aspects into a coherent and yet highly decoupled design.

Some approaches (e.g., [5,11]) solve a different and more limited problem, that is, enabling clients to freely roam from a broker to another. Reconfiguration of the dispatching network is instead addressed by Joram [3], a distributed implementation of the JMS API [25] that provides facilities to add and remove brokers dynamically and to transparently handle network handover and broker failover. Similar features are provided by Hermes [24], a publish-subscribe middleware built on top of a peer-to-peer overlay network organized as a distributed hashtable. With respect to both systems REDS presents two advantages: *i)* it provides mechanisms tailored to mobile wireless networks and in general highly dynamic systems, while the two aforementioned systems are designed to operate in wired networks with low degrees of reconfiguration; *ii)* the mechanisms supporting reconfiguration in REDS are not hard-wired in the broker's code and can be easily changed to fulfill different application requirements.

Recently, several works presented possible solutions to the problem of content-based routing on MANETs. Among them, those closer to the approach implemented in REDS through the `WirelessTopologyManager` are [18,26,27], which try to achieve efficiency by routing messages through an overlay network of brokers, built and maintained by relying on single-hop communication facilities. A different approach is taken by JEcho [8], which assumes the availability of a multi-hop unicast protocol to mask the topology changes induced by mobility. This simplifies the structure of the publish-subscribe system, which can be designed as if operating on a stable network, but may easily result in an overlay topology that rapidly diverges from the physical one. Therefore, JEcho brokers periodically run a link state protocol to build a global view of the physical network and rearrange the overlay accordingly. The open architecture of

REDS can easily encompass all these approaches by developing appropriate TopologyManager components. Again, this is a consequence of the fact that the mechanisms supporting reconfiguration are not hard-wired in the code of REDS brokers.

A different strategy to content-based routing on MANETs is adopted by those systems, like [2, 10, 19, 22], which support highly dynamic scenarios, by forsaking the overlay-based approach. In these systems, routing is performed by using soft-state information about neighbors and, in some cases, by also adopting probabilistic techniques when such information is unavailable or outdated. While we have not direct experience in trying to introduce such innovative form of routing into REDS, it is our belief that they could fit the REDS framework, e.g., by introducing an ad-hoc Overlay component and by redefining the way the Router works.

6. CONCLUSIONS

In this paper we presented REDS, a novel publish-subscribe middleware expressly designed to support topological reconfiguration of the dispatching network. As such, REDS enables publish-subscribe in dynamic environments as peer-to-peer networks and MANETs. Moreover, its highly modular design sharply decouples the management of reconfiguration from the other issues, and in general empowers developers with a high degree of flexibility.

Clearly, we cannot claim that our design works for any algorithm and design aspect of publish-subscribe—one can always conceive some novel, unanticipated feature that holds the potential to break some of the design assumptions in REDS. Nevertheless, REDS is an ongoing project where the shape of the framework is undergoing continuous validation as we implement more modules and progressively extend the range of features we are incorporating in the system. Thus far, our design has proven successful in accommodating our reconfiguration algorithms, as well as many other aspects of publish-subscribe systems.

Acknowledgements. REDS is part of a larger research collaboration with Paolo Costa, Davide Frey, Matteo Migliavacca, Luca Mottola, and Amy L. Murphy. Moreover, REDS developers include Fabrizio Buzzi, Sandro Calzoni, Kubilai Hernandez, and Alessandro Monguzzi. The authors are deeply indebted with all of the aforementioned people.

This work was partially supported by the Italian Ministry of Education, University, and Research (MIUR) under the VICOM project, by the National Research Council (CNR) under the IS-MANET project, and by the European Community under the IST-004536 RUNES project.

7. REFERENCES

[1] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching Events in a Content-Based Subscription System. In *Proc. of the 18th ACM Symp. on Principles of Distributed Computing*, pages 53–61. ACM, 1999.

[2] R. Baldoni, R. Beraldi, G. Cugola, M. Migliavacca, and L. Querzoni. Structure-less content-based routing in mobile ad hoc network s. In *Proc. of the IEEE Int. Conf. on Pervasive Services*. IEEE Computer Society, July 2005.

[3] R. Balter. Joram: The open source enterprise service bus. Technical report, ScalAgent Distributed Technologies, March 2004. www.scalagent.com/pages/en/datasheet/040322-joram-whitepaper-en.pdf.

[4] G. Banavar et al. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In *Proc. of the 19th Int. Conf. on Distributed Computing Systems*, 1999.

[5] M. Caporuscio, A. Carzaniga, and A. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Trans. on Software Engineering*, 29(12):1059–1071, Dec. 2003.

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

[7] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Proc. of ACM SIGCOMM 2003*, pages 163–174, Karlsruhe, Germany, Aug. 2003.

[8] Y. Chen and K. Schwan. Opportunistic overlays: Efficient content delivery in mobile ad hoc networks. In G. Alonso, editor, *Proc. of the 6th ACM/IFIP/USENIX Int. Middleware Conf.*, LNCS 3790, pages 354–374, Grenoble, France, November 2005. Springer.

[9] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation. In *Proc. of the 24th Int. Conf. on Distributed Computing Systems*, pages 552–561. IEEE Computer Society, Mar. 2004.

[10] P. Costa and G. Picco. Semi-probabilistic Content-based Publish-Subscribe. In *Proc. of the 25th Int. Conf. on Distributed Computing Systems*. IEEE Computer Society, June 2005.

[11] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. on Software Engineering*, 27(9):827–850, Sept. 2001.

[12] G. Cugola, D. Frey, A. Murphy, and G. Picco. Minimizing the Reconfiguration Overhead in Content-Based Publish-Subscribe. In *Proc. of the ACM Symp. on Applied Computing (SAC) 2004*, pages 1134–1140. ACM, 2004.

[13] G. Cugola, A. Murphy, and G. Picco. Content-based Publish-subscribe in a Mobile Environment. In P. Bellavista and A. Corradi, editors, *Mobile Middleware*. CRC, 2005. Invited contribution. To appear. www.elet.polimi.it/upload/picco.

[14] G. Cugola, G. Picco, and A. Murphy. Towards Dynamic Reconfiguration of Distributed Publish-Subscribe Systems. In *Proc. of the 3rd Int.*

Workshop on Software Engineering and Middleware (SEM'02), volume LNCS 2596, pages 187–202. Springer, May 2002.

[15] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[16] F. Fabret et al. Filtering algorithms and implementation for very fast publish/subscribe systems. *ACM SIGMOD Record*, 30(2):115–126, 2001.

[17] D. Frey and A. Murphy. Maintaining publish-subscribe overlay tree in large scale dynamic networks. Technical report, Politecnico di Milano, 2005. Submitted for publication. `www.elet.polimi.it/upload/frey`.

[18] Y. Huang and H. Garcia-Molina. Publish/subscribe tree construction in wireless ad-hoc networks. In *Proc. of the 4th Int. Conf. on Mobile Data Management (MDM03)*. Springer, 2003.

[19] R. Meier and V. Cahill. Steam: Event-based middleware for wireless ad hoc networks. In *Proc. of the Int. Workshop On Distributed Event-Based Systems*, Vienna, Austria, 2002.

[20] L. Mottola, G. Cugola, and G. Picco. A Self-Repairing Tree Overlay Enabling Content-based Routing in Mobile Ad Hoc Networks. Technical report, Politecnico di Milano, 2006. Submitted for publication. `www.elet.polimi.it/upload/picco`.

[21] G. M uhl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[22] M. Petrovic, V. Muthusamy, and H.-A. Jacobsen. Content-based routing in mobile ad hoc networks. In *Proc. of the 2nd Annual Int. Conf. on Mobile and Ubiquitous Systems: Networking and Services*, pages 45—55, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

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

[24] P. Pietzuch and J. Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In *Proc. of the 2nd Int. Workshop on Distributed Event-Based Systems*, San Diego, CA, June 2003.

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

[26] E. Yoneki and J. Bacon. An adaptive approach to content-based subscription in mobile ad hoc networks. In *Proc. of the 2nd IEEE Annual Conference on Pervasive Computing and Communications Workshops (PERCOMW04)*, 2004.

[27] E. Yoneki and J. Bacon. Content-based routing with on-demand multicast. In *Proc. of the 24th Int. Conf. on Distributed Computing Systems Workshops (ICDCSW04)*, 2004.