Building a Configurable Publish/Subscribe Notification Service

C. Fiorentino*, M. Cilia**, L. Fiege, A. Buchmann

Databases and Distributed Systems Group, Dept. of Computer Science
Technische Universität Darmstadt, Darmstadt, Germany
<lastname>@dvs1.informatik.tu-darmstadt.de

Abstract. The convergence of technologies and information-driven applications require a middleware that supports data streams. This middleware needs to interpret, aggregate, filter and analyze streams of messages usually in a distributed environment. Publish/Subscribe middleware basically deals with some of these issues, but it is typically monolithic and includes only a subset of features. A problem arises when users want to find a middleware that completely fulfills their application requirements. Based on our experience, we propose a framework that allows the configuration/adaptation of a Pub/Sub solution based on a reusable and extensible set of components.

1 Introduction

New applications and the convergence of technologies, ranging from sensor networks to ubiquitous computing and from autonomic systems to event-driven supply chain management, require middleware platforms that support the handling of streams of data. In these cases, the underlying infrastructure needs to deal with the dissemination of data in a distributed environment from where it is produced to the destinations where consumers are located. Streaming data needs to be interpreted, aggregated, filtered, analyzed, reacted to and eventually disposed of. Publish/Subscribe (Pub/Sub) middleware basically deals with this problem.

Pub/Sub is founded on the principle that producers simply make information available and consumers place a standing request for information by issuing subscriptions. The pub/sub notification service (NS) is then responsible for making information flow from a producer (publisher) to one or more interested consumers (subscribers). Such a notification service provides asynchronous communication, it naturally decouples producers and consumers, makes them anonymous to each other, and allows a dynamic number of publishers and subscribers.

As you can imagine, there are commercial products (WebSphereMQ (formerly MQ-Series), TIB-Rendezvous, or pure JMS) that deal with some of these issues. In academia there are several projects that focus on one or the other issue

* Faculty of Sciences, UNICEN, Tandil, Argentina - cfiorent@exa.unicen.edu.ar
** Also Faculty of Sciences, UNICEN, Tandil, Argentina

of interest, like efficient dissemination, addressing models, message correlation, mobility, scalability, fault-tolerance, access control, data integration, security, privacy protection, transactions, caching, etc.

But specific requirements of a given application may need a combination of some of the previously mentioned aspects. For instance, a news ticker application basically requires topic-oriented addressing, efficient data dissemination and scalability, but does not include transaction support. On the other side, supply-chain management based on the AutoID Infrastructure requires content-based addressing, transactions, security, fault-tolerance, data integration and scalability, while other aspects like access control or caching are optional. Today no product does offer all features. Moreover, these products are often hardly extendible so that missing features cannot be added to fulfill all application requirements.

Based on our own experience in building notification services [3, 8, 9] and specific features for them [10–15] and the experience of others [6, 7, 16–20], we develop a framework for building pub/sub notification services. It facilitates the combination of required features in a notification service, and as a result the middleware is 'shaped' towards application requirements. This offers us a testbed where different ideas can be experimented and proven together with other approaches.

In Section 2 we present the basic idea behind pub/sub notification services including an overview of related projects in this area, and we also include related work on service-based infrastructures. Section 3 enumerates the system requirements and sketches our proposal. In Section 4 we present our main design decisions including architecture, interfaces, and pre-defined components. Section 5 briefly describes the deployment strategy together with the runtime environment. Section 6 presents our conclusions and future work.

2 Background & Related Work

In a notification service there are different participants: *applications*, which basically produce and/or consume messages, and *brokers*, which help to disseminate messages across the network between producers and consumers. Each broker maintains a routing table that determines in which directions a message needs to be forwarded. This table needs to be maintained up-to-date with respect to active consumers and their subscriptions. We basically distinguish three types of brokers: *local brokers* constitute the clients' access point to the middleware and they are usually part of the communication library loaded into the clients. A local broker is connected to at most one border broker. *Border brokers* form the boundary of the distributed communication middleware and maintain connections to local brokers. *Inner brokers* are connected to other inner or border brokers and do not maintain any connections to the applications.

In recent years, academia and industry have concentrated on publish/subscribe mechanisms because they offer loosely coupled exchange of asynchronous notifications, facilitating extensibility and flexibility. These approaches have evolved

from restricted channels to more flexible subscription mechanisms. For instance, subject-based addressing [21] define a uniform name space for messages and their destinations. Here, to every message a subject is attached in order to find matchings with subscriptions, also expressed with subjects.

To improve expressiveness of the subscription model the content-based approach was proposed where predicates on the content of a notification can be used for subscriptions. This approach is more flexible but requires a more complex infrastructure [22]. Many projects in this category (like Rebeca, Siena, JEDI, Gryphon) concentrate on scalability issues in wide-area networks and on efficient algorithms and techniques for matching and routing notifications to reduce network traffic [23–25]. Most of these approaches use simple Boolean expressions as subscription patterns and assume homogeneous name spaces.

More recently a new generation of publish/subscribe systems built on top of an overlay network has emerged. These systems mostly pursue wide-area scalability based on a topic-oriented addressing model. This is the case of Scribe [17] which is implemented on top of Pastry [16]. The mapping of topics onto multicast groups is done by simply hashing the topic name. Hermes [6] uses a similar approach but tries to get around the limitations of topic-based publish/subscribe by implementing a so-called "type and attribute based" publish/subscribe model. It extends the expressiveness of subscriptions and aims to allow multiple inheritance in event types. A content-based addressing on top of a dynamic peer-to-peer network was proposed in [8, 9] where the efficient routing of notifications takes advantage of the topology graph underneath. This work combines the high expressiveness of content-based subscriptions and the scalability and fault tolerance of a peer-to-peer system.

Most of the projects mentioned above are monolithic and they provide a static set of features. Few of them are in some sense extensible while the extensions are hard to develop. What is needed is the possibility to easily combine features in order to completely fulfill application requirements. This can be achieved if pub/sub notification services are built based on extensible set of components that rely on an appropriate architecture/infrastructure. The main requirements of such an infrastructure include: components' life-cycle management, remote and easy deployment, configurability, manageability, and monitoring capabilities. Component containers founded on the principles of inversion of control [4] fulfill some of these requirements.

Pico-Container is a small, light weight container but it does not provide components' management nor runtime configuration. DustDevil (a limited container implemented in Bamboo) is also light weight and has a nice desired communication structure but does not cover most of the previously mentioned requirements. JMX (Java Management eXtension) [2] is a little complex to use and heavy weight (not specially suited for small devices). OSGi (Open Services Gateway Initiative) [1] offers a container and management environment for managing service life-cycle. Services are bundles, JAR files, which contain classes and a configuration file. OSGi has a small footprint and is compliant with the J2ME specification. OSGi itself is built as configurable set services.

3 A Pub/Sub Notification Service Framework

3.1 System Requirements

Based on our own experience on building notification services [3, 8–15] and the experience of others [6, 7, 16–20] we shortly enumerate here the main requirements that, from our perspective, need to be included in the resulting system.

As in every framework, *reusability* is an issue. Here, in particular, we want to reuse a predefined set of specific functionality, like serlialization of messages, routing algorithms, or topology-related strategies. But also *extensibility* in the sense of adding new features plays a fundamental role since optimizations and new routing strategies appear frequently. As notification services are usually distributed in a network of brokers, *easy* and *remote deployment* is a key factor. Moreover, in such a distributed NS *manageability* and *monitoring capabilities* are required in order to tune parameters or replace components if needed.

In most cases the user wants the resulting NS to be *efficient* and *scalable*. The possibility to allow the user to find a trade-off between these two factors is desired. Additionally, the possibility to build an *adaptable* NS where load- or fault-related adaptations can be managed. In many cases, the underlying infrastructure needs to run diverse and concurrent notification services just to fulfill the requirements of diverse applications, resulting on a unreasonable use of resources. This leads to an approach where the *rational use of resources* needs to be taken into account.

3.2 Proposed Approach

We want to provide a framework to build pub/sub notification services according to applications' requirements, selecting from a set of predefined NS characteristics like the underlying NS topology, routing strategies, message serialization and also defining other features like how to deal with failures, caching, security or access control. The resulting NS must be easily deployable. According to this, our goal is to offer an environment that supports the development of the application in question in the following three steps which are also sketched in Figure 1.

1. Based on the application requirements, the set of functions and mechanisms are selected from a pre-established component framework. These components can be simply extended by following clearly established interfaces. Selected components plus some Quality of Service (QoS) decisions are introduced through a configuration tool. After a generate operation, the desired building blocks are ready for the next phase.
2. The resulting NS is deployed by means of a deployment tool.
3. After the NS is deployed on the desired nodes, they can be monitored allowing calibration and tuning of the whole service where QoS parameters can be influenced.

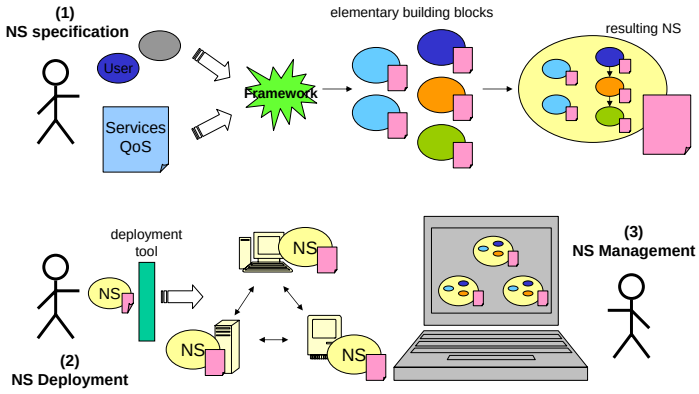

Fig. 1. Our Approach

4 Design Decisions

We have first analyzed and identified possible interactions among elementary building blocks of functionality based on a study that considered the projects mentioned in the related-work section [5]. We then grouped those blocks, what we call here *components*, according to their functionality and interaction patterns. This analysis led us to a layering architecture. Figure 2 shows an overview of the architecture.

Based on the analyzed interaction patterns, we have defined the interfaces between layers. The communication among layers and components can be synchronous or asynchronous. Messages can be passed between layers as events by using the corresponding layer interface.

As it was mentioned before, a pub/sub NS relies on a distributed network of nodes that cooperate. The way this network is modelled has an enormous impact on the design of the infrastructure. For this reason, we assume an *Overlay* network which can be seen as the most generic case, where main characteristics of P2P (like self-organization and healing, robustness, load balancing, and scalability) are considered. Nevertheless, the static tree approach can still be modelled as a restricted overlay that does not offer such characteristics, since the nodes are simply static. Founded on [26] and our own experience we have defined a common overlay interface that condenses the kernel functionality of different overlay networks.

From the runtime perspective we focus our approach on a 'distributed' runtime environment that serves as a container for the resulting NS. This environ-

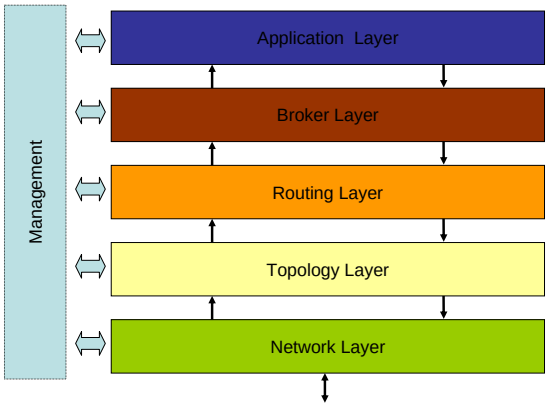

Fig. 2. High-level Architecture

ment should support remote deployment and management, and also monitoring capabilities on the running NS.

The functionality to manage the whole system is orthogonal to the main stack of layers.

4.1 Interfaces

The design of a common set of interfaces between layers was crucial. They make possible to add and change functionality in every layer with minimal impact (if any) on others components.

Because of the nature of the system we are building (which is basically message passing), we can not restrict to synchronous interactions. Consequently, we provide two different kinds of interfaces[1]. Within the *asynchronous interfaces* we apply an event interaction. These interfaces are divided between pair-of-layers messages (also distinguishing between upwards and downwards) and connection-related messages. Figure 3 sketches the asynchronous interactions that may happen between layers.

Synchronous interfaces define a direct communication between layers and components. They are basically defined with the purpose to get or set state of components (in the same or neighbor layers), and also to get/set configuration parameters by the management functionality.

[1] Due to space reasons we cannot present the complete definition of all interfaces.

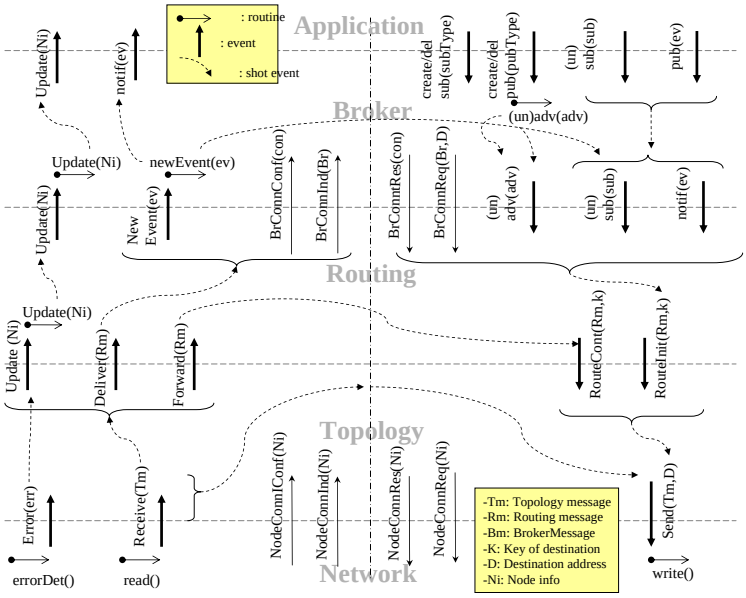

Fig. 3. Interfaces

4.2 Components' Overview

In this section the responsibility of every layer is presented accompanied with a description of possible components at each corresponding layer.

Network Layer: Since this layer is at the bottom of the stack, its responsibility is oriented to the mediation with the physical network. On one side, messages coming from the topology layer need to be serialized and transmitted to a specific network address by using a determined protocol. On the other side, incoming messages from the protocol-side need to be de-serialized and passed to the upper layer. In this layer, different components can be used in order to apply diverse serialization or compression approaches and even different protocols.

Topology Layer: The main responsibility of this layer is to maintain the status information of the overlay up-to-date considering, for instance, dynamic changes like node joins and leaves, or errors. These situations must generate management events that need to be handled accordingly. Possible components within this layer are:

– **DHT**: is a specific kind of overlay network (bases on Distributed Hash Tables) that provides self-stability (dealing with failures and dynamic changes on the participant nodes), high scalability, good distributed object location in wide-area peer-to-peer applications, like Pastry and Bamboo.

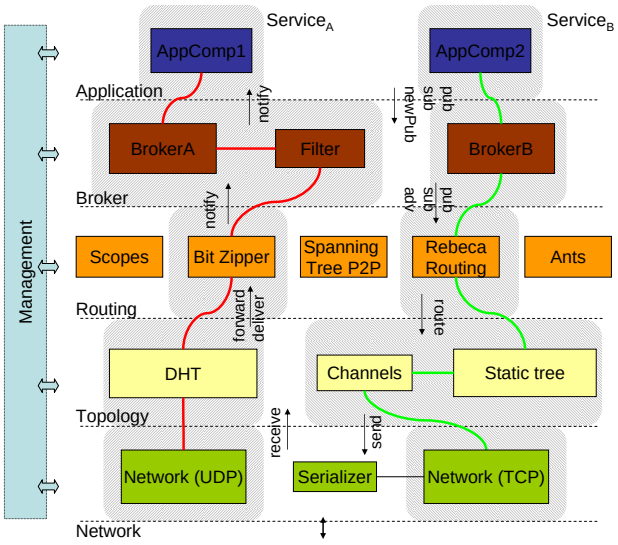

Fig. 4. Layers and components

- **Static tree**: is the simplest case of overlay networks. It is conformed as a static set of nodes, with fixed network physical addresses.
- **Channels**: may be defined as a dynamic set of paths, through which messages are transmitted. They can be thought as shortcuts within the network (reaching more directly different destinations). This improves performance and organization. This component can be combined with other components within the same layer.

Routing Layer: This layer performs the main message routing decisions. According to the addressing model, it must decide where to transmit specific events. Here, subscriptions, publications and advertisings are differentiated to decide when to change routing tables and transmit messages. The Routing Layer basically maintains message destinations according to the available subscription information. The destinations types may vary with respect to the selected service, for example they may be specific node addresses or just DHT keys. Within this layer, diverse components applying different routing strategies can be used, like the ones mentioned below:

- Traditional (Rebeca) routing algorithms [3], like Flooding, Simple Routing, Identity Routing, Covering, or Merging.
- P2P routing algorithms, like Bit Zipper, or the P2P approach presented in [11].
- The Ants approach [18], where the routing strategy is based on real ant behavior. This was simply traduced to a message routing algorithm that

first floods the broker network, and then fills routing tables (according to some probabilities) to better route messages.
- The use of Scopes [27] which is an interesting routing strategy based on a selection criteria to restrict/group the set of message destinations.

Broker Layer: This broker basically offers the Pub/Sub functionality to the application layer. Additionally, it handles client and broker connections and message filtering. In general most events like subscriptions and notification reach this layer to be treated. Nevertheless, not all messages reach this layer due to shortcuts in the lower layers Event correlation filters are handled at this layer. As mentioned before, brokers are classified on a) inner brokers, that manage the connections to other brokers and can apply correlation filters to the flow of messages b) border brokers, that handles connections with clients (which may offer, for instance, a store and forward functionality) and c) local broker, that basically manages the pub/sub interactions between the border broker and the client.

Application Layer: This layer is the one that includes application's logic related to NS usage. It accesses the NS by calling the pub/sub interface always on the broker layer. There may be at least two typical cases where this layer is used. The first one is in the case of P2P applications where the application is also a peer that runs broker functionality (see Figure 4). The other case is characterized by applications acting as a pure end consumer and/or producer of messages. They are not in charge of routing messages since they delegate this task to the corresponding border broker. In this particular case, the topology and routing layers are empty.

4.3 Services

Combining components from different layers is not trivial and there are obviously combinations that are inadequate. Dependencies among components are also represented within the framework. The combination of components crossing all layers is called here a *service*. Different services (or NS instances) can run concurrently within a single runtime environment. Moreover, and with the purpose of better using resources, a single component in a layer may serve various services within the same runtime environment.

In Figure 4 two services that bundle different components across layers is presented. $Service_A$ represents the BitZipper approach [9] relying on P2P infrastructure by using a UDP network connection component. On the other hand, $Service_B$ is characterized by a Rebeca routing strategy [3] relying on a tree topology combined with the use of channels and a traditional TCP network connection.

5 Deployment & Runtime

As can be clearly seen from the interfaces (Figure 3), they basically model three main roles within NS participants. The upward flow of data (left-side of the figure) basically represents *consumers of messages*, since messages are injected from the network layer and they leave the stack at the top (application layer). The downward flow (right-side of the figure) characterizes *producers of messages* since the flow begins at the top of the stack (application layer) and leaves it at the bottom. The third case basically combines the previous two cases and it is characterized by participants that act as brokers within the NS network. This is clearly visible in the figure by the upward flow (up to the broker layer) and the downward flow (up to the network layer) forming an inverted U. The picture also shows possible shortcuts within this flow representing routing optimizations. It must be noticed that NS participants can play just one of these roles or simply all roles simultaneously.

The identification of these flows within the layered architecture leads us to apply another idea for deployment purposes. We adopted a solution that is based on subscribing to components (instead of subscribing to certain messages) that offer certain features. This ends up with a pipeline of components, defining a straight communication between layers.

According to the specified service, a sequence of components can be built which basically represents the previously mentioned flow of messages. Depending on the roles established for participants, different sequences can be built for that purposes. At deployment-time, we build then pipelines of components (with limited control flow just to skip some of them if necessary) that basically process messages. This pipeline is ready to be deployed on a runtime environment.

We have selected OSGi as the runtime platform for our implementation. It offer most of our required functionalities: components life cycle management and remote configuration and deployment, it is light weight and easy to use.

Installing a whole NS is simple by relying on the OSGi platform. We obtain the desired NS as a file that contains all necessary components (in the form of service bundles). With this, the deployment of the NS can be performed (locally or remotely). After deployment, any component can be started and monitored. The system supports dynamic updates of new installed functionalities, gaining flexibility and runtime configurability.

6 Conclusions

We have presented in this paper a framework that allows system engineers to bundle components to satisfy a set of notification service requirements. The layered architecture of building blocks helps to understand, organize, and build notification service. This bundle is then deployed in our runtime environment that controls component life-cycle and offers monitoring capabilities that allow tuning and adaptation of the resulting NS. The presented solution covers pub/sub

NSs that range from a light weight single purpose static NS to multi-broker, dynamic, remotely manageable NS. The decision of relying on an abstract overlay network simplifies the unification of different topology approaches.

Our solution for building pub/sub NSs can be used as a testbed for improvements on routing algorithms and other ongoing research projects. By having reusable components at hand, it is easy to pick the functionality you require for your experiments and also for probing these ideas under diverse NS constellations.

We have not (fully) automated the process of searching and selecting components from a repository. This is part of our ongoing work. Another pending task is a performance analysis comparing native pub/sub notification services like Rebeca, Hermes or BitZipper with their implementation based on our framework.

References

1. OSGi Alliance. The OSGi Service Platform. Technical report, July 2002.
2. Sun Microsystems. Java Management Extensions. White paper, 1999.
3. Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems.* PhD thesis, Darmstadt University of Technology, Germany, September 2002.
4. Martin Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, January 2004.
5. Cristian Fiorentino. Building a configurable notification service (under preparation). Master's thesis, Faculty of Sciences, UNICEN, Tandil, Argentina, April 2005.
6. Peter Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In J. Bacon, L. Fiege, R. Guerraoui, A. Jacobsen, and G. Mühl, editors, *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, July 2002.
7. Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX '04)*, Boston, Massachusetts, June 2004.
8. Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A Peer-to-Peer approach to Content-Based publish/subscribe. In *In Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, June 2003.
9. Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Jussi Kangasharju, and Alejandro Buchmann. Bit zipper Rendezvous—Optimal data placement for general P2P queries. In *EDBT 04 Workshop on Peer-to-Peer Computing & DataBases*, 2004.
10. Mariano Cilia, Ludger Fiege, Christian Haul, Andreas Zeidler, and Alejandro Buchmann. Looking into the past: Enhancing mobile publish/subscribe middleware. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, San Diego, California, June 2003. ACM Press.
11. Ludger Fiege, Felix C. Grtner, Oliver Kasten, and Andreas Zeidler. Supporting mobility in Content-Based publish/subscribe middleware. In *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, pages 103–122, June 2003.
12. Ludger Fiege, Andreas Zeidler, Alejandro Buchmann, Roger Kilian-Kehr, and Gero Mhl. Security aspects in publish/subscribe systems. In *Third Intl. Workshop on Distributed Event-based Systems (DEBS'04)*, May 2004.

13. Mariano Cilia, Mario Antollini, Christof Bornhvd, and Alejandro Buchmann. Dealing with heterogeneous data in pub/sub systems: The Concept-Based approach. In *International Workshop on Distributed Event-Based Systems (DEBS'04)*, Edinburgh, Scotland, May 2004.
14. Jose Antollini, Mario Antollini, Pablo Guerrero, and Mariano Cilia. Extending rebeca to support Concept-Based addressing. In *In Proceedings of the Argentinean Symposium on Information Systems (ASIS'04)*, Cordoba, Argentina, September 2004.
15. M. Cilia, C. Bornhövd, and A. Buchmann. CREAM: An Infrastructure for Distributed, Heterogeneous Event-based Applications. volume 2172 of *LNCS*, Italy, November 2003. Springer.
16. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.
17. Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.
18. M. Gunes, U. Sorges, and I. Bouazzi. Ara – the ant-colony based routing algorithm for manets, 2002.
19. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.
20. Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.
21. B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus – An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th Symposium on Operating Systems Principles (SIGOPS)*, pages 58–68, USA, December 1993.
22. A. Carzaniga, D. R. Rosenblum, and A. L. Wolf. Challenges for Distributed Event Services: Scalability vs. Expressiveness. In *Engineering Distributed Objects (EDO'99)*, Los Angeles, CA, May 1999.
23. Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman. Exploiting IP Multicast in Content-based Publish-Subscribe Systems. volume 1795 of *LNCS*, pages 185–207. Springer, 2000.
24. G. Mühl, L. Fiege, and A.P. Buchmann. Filter Similarities in Content-Based Publish/Subscribe Systems. In *Proc. of ARCS*, volume 2299 of *LNCS*, 2002.
25. F. Fabret, F. Llirbat, J. Pereira, A. Jacobsen, K. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. pages 115–126, 2001.
26. Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, February 2003.
27. L. Fiege, M. Mezini, G. Mühl, and A.P. Buchmann. Engineering Event-Based Systems with Scopes. In *Proc. of ECOOP'02*, volume 2374 of *LNCS*, 2002.