Supporting Mobility with Persistent Notifications in Publish/Subscribe Systems

Ivana Podnar and Ignac Lovrek
Department of Telecommunications
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, HR-10000 Zagreb, Croatia
ivana.podnar@fer.hr, ignac.lovrek@fer.hr

Abstract

The paper proposes a novel approach to mobility in publish/subscribe (pub/sub) systems that relies on notification persistency. The existing solutions apply proxy subscribers that take the role of disconnected subscribers, store notifications published during disconnections, and deliver them to subscribers after reconnection to the pub/sub system. The system perceives a constant number of subscribers which is inefficient if it serves a large number of subscribers that are often disconnected. We argue that the system should not be burdened by proxy subscribers during subscriber disconnections, but rather store persistent notifications, and deliver valid notifications to subscribers when they reconnect to the system. Preliminary experiments show that the proposed solution can improve the observed scalability weakness of the existing solutions.

1. Introduction

The inherent characteristics of pub/sub middleware, such as loose coupling, asynchronous and persistent communication, and system extendibility, are found suitable for the design of mobile applications that need to adapt to highly dynamic and volatile conditions in mobile environments [3]. The existing pub/sub systems are optimized for stationary environments [2, 4, 8, 6]. The mobility-related operation is managed by the application layer through a sequence of *subscribe-unsubscribe-subscribe* requests: A subscriber defines subscriptions when connecting to a pub/sub system, unsubscribes prior to disconnection, and re-subscribes after reconnecting to the system. However, the subscriber will not receive notifications that are published during the time of disconnection. The *queuing approach* is commonly applied to solve the problem of lost notifications [1, 5, 10]. A system broker, or a special proxy acts as a proxy subscriber during subscriber disconnections. The proxy stores notifications published during disconnections in a special subscriber's queue, and delivers them to the subscriber after its reconnection. If the subscriber reconnects through a new system broker, a handover procedure is performed which transfers the stored notifications to the subscriber and updates its subscriptions in the broker network.

We propose a novel approach to mobility in pub/sub systems: Rather than queuing notifications for disconnected subscribers, the network of brokers stores persistent notifications until their validity period expires. We argue that notification publishers need to define the validity period of published notifications, the system must assure notification storage, and deliver valid notifications to subscribers when they reconnect to the pub/sub system. If a subscriber connects to the system after notification expiry, the notification will not be delivered to the subscriber because it no longer holds valuable information. Notification persistency is by no means a new characteristics in the existing pub/sub systems: The Java Message Service (JMS) supports mobility using durable subscriptions and persistent notifications. However, the available JMS implementations are mainly centralized and offer no routing optimizations for distributed pub/sub architectures.

We have implemented a prototype system to show the adequacy of the proposed solution. The prototype can be distinguished from other pub/sub implementations by the inherent support for publisher and subscriber mobility. Furthermore, we have used the prototype implementation to evaluate the proposed solution, and to compare it with the approach based on queues.

The paper is structured as follows. Section 2 presents the proposed solution: We outline the extensions of the routing algorithms to support mobility using persistent notifications, and discuss the characteristics of the approach. Section 3 describes the prototype implementation, defines the metrics for performance evaluation of pub/sub systems, and presents initial evaluation results that compare the proposed approach with the queuing approach. In Section 4, we discuss mobility solutions in the existing systems, and Section 5 concludes the paper.

2. Mobility Support with Persistent Notifications

A pub/sub system comprises a set of publishers and a set of subscribers that interact by performing actions. The occurrence of an action is an event that changes a system state. The events occurring in the system are *connect*, *disconnect*, *publish*, *subscribe*, *unsubscribe*, and *notify*. Publishers and subscribers connect and disconnect from the system, and change the system state by modifying the set of connected publishers and subscribers. Publishers publish notifications using the event *publish*, and change the set of published persistent notifications. Publishers define a validity timestamp which specifies the validity period of a published notification. A notification is removed from the set of persistent notifications when its validity period expires. Subscribers activate subscriptions by generating an event *subscribe*, and invalidate them using the event *unsubscribe*. The system is responsible for matching a published notification to the set of active subscriptions, and for notifying subscribers with a matching subscription about the existence of the notification using the event *notify*.

A pub/sub system may have a distributed architecture to solve the scalability problem which is particularly important in mobile environments where clients roam in different network domains and can connect to “the closest” broker. A distributed pub/sub system comprises a set of interconnected brokers that exchange messages carrying events to maintain a consistent view of active subscriptions and persistent notifications in the system. Brokers exchange subscription messages to create *delivery paths* connecting a publisher to a set of potential subscribers. The routing algorithms used for creating delivery paths are mainly based on *reverse path forwarding* [2, 6], and use subscription equality or subscription covering to reduce the number of messages exchanged between system brokers, and the number of entries in broker routing tables. A recently proposed solution applies an algorithm based on core trees [8].

The existing solutions for client mobility in distributed pub/sub systems use the queuing approach (Q-app) for storing notifications in special queues during subscriber disconnections [1, 5]. We propose an approach based on persistent notifications (PN-app) that requires the delivery of valid notifications after the activation of a new subscription in the system. When a subscriber reconnects to the system, it reactivates its subscriptions and therefore receives valid stored notifications. The PN-app is an extension of the subscribe-unsubscribe-subscribe procedure that does not modify the applied routing algorithm, e.g., the reverse path forwarding, or the core-based tree algorithm. Here we shortly describe the extensions of the basic algorithm:

- When a broker receives a *publish* message, it first stores the notification in the persistent notification container, and subsequently delivers it to interested subscribers and neighboring brokers. The broker removes the notification from the container when its validity period expires, and maintains a list of valid notification ids that have been sent to subscribers and brokers.
- When a broker receives a *subscribe* message, either from a subscriber, or a neighboring broker, it checks the list of persistent notifications and delivers a matching valid, and previously undelivered notification to the subscriber, or the broker.
- When a broker receives a *connect* message from a subscriber, it activates the subscriber’s subscriptions. The subscriber must provide a list of active subscriptions and a list of received valid notification ids to avoid duplicate notifications.
- When a broker receives a *disconnect* message from a subscriber, it deactivates the existing subscriber subscriptions, and terminates delivery paths in the broker network.

The mechanism that compares the list of valid received notification ids to those that should be sent to a subscriber, or a neighboring broker prevents the delivery of duplicate notifications. We assume that notification ids are unique within the system, and that they are removed from the list when notifications expire. A notification can be undelivered because a broker network may introduce the delay that can lead to notification expiry prior to its delivery to a subscriber.

The differences between the Q-app and the PN-app are in the following:

- **Notification storage.** In case of the Q-app, system brokers store queues per each disconnected subscriber. For the PN-app, brokers maintain persistent notifications, and the list of valid notifications sent to subscribers and neighboring brokers.
- **Subscriber’s reconnection to the system.** When applying the Q-app, a reconnecting subscriber first reactivates its subscriptions at the new broker to update the delivery path in the broker network, and next, the new broker retrieves the notifications from the subscriber’s queue maintained by the old broker and delivers them to the subscriber. In case of the PN-app, the subscription reactivation will initiate the delivery of valid notifications along the new delivery path to the subscriber. Prior to notification delivery to the client, the edge broker checks whether the subscriber has already received a valid notification by comparing it to the list of received notification ids.
- **Subscriber’s data.** A subscriber in the system applying the Q-app needs to know the identifier of the old broker together with the list of active subscriptions to reconnect to the system. In case of the PN-app, a list of received and valid notification ids, and the list of active

subscriptions is needed.

- **Perceived number of system subscribers.** Subscriber queues act as proxy subscribers for disconnected clients which gives the impression that subscribers are constantly active in a system that uses the Q-app. The PN-app maintains no active subscriptions for disconnected subscribers.

A persistent notification is stored by a subset of network brokers until its validity period expires. It is maintained by a single broker if, at the time of its publishing, there were no remote subscribers for the notification. The notification will eventually reach a reconnecting subscriber following a newly-created delivery path, and be stored on each broker it traverses. The subscriber might have already received the notification if it has previously resided on the broker through which the notification has been published: The notification will be routed to the new broker, however, it will not be delivered to the subscriber since its id is in the list of subscriber's received notifications. This in the known overhead of the approach that causes superfluous traffic in the broker network, and increases the usage of broker memory and processing time. At the other extreme is the situation in which all brokers have a notification copy. A reconnecting subscriber will receive a notification copy from the access broker without causing extra traffic in the broker network.

Potential advantages of the proposed approach when compared with the Q-app are the following: avoidance of the handover procedure that transfers notifications from the old to the new broker, reduced size of broker routing tables due to decreased number of perceived subscribers in the system, and memory consumption related to the storage of notifications in the system. The expected disadvantage is related to control traffic: The PN-app generates an increased number of subscriptions and unsubscriptions for terminating the old and creating the new delivery paths. The Q-app suffers from the same problem if the probability that a subscriber reconnects to the same broker is low. If subscribers frequently connect to the same broker, the number of control messages is reduced because there is no need to update an existing delivery path. The maintenance of the list of received and valid notification ids is an additional broker overhead in case of the PN-app.

3. Evaluation

We use the implementation of the prototype system MOPS (**Mo**bile **P**ublish **S**ubscribe) to investigate the adequacy of the proposed approach, to evaluate the system performance in mobile environments, and to compare it with the system performance when applying the Q-app. Both approaches are used with the routing algorithm based on subscription covering.

3.1. The Prototype System MOPS

The MOPS infrastructure comprises a set of interconnected brokers that form an acyclic communication graph. There is a single spanning tree for notification delivery connecting a publisher to a group of subscribers, and each broker is a single point of system failure. The broker network is built incrementally by connecting a new broker to an active broker, and can be extended during system operation. Clients, i.e., publishers and subscribers, are mobile entities that can connect to different brokers. The communication is realized in the form of messages transported using TCP to assure reliable communication. The communication between a client and a broker is truly push-based without open connections. A client registers as connected with a broker, and provides a time-to-live parameter (TTL) specifying the period it expects to be served by the broker. The broker maintains a list of connected clients, and removes a subscriber from the list in case a notification cannot be delivered to the subscriber prior to TTL expiry.

The MOPS system supports typed notifications that carry a list of attributes. It offers type-based and attribute-based subscriptions, and implements type-based routing with support for subscription covering in the broker network. Attribute-based notification filtering is performed by the edge broker prior to notification delivery to subscribers. What distinguishes it from other pub/sub prototype implementations is the inherent support for publisher and subscriber mobility that has been integrated into the system design, rather than added as an extension to an existing system supporting stationary clients. The system can be configured to use either queues for storing notifications on behalf of disconnected subscribers, or persistent notifications maintained by the brokers.

3.2. Metrics

The performance of a pub/sub system in a dynamic environment with mobile clients is largely influenced by its efficiency. We propose the usage of the following metrics for mobile pub/sub system evaluation:

- **Broker processing load.** The processing load experienced by a broker can be measured by the rate of processed messages. Messages carrying notifications transport the actual information, while subscription and unsubscription messages represent control load that creates and updates delivery paths. We differentiate between received and sent messages, and classify them according to the type of events they transport.
- **Bandwidth consumption.** A desirable property of a distributed pub/sub system is to consume minimal bandwidth. The rate of processed messages, as in the case of broker processing load, gives a good estimate

of the physical bandwidth consumption.

- **Notification delay.** Efficient notification delivery requires minimal delay, i.e., the period between notification publication and receipt. In case of mobile subscribers, the delay is increased due to subscriber disconnections from the system, and it depends on the duration of disconnection periods and notification validity periods.

3.3. Preliminary Experimental Results

We show the experimental results that enable preliminary assessment of system performance when applying the PN-app, and the Q-app. The results are obtained using a working prototype that simulates the real working environment, instead of a model simulation. There are some implementation differences that cause an increased processing load for system brokers in case of the PN-app due to a garbage collector that purges expired notifications from persistent notification containers. Therefore, we have decided to use the metrics that are not largely influenced by the processing latency to enable a just comparison of the two approaches. The experiment investigates the broker's processing load, and bandwidth consumption in terms of the rate of processed messages, and the number of stored notifications. The experiment does not investigate the notification delay because it is largely influenced by the applied routing policy, which is the same for both approaches, and by the actual system implementation which would favor the Q-app.

We ran the experiment under the same initial conditions for the Q-app and the PN-app. After forming a network of brokers, we initiated a set of mobile subscribers and stationary publishers. Each experiment run lasted 20 minutes, and we conducted 5 runs with the same initial setting.

Input parameters. We are using a network of seven brokers forming a tree structure. The number of publishers is constant, $p = 7$, and each publisher is stationary and connected to one of the brokers. Publishers publish notifications at a constant rate of $pubRate = 0.5$ notifications/s. We use a complex type hierarchy consisting of 20 types, and publishers generate notifications of a randomly chosen type with a uniform probability. Each notification carries a payload of 100 bytes. The validity period for notifications in case of the PN-app is set to 5000 ms to avoid undelivered notifications.

We varied the number of subscribers in the system, $s = 1, 5, \ldots, 35$. Subscribers are mobile and can connect to all system brokers except to B_1 because it is the root node of the broker network, and therefore the system bottleneck. We use the random mobility model in the experiment: A subscriber chooses the next broker randomly from the set of available brokers. Each subscriber connects to a new broker with a constant connection rate in the range from 0.2 to 0.6 connections/s, and the connection duration is 50% of the connection period.

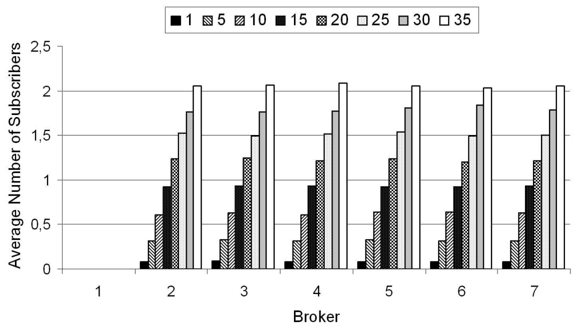

Figure 1: Connected subscribers per broker

Figure 1 shows the measured average number of subscribers connected to a broker as the total number of subscribers in the system changes. It is visible that subscribers do not connect to B_1, and that other brokers evenly share the subscriber load. In case of the total of 15 subscribers in the system, there is on average one subscriber connected to each broker. All subscribers subscribe to the top notification type, and should receive all published notifications. We use the simple subscription scenario to test the implementation and check that notifications are received without duplicates.

Experimental results. Figure 2 shows the average number of stored notifications on a broker as the number of subscribers in the system increases. As expected, the number of stored notifications increases linearly with the number of subscribers for the Q-app, since notifications are stored per each disconnected subscriber in a separate queue. The number of stored notifications for the PN-app reaches its maximum value as soon as each published notification is stored once on each broker and remains constant regardless of the number of subscribers in the system.

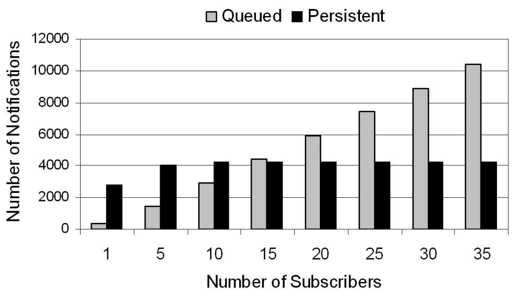

Figure 2: Queued and persistent notifications

Figure 3 shows the average rate of *received and sent notification messages* per each broker. The rate of received notification messages increases for both approaches until

$s = 15$ when there is on average 1 subscriber connected to each broker, and reaches the maximum value determined by the number of system publishers, and their publishing rates. Clearly, the rate of sent notification messages increases as the number of subscribers increases, because the number of notification destinations increases accordingly. The Q-app generates a larger number of notification messages than the PN-app when $s \geq 7$, because in case of the Q-app, notifications are sent to subscriber queues during the disconnection period, while in case of the PN-app, notifications are cached on brokers they traverse, and from there delivered to reconnecting subscribers. The notification rate in case of the Q-app is further increased by notification exchange during the handover procedure.

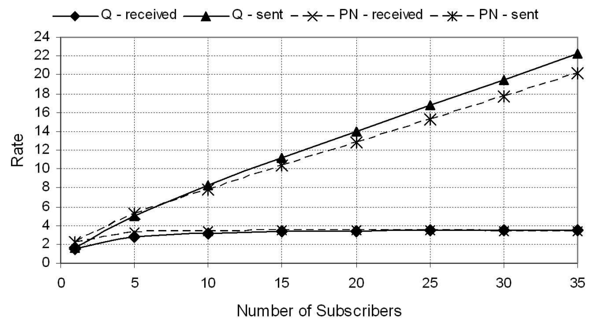

Figure 3: Rate of received and sent notifications

Figure 4 shows the average rate of *received and sent unsubscription messages* per broker. As expected, the Q-app generates less unsubscription messages than the PN-app because the Q-app does not generate unsubscription messages in case the old and the new broker are the same, which is the case for the PN-app. The rate of sent unsubscription messages decreases as the number of subscribers increases, because there is no need to propagate unsubscriptions since there are other subscribers with an active matching subscription that are connected to brokers.

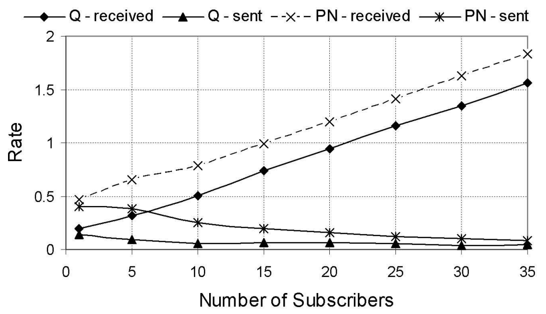

Figure 4: Rate of received and sent unsubscriptions

Finally, Figure 5 depicts the average rate of all received and sent messages per broker and can be used to asses the broker processing load. It is visible that the PN-app poses less load on a broker as the number of subscribers in the system increases. The gain is 10% in this particular experiment when $s = 35$, and would become significant in case of a large number of subscribers.

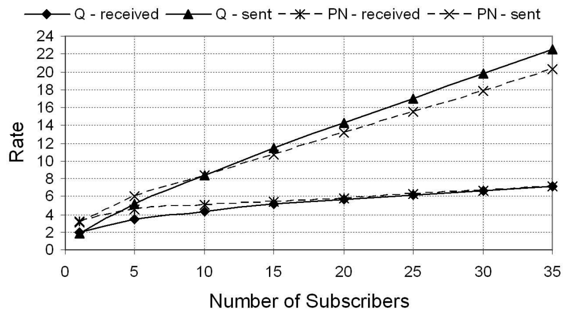

Figure 5: Rate of received and sent messages

To conclude, the PN-app is superior when compared to the Q-app with respect to the consumption of broker memory for storing undelivered notifications if we assume that the number of subscribers in the system is significantly larger than the number of brokers. Furthermore, the PN-approach introduces less processing load on brokers for the conducted experiment as the number of subscribers in the system increases, and therefore consumes less bandwidth on links connecting the brokers. The preliminary results show that the load introduced by control messages for the PN-app is acceptable when compared to the Q-app.

4. Related Work

A number of authors agree that the mobility support in pub/sub systems should be offered by the pub/sub middleware itself, and not delegated to the application layer [9, 12]. The existing solutions extent the established pub/sub systems that are optimized for stationary environments by adding special proxy subscribers taking the subscriber's role for disconnected subscribers [1, 5].

The mobility service developed within the project SIENA [1] uses *client proxies* and a special *client library* to manage subscriptions on behalf of a subscriber. A client proxy runs as a special component at an access point, and stores messages for a disconnected subscriber in a special queue. The client library mediates a move-out procedure from the old broker, and a move-in procedure when a subscriber reconnects to the new broker. The old and the new proxy perform a handover procedure that transfers messages from the old proxy to the new one, and subsequently to the subscriber. Clearly, the overhead of the proposed solution is the existence of a new component in system, the client proxy. The reported experimental results prove the applicability of the implementation, investigate the number of duplicate and lost messages, but define no other metrics to evaluate system performance. Best to the author's knowledge, the presented experimental results are the first attempt to evaluate pub/sub system performance in a mobile setting.

The solution presented in this paper is compliant with the algorithm for physical mobility developed within the project REBECA [5, 12] that uses the Q-app where the last serving broker takes the role of a proxy subscriber. When a subscriber connects to a new broker, it re-issues its subscriptions, and the broker network finds the old broker by locating a broker at the junction of delivery paths to the new and the old broker. It is straightforward to find a junction broker if the broker network applies simple routing, because each broker keeps entries about all active subscriptions. The notifications stored by the old broker are routed through the junction to reach the new broker, and subsequently the subscriber. The authors do not justify why subscribers cannot maintain the information about the last visited broker which significantly complicates the algorithm that needs to locate the junction broker. The algorithm needs further extensions in case routing based on covering is applied since simple routing generates large routing tables. There are currently no results that evaluate the performance of the approach.

Mobility support in Elvin [10] is one of the first implementations offering mobility to subscribers in a pub/sub system. The solution puts a proxy server between the original server and a mobile device for storing messages for disconnected subscribers. A subscriber must always connect to the central proxy server which can become a performance bottleneck and induce significant network traffic due to potential triangular routing.

Recently, some of the systems that implement the JMS specification support mobility [7, 11] by offering a lightweight JMS-compliant API for Java-enabled mobile terminals that can be used to implement JMS-based publishers and subscribers. However, the available implementations offer no routing optimizations for distributed architectures.

5. Conclusion

The paper presents an approach to mobility in pub/sub systems that uses a sequence of subscribe-unsubscribe-subscribe requests, and relies on notification persistency to assure the delivery of notifications published during subscriber disconnections. We have investigated the adequacy of the approach using a prototype system, presented initial results that investigate its performance, and compared it to the solution based on queues. The preliminary experimental results show that the proposed approach is superior to the queuing approach with respect to the broker processing load and memory consumption as the number of subscribers in the system increases, assuming that subscribers move following the random mobility model. Future work should examine the notification delay introduced by the approach, and investigate system performance for different subscriber mobility models and subscription scenarios.

Acknowledgements. The authors thank Branimir Turk and Matija Mićin for their contributions to the implementation of the MoPS system.

References

[1] M. Caporuscio, A. Carzaniga, and A. L. 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.

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

[3] G. Cugola and H.-A. Jacobsen. Using publish/subscribe middleware for mobile systems. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):25–33, 2002.

[4] G. Cugola, E. D. 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):827–50, September 2001.

[5] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. In *Proc. of the Middleware 2003*, volume 2672 of *LNCS*, pages 103–122. Springer-Verlag, June 2003.

[6] G. Mühl, L.Fiege, F. C. Gärtner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *Proc. of the 10th IEEE Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, pages 167–176. IEEE Computer Society, Oct. 2002.

[7] ObjectWeb Open Source Middleware. JORAM (release 3.6.0), August 2003. http://www.objectweb.org/joram/.

[8] 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 (DEBS'03)*. ACM Press, June 2003.

[9] I. Podnar, M. Hauswirth, and M. Jazayeri. Mobile Push: Delivering content to mobile users. In *Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS'02)*, pages 563–568. IEEE Computer Society, July 2002.

[10] P. Sutton, R. Arkins, and B. Segall. Supporting disconnectedness – Transparent information delivery for mobile and invisible computing. In *Proc. of the IEEE Int. Symp. on Cluster Computing and the Grid*, pages 277–285. IEEE Computer Society, May 2001.

[11] E. Yoneki and J. Bacon. Pronto: MobileGateway with publish-subscribe paradigm over wireless network. Technical Report UCAM-CL-TR-559, Computer Laboratory, University of Cambridge, 2003.

[12] A. Zeidler and L. Fiege. Mobility support with REBECA. In *Proc. of the 2nd Int. Workshop on Distributed Event-Based Systems (DEBS'03)*, pages 354–360, May 2003.