Reliable Publish/Subscribe in Dynamic Systems

Ugaitz Amozarrain, Mikel Larrea
Computer Architecture and Technology Department
University of the Basque Country UPV/EHU
Paseo Manuel de Lardizabal 1
20018 Donostia-San Sebastián, Spain
Email: uamozarrain001@ikasle.ehu.eus, mikel.larrea@ehu.eus

March 15, 2016

Abstract

This work addresses content-based publish/subscribe in dynamic scenarios. We present an approach based on the simple routing strategy, that is incrementally extended in order to support client and broker mobility.

1 Introduction

Publish/subscribe is a paradigm that allows a loosely coupled dissemination of events on a distributed network. The objective of such a system has been the distribution of the messages over a wide area and static topology [3, 11].

The increase of wireless devices on recent years, be it wireless sensor networks, smart-phones or other mobile devices has created a need to adapt the initial publish/subscribe protocols designed for static systems. Indeed, in wireless sensor networks we lose the robustness of the Internet and focus more on challenges such as the mobility of the network [2, 5, 6, 7, 9, 10, 14, 15]. Often the simplest approach is to use message flooding, though this creates an unnecessary overhead for the network and the devices on it.

Mobility is an intrinsic property of a wireless network and the protocols designed for it should take this into account. Devices are able to freely move constantly changing the topology of the network.

Following a previous work [13] where client mobility for content based publish/subscribe systems was introduced, we expand it to include mobility also on the broker network. With this we allow a full range of dynamicity for all devices that compose the publish/subscribe system ensuring that messages are delivered to connected clients even while part of the broker network is disconnected from the rest.

2 System Model

A basic publish/subscribe system is composed of two main components: a set of *clients* that produce and consume events, and a *notification service* that

handles the subscriptions issued by clients and the delivery of events to the corresponding clients.

There are two types of clients: *publishers* that produce events and send them to the system, and *subscribers* that register to one or more filters and consume events. We will use $p \in P$ to refer to a publisher belonging to the global set of publishers P. Similarly, we will use $s \in S$ to refer to a subscriber belonging to the global set of subscribers S. Any client can behave as a publisher, subscriber or even both at the same time. Finally, we will use $f \in F$ to refer to a filter belonging to the global set of filters F.

On the other hand the notification service is composed of a set of brokers B which we will refer to by using the notation $b \in B$. This set of brokers will be connected at the logical level by an acyclic graph. The brokers will store the subscriptions issued by subscribers and route correctly the published events. A broker will have at any moment a set of brokers it can communicate with, which will be its neighbors in the graph, we will call this set N_i for broker b_i. Furthermore, a broker will also communicate directly with clients that are connected to it, for this reason we call I_i to the set of interfaces that broker b_i can communicate with, be it clients or other brokers. All communications are by point-to-point message passing over reliable and FIFO links.

3 From Static to Dynamic Publish/Subscribe

This section describes our incremental approach to support dynamicity in publish/subscribe. First, we present a Simple Routing protocol for a system where both clients and brokers are static. Then, we describe Phoenix [13], a protocol that extends the Simple Routing protocol in order to support dynamic clients. Finally, we present our extension to Phoenix in order to support dynamic brokers.

3.1 Simple Routing

This section presents a publish/subscribe routing protocol which implements the Simple Routing [1] strategy in a static system where brokers are connected in an acyclic graph, and each client is permanently bound to a single broker. This routing strategy is based on the propagation of subscription and unsubscription messages to all of the brokers in the system. Every broker b_i maintains a routing table R_i that is based on the received *SUB* and *UNS* messages and models the subscriptions in the system. The routing tables enable brokers to filter incoming events received as *PUB* messages, and forward them only towards those subscribers with matching subscriptions.

Algorithm 1 is ran by every broker in the system and routing tables contain, for every subscription in the system, a routing entry (f, z) where $f \in F$ and $z \in I_i$, to indicate that the publication of an event e matching f must either be forwarded towards broker z (if $z \in B$) or delivered to subscriber z (if $z \in S$).

A generic correctness proof of the Simple Routing strategy can be found in [8]. In addition, the interested reader is referred to [12] for the correctness proof of Algorithm 1.

```
1  when receive(SUB, f) from z ∈ I_i do
2      R_i ← R_i ∪ {(f, z)}
3      foreach b ∈ N_i where b ≠ z do
4          send(SUB, f) to b

5  when receive(UNS, f) from z ∈ I_i do
6      R_i ← R_i \ {(f, z)}
7      foreach b ∈ N_i where b ≠ z do
8          send(UNS, f) to b

9  when receive(PUB, e) from z ∈ I_i do
10     X ← ∅
11     foreach (f, y) ∈ R_i where y ∉ X ∧ y ≠ z do
12         if f(e) = true then
13             X ← X ∪ {y}
14     foreach y ∈ X do
15         send(PUB, e) to y
```

Algorithm 1: Simple Routing (code executed by broker b_i)

3.2 Supporting Dynamic Clients

Phoenix [13] is an extension to the Simple Routing protocol that is able to seamlessly handle subscriber migrations (publisher migration is inherently supported by Simple Routing). In order to achieve this some changes had to be made to the Simple Routing protocol of Algorithm 1. In the usual case, when a subscriber changes the broker it uses to connect to the network, the subscriber would have to send the subscription messages again to the new broker in order to continue receiving events. To avoid the potential flooding of the network that such migrations might cause, Phoenix extends the routing table of the brokers to include the subscriber that registered each filter. Using this information the broker can route not only publication messages, but also messages regarding the subscriber.

Two new messages were added on top of the preexisting ones in the protocol of Algorithm 1. A *MIG* message used to notify the broker network when a subscriber has migrated, and a *REP* message used to send the subscriber any event it might have not delivered during the migration. Both of these messages will follow the delivery path of a published message so there is no need to flood the network.

In order to adapt the Phoenix protocol for supporting broker migrations, some changes have been made, see Algorithm 2. For simplicity on this first approach the replaying of undelivered messages has been omitted. On the handling of *SUB* and *UNS* messages we now check for the existence of the values on the routing table, seen on lines 2 and 7, since if the network is still converging

```
1  when receive(SUB, f, s) from z ∈ I_i do
2      if ∄(f, _, s) ∈ R_i then
3          R_i ← R_i ∪ {(f, z, s)}
4      foreach b ∈ N_i where b ≠ z do
5          send(SUB, f, s) to b

6  when receive(UNS, f, s) from z ∈ I_i do
7      if ∃(f, _, s) ∈ R_i then
8          R_i ← R_i \ {(f, _, s)}
9      foreach b ∈ N_i where b ≠ z do
10         send(UNS, f, s) to b

11 when receive(PUB, e) from z ∈ I_i do
12     X ← ∅
13     foreach (f, y, _) ∈ R_i where y ∉ X ∧ y ≠ z do
14         if f(e) = true then
15             X ← X ∪ {y}
16     foreach y ∈ X do
17         send(PUB, e) to y

18 when receive(MIG, s, b) from z ∈ I_i do
19     if z = s then
20         X ← ∅
21         foreach (f, _, s) ∈ R_i do
22             X ← X ∪ {f}
23         send(FILTERS, X) to s
24     if b ≠ b_i then
25         if ∃(_, _, s) ∈ R_i then
26             b_j ← y ∈ N_i where (_, y, s) ∈ R_i
27             send(MIG, s, b) to b_j
28     foreach (_, _, s) ∈ R_i do
29         replace (_, _, s) with (_, z, s) in R_i
```

Algorithm 2: Simple Routing with dynamic clients

a broker could receive some of the messages out of order.

Some changes were also made in the handling of a *MIG* message so that the first broker that receives the message will send back to the subscriber a set containing all the subscriptions for that subscriber from its routing table, see lines 19-23. A new message type *FILTERS* was added to send this information. This is done in order to support the migration of the subscribers from a partition

to another. If the network is partitioned and a subscriber is connected to just one of the partitions, the other ones do not receive any *SUB* messages. And if the subscriber then migrates to another partition this partition will have no information on the subscriber so it will depend on the subscriber to review the set of subscriptions and correct it if necessary. Another change consists in the subscriber sending the identity of the previous broker it was connected to, which together with the check of line 24 allows us to handle reconnections as a particular case of migration. The inclusion of the check on line 25 is for the same reason *SUB* and *UNS* handling had to be changed, we need to check for consistency on the routing table. Finally, the entries of the routing table corresponding to the migrating subscriber are adjusted in order to correctly route future events.

3.3 Supporting Dynamic Brokers

After reviewing mobility support for clients, in this section we introduce an extension to the protocol for supporting broker mobility and crash-recovery. Unlike subscriber mobility, when a broker migrates, the physical change on the network might force a recalculation of the spanning tree that is used for routing the events. Using standard spanning tree algorithms this recalculation might end up changing several of the previous stable connections between brokers, thus potentially forcing the migration of more brokers than the one that is actually migrating.

In order to avoid this issue a leader election algorithm [4] was chosen for the broker network. Using the periodic heartbeat message from the leader that the algorithm uses we can create a spanning tree with the leader itself as the root node. Whit this heartbeat message the brokers will realize when they have moved from their previous position on the network and migrate accordingly. The leader election algorithm is also able to handle small independent partitions on the network and once connectivity is restored between them, they come together forming a bigger network with a unique leader. For all migrations we consider the partition with the final leader as the primary partition and the others as migrating partitions.

Each broker will keep stored the next hop to the leader, following the opposite path of the heartbeat message, and a set of brokers that are connected to itself. Using this information we can easily create a spanning tree. All brokers keep a set of connected subscribers which will change according due to the connectivity of the subscribers. We will refer to this set as C_i for broker b_i.

We consider that a broker has migrated when its path to the leader changes. Taking as a reference the initial network of Figure 1a, an example of broker migration can be seen in Figure 1b. In this case the migrating broker b_i (b_5 in Figure 1b) will connect to a new broker b_j (b_2 in Figure 1b) designated as next hop for the leader and send a *BMIG* message. This message contains the set C_i of local subscribers connected to b_i.

This message causes the broker to function as a proxy for subscriber migration and it will follow the delivery path for those subscribers as with single subscriber migrations, without the need to flood the whole network, note that on all migrations shown on Figure 1 broker b_6 will not receive any notification. Algorithm 3 shows on lines 30-39 how a broker will handle a *BMIG* message. As seen on lines 32-35 the behavior is similar to a *MIG* message for subscriber

```
30 when receive(BMIG, C_j, b_j) from z ∈ N_i do
31 |  X ← ∅
32 |  foreach s ∈ C_j do
33 |  |  X ← X ∪ {b ∈ N_i where (_, b, s) ∈ R_i ∧ b ≠ z}
34 |  |  foreach (_, _, s) ∈ R_i do
35 |  |  ⌊  replace (_, _, s) with (_, z, s) in R_i
36 |  foreach y ∈ X do
37 |  ⌊  send(BMIG, C_j, b_j) to y
38 |  if z = b_j then
39 |  ⌊  send(BTAB, R_i) to b_j

40 when receive(BTAB, R_j) from b_j ∈ N_i do
41 |  foreach (_, _, s) ∈ R_i where s ∉ C_i do
42 |  ⌊  R_i ← R_i \ {(_, _, s)}
43 |  foreach (_, _, s) ∈ R_j where s ∉ C_i do
44 |  ⌊  R_i ← R_i ∪ {(_, b_j, s)}
45 |  foreach (f, _, s) ∈ (R_i − R_j) do
46 |  ⌊  send(SUB, f, s) to b_j
47 |  foreach (f, _, s) ∈ (R_j − R_i) do
48 |  ⌊  send(UNS, f, s) to b_j
49 |  foreach b ∈ N_i where b ≠ b_j do
50 |  ⌊  send(FMIG) to b

51 when receive(FMIG) from z ∈ N_i do
52 ⌊  send(BMIG, C_i, b_i) to z
```

Algorithm 3: Simple Routing with dynamic brokers

migration, we are just generalizing it for a set of subscribers, using a loop. It is important to save the value of the next hop on line 33 since we are changing its value on line 35. After replacing all required values the broker will send the same message it received to the previously stored brokers. Finally, on lines 38-39 we specify that if the broker from which the *BMIG* message has been received is the same as the originator of the message, meaning that the broker that received the message is actually b_j, the broker will reply with a *BTAB* message.

A *BTAB* message contains the routing table of the broker that sends the message. Since while a broker is migrating it might have lost all communication to the primary partition, it will not be able to receive any message that indicates a change in the subscriptions. This message is used to update the table on b_i to the latest version on the primary partition. Broker b_i knows the latest subscriptions from its local subscribers but it cannot trust any other entry on its routing table, for this reason once it receives a *BTAB* message it will remove

Table 1: Message types for a system with mobile clients and brokers

Message	Payload	Client/Broker	Meaning
SUB	$f \in F$	$s \in S$	Subscribe s to filter f
UNS	$f \in F$	$s \in S$	Unsubscribe s from filter f
PUB	$e \in E$	$p \in P$	Publish event e
MIG	—	$s \in S$	Notify the migration of s
FILTERS	$f : f \in F$	$b \in B$	Send active subscriptions
BMIG	C_b	$b \in B$	Notify the migration of b
BTAB	R_b	$b \in B$	Updated routing table of b
FMIG	—	$b \in B$	Force the migration of a broker

all entries from its table that do not belong to its local subscribers, lines 41-42, and it will store all entries, except the ones pertaining to b_i's subscribers, from the table it received, changing the next hop to the broker that sent the message, lines 43-44, so that any *PUB* message can be routed correctly. Then b_i will check to see what new information it has on its table with respect to b_j, lines 45-48, this way b_i will know if the subscriptions of its subscribers in the primary network are the correct ones, if there are any inconsistencies it will fix them by sending a *SUB* or *UNS* message for the corresponding subscription propagating it to the network using the standard protocol.

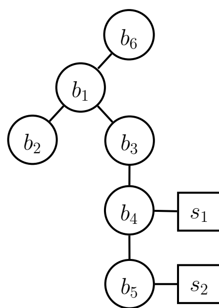
Until this point, the protocol works if a broker migrates alone, but if the broker that migrates has more brokers connected to it, b_i cannot know if there has been any change on that side of the network. For example if a subscriber migrates to the primary partition, b_i might not have received the corresponding *MIG* message. If we look at Figures 1c and 1d, both of the migrations, even though they are different, they are the same from the point of view of broker b_4. For modularity we have decided to force a sequential migration of all the brokers that are hanging from the migrating broker, this way each broker will update in the primary partition the information corresponding to its local subscribers. In Figures 1c and 1d the migration of b_4 will force the migration of b_5. This behavior can be seen on Algorithm 3, lines 49-50 where the message sent to all brokers still to b_i, and lines 51-52 where we show how to handle the message.

Table 1 summarizes all the message types used in the proposed protocol.

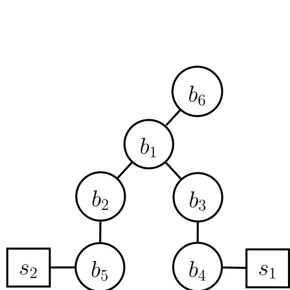
4 Conclusion

This work has presented an extension to the Phoenix framework that supports broker mobility, besides the client mobility already supported. This is done without the need to use a flooding mechanism for mobility messages.

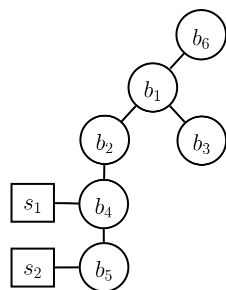
Further work is needed in order to solve possible mobility related inconsistencies on the network. Also an evaluation of this method using network simulation tools is required to asses the performance of the protocol.



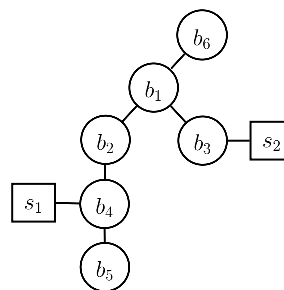
(a) Initial network



(b) b_5 migrates from b_4 to b_2



(c) b_4 migrates from b_3 to b_2



(d) b_4 migrates from b_3 to b_2 and s_2 migrates from b_5 to b_3

Figure 1: Network example and 3 posible migrations. Broker b_1 is the leader

References

[1] Guruduth Banavar, Tushar Deepak Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th International Conference on Distributed Computing Systems, Austin, TX, USA, May 31 - June 4, 1999*, pages 262–272. IEEE Computer Society, 1999.

[2] Ioana Burcea, Hans-Arno Jacobsen, Eyal de Lara, Vinod Muthusamy, and Milenko Petrovic. Disconnected operation in publish/subscribe middleware. In *5th IEEE International Conference on Mobile Data Management (MDM 2004), 19-22 January 2004, Berkeley, CA, USA*, page 39. IEEE Computer Society, 2004.

[3] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[4] Carlos Gómez-Calzado, Alberto Lafuente, Mikel Larrea, and Michel Raynal. Fault-tolerant leader election in mobile dynamic distributed systems. In *IEEE 19th Pacific Rim International Symposium on Dependable Computing, PRDC 2013, Vancouver, BC, Canada, December 2-4, 2013*, pages 78–87. IEEE, 2013.

[5] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile enviroment. In *Proceedings of the Second ACM International Workshop on Data Engineering for Wireless and Mobile Access, May 20, 2001, Santa Barbara, California, USA*, pages 27–34. ACM, 2001.

[6] Michael A. Jaeger, Helge Parzyjegla, Gero Mühl, and Klaus Herrmann. Self-organizing broker topologies for publish/subscribe systems. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*, pages 543–550. ACM, 2007.

[7] Luca Mottola, Gianpaolo Cugola, and Gian Pietro Picco. A self-repairing tree topology enabling content-based routing in mobile ad hoc networks. *IEEE Trans. Mob. Comput.*, 7(8):946–960, 2008.

[8] Gero Mühl. *Large-scale content based publish, subscribe systems.* PhD thesis, Darmstadt University of Technology, 2002.

[9] Gero Mühl, Andreas Ulbrich, Klaus Herrmann, and Torben Weis. Disseminating information to mobile clients using publish-subscribe. *IEEE Internet Computing*, 8(3):46–53, 2004.

[10] Vinod Muthusamy, Milenko Petrovic, and Hans-Arno Jacobsen. Effects of routing computations in content-based routing networks with mobile data sources. In Thomas F. La Porta, Christoph Lindemann, Elizabeth M. Belding-Royer, and Songwu Lu, editors, *Proceedings of the 11th Annual International Conference on Mobile Computing and Networking, MOBICOM*

2005, Cologne, Germany, August 28 - September 2, 2005, pages 103–116. ACM, 2005.

[11] David S. Rosenblum and Alexander L. Wolf. A design framework for internet-scale event observation and notification. In Mehdi Jazayeri and Helmut Schauer, editors, *Software Engineering - ESEC/FSE '97, 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering, Zurich, Switzerland, September 22-25, 1997, Proceedings*, volume 1301 of *Lecture Notes in Computer Science*, pages 344–360. Springer, 1997.

[12] Zigor Salvador, Alberto Lafuente, and Mikel Larrea. *Client Mobility Support and Communication Efficiency in Distributed Publish/Subscribe*. PhD thesis, University of the Basque Country, Spain, 2012.

[13] Zigor Salvador, Mikel Larrea, and Alberto Lafuente. Phoenix: A protocol for seamless client mobility in publish/subscribe. In *11th IEEE International Symposium on Network Computing and Applications, NCA 2012, Cambridge, MA, USA, August 23-25, 2012*, pages 111–120. IEEE Computer Society, 2012.

[14] Jan Hendrik Schönherr, Helge Parzyjegla, and Gero Mühl. Clustered publish/subscribe in wireless actuator and sensor networks. In Sotirios Terzis, editor, *Proceedings of the 6th International Workshop on Middleware for Pervasive and Ad-hoc Computing (MPAC 2008), held at the ACM/IFIP/USENIX 9th International Middleware Conference, December 1-5, 2008, Leuven, Belgium*, pages 60–65. ACM, 2008.

[15] Gerry Siegemund, Volker Turau, and Khaled Maamra. A self-stabilizing publish/subscribe middleware for wireless sensor networks. In *2015 International Conference and Workshops on Networked Systems, NetSys 2015, Cottbus, Germany, March 9-12, 2015*, pages 1–8. IEEE, 2015.