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 ∈ Iᵢ do
2     Rᵢ ← Rᵢ ∪ {(f, z)}
3     foreach b ∈ Nᵢ where b ≠ z do
4         send(SUB, f) to b
5 when receive(UNS, f) from z ∈ Iᵢ do
6     Rᵢ ← Rᵢ \ {(f, z)}
7     foreach b ∈ Nᵢ where b ≠ z do
8         send(UNS, f) to b
9 when receive(PUB, e) from z ∈ Iᵢ do
10    X ← ∅
11    foreach (f, y) ∈ Rᵢ 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 bᵢ)

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
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. With 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
Algorithm 3: Simple Routing with dynamic brokers

when receive(BMIG, C_j, b_j) from z ∈ N_i do
    X ← ∅
    foreach s ∈ C_j do
        X ← X ∪ {b ∈ N_i where (∊, b, s) ∈ R_i ∧ b ≠ z}
        foreach (∊, s) ∈ R_i do
            replace (∊, s) with (∊, z, s) in R_i
    foreach y ∈ X do
        send(BMIG, C_j, b_j) to y
    if z = b_j then
        send(BTAB, R_i) to b_j

when receive(BTAB, R_j) from b_j ∈ N_i do
    foreach (∊, s) ∈ R_i where s /∈ C_i do
        R_i ← R_i \ {(∊, s)}
    foreach (∊, s) ∈ R_j where s /∈ C_i do
        R_i ← R_i ∪ {(∊, b_j, s)}
    foreach (f, s) ∈ (R_i - R_j) do
        send(SUB, f, s) to b_j
    foreach (f, s) ∈ (R_j - R_i) do
        send(UNS, f, s) to b_j
    foreach b ∈ N_i where b ≠ b_j do
        send(FMIG) to b

when receive(FMIG) from z ∈ N_i do
    send(BMIG, C_i, b_i) to z
Table 1: Message types for a system with mobile clients and brokers

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

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 assess the performance of the protocol.
Figure 1: Network example and 3 possible migrations. Broker $b_1$ is the leader
References


