Department of Computer Architecture and Technology
Konputagailuen Arkitektura eta Teknologia saila (KAT)
Departamento de Arquitectura y Tecnología de Computadores
(ATC)

eman ta zabal zazu

Universidad del País Vasco Euskal Herriko Unibertsitatea

University of the Basque Country UPV/EHU

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Towards a Fully Mobile Publish/Subscribe System

Ph.D. Dissertation presented by
Ugaitz Amozarrain

Supervised by
Mikel Larrea

Donostia-San Sebastián, 2021

Science is not about building a body of known "fact". It is a method for asking awkward questions and subjecting them to a reality-check, thus avoiding the human tendency to believe whatever makes us feel good.

Terry Pratchett

Abstract

This PhD thesis makes contributions to support mobility and fault tolerance in a publish/subscribe system. Two protocols are proposed in order to support mobility of all devices in the system, including inside the event notification service. The protocols are designed with the idea that any change due to mobility is completely beyond our control and ability to predict. Moreover, the proposed solutions do not need to know neither the amount of nodes in the system nor their identities before starting, the system is able to adapt to new devices or disconnections and is able to keep operating correctly in a partitioned network. To do so we extend a previously proposed framework called Phoenix that already supported client mobility. Both protocols use a leader election mechanism to create a communication tree in a highly dynamic environment, and use a characteristic of that algorithm to detect topology changes and migrate nodes accordingly. Thus, our first approach was developed with the idea of making the fewest amount of changes to Phoenix while trying to maintain its main benefits. The resulting solution uses a recursive approach to notify migrations and update routing tables where each broker is responsible only for the subscribers that are directly connected to it. The second protocol, that we have decided to call MFT-PubSub, improves on the first one, introducing timestamps and consequently reducing the amount of messages to reconfigure the routing tables after a migration. Additionally, the message size is also reduced, and we also prevent possible inconsistencies on partitioned networks. Finally, we prove the validity of MFT-PubSub by simulating it with the Castalia framework for wireless sensor networks and compare it to AODV in terms of performance.

Acknowledgements

I would like to thank everybody that helped me these last five years and made this thesis possible.

First of all, I would like to express my gratitude to my supervisor, Mikel Larrea, for his guidance during this arduous task.

My parents and my sister, who, without fully understanding what I was getting into, decided to help and support me.

To my friends, with whom I could simply have fun and unwind from work or discuss my research topic, often ending up with new suggestions to try.

I am grateful to all the people on the third floor of the faculty of computer science. People that were in the same situation as me, and with whom I could talk about the day to day life of a PhD student.

Finally, I would also like to thank the people that work at LIP6 at Sorbonne University who helped me during my research stay in Paris.

This work has been funded by the following units: by the Spanish Research Council (MINECO), grants TIN2013-41123-P (DECIDE) and TIN2016-79897-P (BFT-DYNASTIE), and by the by the research group ADIAN that is supported by the Department of Education, Universities and Research of the Basque Government, grant IT980-16.

Contents

List of Algorithms

List of Algorithms

List of Figures

Chapter 1

Introduction

Distributed systems are commonplace nowadays. Everyone has used the Internet. We use it every day for a multitude of purposes, from navigating the World Wide Web to using messaging applications on our phones. We also create smaller networks in our homes with smart devices that have the ability to turn lights on or off with a spoken command. The ability to do all that is thanks to devices that are configured in such a way that they are able to intercommunicate using standard protocols. Moreover, for this communication, they can use wired or wireless connections to talk to each other. These collections of devices that work towards a common goal are what we call distributed systems.

There are several ways to classify these systems, we could organize them by how they see the passage of time. In a *synchronous* system all the devices that form part of it are completely synchronized. If the clock in one device perceives 10 seconds passing by, then we can assure that the clocks in the other ones have also measured 10 seconds. This is often used to simplify algorithms since it makes them easier to understand. But if we look at how computing really behaves we see that it often does so *asynchronously.* The clock in a device might be running slightly faster than the others, or a device might get stuck waiting for a process to end. The asynchronous model is closer to a real system than the synchronous one. Thought often, a barrier or other kind of synchronization mechanism can be used in order to force an asynchronous system to behave in the way of a synchronous one.

Another classification is by how tightly coupled the devices in a distributed network are. The client/server model that is most commonly found is a tightly coupled distributed system. A peer to peer network on the other hand is a loosely coupled system. The coupling of a system informs us about the difficulty of changing a device. The less coupled the system, the less a device needs to know about the other devices it interacts with.

1.1 Introduction to Publish/Subscribe systems

The publish/subscribe communication paradigm provides a mechanism for anonymous and loosely coupled communications between event producers and interested subscribers [24]. It was initially used in large scale systems e.g., the Internet [14, 16, 42]. Thought it can be seen as a messaging solution for separate devices connected to a network it can also be used to better structure the communication between any group of processes. It has also been used in wireless sensor networks [7, 11, 51] and the Internet of Things [2, 28, 29].

The main idea behind the publish/subscribe paradigm is to separate the devices that generate content from those that consume it. The content generated can range from a temperature reading of a sensor to an access notification on a web page, or even the distribution of a live television broadcast through the internet. In a publish/subscribe system the processes that generate and send content to the network are called the *publishers*, and those that consume the events are called subscribers. The decoupling is complete between both such processes. A publisher does not need to know which is the subscriber that is receiving the information it is sending to, nor do both of them need to be communicating at the same time in order for the message exchange to happen. There is an event notification service that receives the messages that the publisher sends and routes them to the subscribers that are interested in that information. In Figure 1.1 we see a representation of how such a service would look like. The notification service handles the event delivery without the publishers or subscribers knowing how the message has been routed inside it.

The event notification services are usually classified into two main groups, depending on how a subscriber specifies the events they are interested in.

Topic-based publish/subscribe: The oldest scheme. As its name implies it is based on topics. The publisher will send events belonging to a topic defined using keywords. The subscriber will register an interest in different topics and the event notification service will route those events matching the specified topic to the subscriber. It was created following

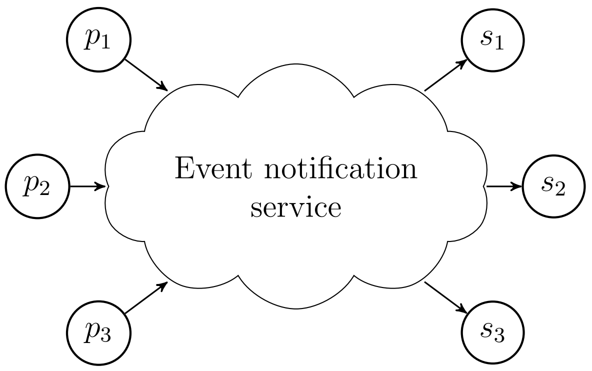


Figure 1.1: Representation of a publish/subscribe sytem.

the notion of groups, and a participant producing or consuming event from the same topic could be defined as belonging to the same group. An example of this would be for the subscriber to ask to receive all the events relating to a temperature sensor.

Content-based publish/subscribe: The topic based schema can be a little constraining in some cases. In content-based publish/subscribe events are not classified simply by a topic name. Event categorization is done using the content of the event itself. Subscribers, instead of showing interest in a set of topics, define the events they are interested in using a filter. This filter is usually defined using a subscription language specific to the implementation, SQL or XPath could be used as this subscription language. The event notification service would then try to test a publication according to the filters it has from the subscribers, if any filter matches the publication, it is routed towards that subscriber.

1.1.1 Event notification service

The event notification service is the most complex part of a publish/subscribe system. It is responsible for correctly delivering the messages to the subscribers. The devices that constitute the event notification service are called *brokers*. A publish/subscribe system can be centralized or distributed depending on the amount of brokers it has.

A centralized topology is much simpler to develop and deploy, since they only require a single broker to constitute the event notification service. But this simplicity also comes with some drawbacks, for example they lack scalability due to the bottleneck a single device can cause. Moreover, a single broker redirecting all messages is also a single point of failure.

A distributed event notification service will have an arbitrary number of brokers working in concert to deliver messages. This distribution of brokers greatly enhances the resilience of the service. If one broker fails, a new route can be found through the remaining brokers, moreover the load of each broker is reduced since there is not a single one redirecting all the messages. One of the complications is that in a distributed publish/subscribe service the brokers have to organize themselves in order to be able to communicate correctly. This organization can be made by the creation of a communication tree, ring or any other mechanism commonly used for distributed systems.

1.2 Mobility in distributed systems

Mobility in a distributed system is quite a difficult topic. A network composed of fully mobile devices, each working independently and sometimes communicating with each other, without a central connection point is difficult to handle.

What we call mobility is not simply the physical change in location of a device. Let us consider a set of people with mobile phones, while these people are walking on the streets they are changing their physical location. However, they are probably connected to the same antenna that has a range of up to several kilometers. Even though they are moving, the logical topology of the network does not change. Consequently, what we refer to as mobility is related more to the change in network topology, which is usually caused by physical mobility.

The algorithms designed for distributed systems have been commonly developed with the idea of having static devices with known locations and with stable links, unless there is a failure. The addition of mobility causes the creation of more asynchronicity in the network [3].

Some algorithms try to solve the mobility problem by adding some kind of synchronicity mechanism. For example, a moving node could handle mobility gracefully by notifying the system of its movement before losing connection, sometimes even providing its future location. This way, the system can prepare for the migration of that node and no links are broken suddenly. Another approach is to be more reactive. They do not keep track of the location, or their links to neighboring devices, and do not notify of a change before it occurs.

However, note that as mentioned before, from the point of view of a device connected to a network, the disconnection caused by physical mobility is indifferent to a link failure. A link that was correctly working for two devices to communicate has dropped unexpectedly. If we are able to handle link failures we are already a step closer to supporting device mobility.

For this reason in this dissertation we consider mobility as something completely beyond our control. Devices will be physically moving and links will fail, causing a change in the logical topology of the network. It is the responsibility of the proposed protocol to correctly handle these cases by detecting a broken link and trying to find an alternate route, or waiting until the connection is available again.

1.3 Objective

The main objective of this dissertation is to create a protocol that handles full mobility of all participants in a publish/subscribe system. This means that not only publishers and subscribers are able to join or leave the system, brokers are also allowed this ability. Consequently, one of the main challenges is maintaining the communication topology between nodes. Furthermore, any change in the logical topology will also cause a change in the routing tables the brokers use to efficiently deliver messages. The objective is to design a system in which the publish/subscribe service is not interrupted due to the mobility of one of its devices. The service should keep working for the clients that are connected, and the mobility support has to be as transparent as possible for the clients.

This thesis analyzes the proposals in the literature that try to solve mobility, first only taking into account client mobility and latter also the mobility inside the event notification service. This thesis follows a proposal made in the research group for client mobility and extends it to also support broker mobility, while trying to maintain its main features.

1.4 Organization of this dissertation

This dissertation is organized as follows. This Chapter introduces some basic concepts and motivates the research work. Chapter 2 introduces the related work in the area and defines some ground rules for the research which has been carried out, explaining the previous work it is based on and defining the system model we use. Chapter 3 describes a first approach to the objective of a mobile publish/subscribe system. Chapter 4 introduces the main contribution of this dissertation that improves the protocol described on the previous chapter. Finally, Chapter 5 summarizes the results obtained and proposes possible future research lines.

Chapter 2

Background and related work

2.1 Related work

Most of the research done in publish/subscribe systems is centered on improving current solutions, be it the reliability of delivering an event [23], improving the performance or increasing the fault tolerance [56]. Some work tries to improve on a typical tree structure for event delivery. In [20] authors propose the creation of a tree for each topic a subscriber can subscribe to with the publisher being the root of the tree for optimal message delivery. In some cases, a communication tree might be too weak against node failure, the reconfiguration of the tree might be too costly. The authors of [43] propose using gossiping so that the system can keep working while the tree is being repaired due to a node failure.

Another topic is the support for mobility. Though there are various protocols for publish/subscribe middleware, few of them support mobility [48]. In [32], authors mention some possible solutions for mobility support in publish/subscribe. Strategies are suggested to extend existing solutions, both in centralized and decentralized networks. In the case of a mobile network, nodes will need to adapt to disconnections, partitions of the network or the merging of those partitions, and the storage of undelivered events. In [31], Huang and Garcia-Molina study the tree construction problem in wireless ad hoc publish/subscribe systems. They define the optimality of a publish/subscribe tree by developing a metric to evaluate its efficiency, and propose a greedy algorithm that builds the publish/subscribe tree in a fully distributed fashion. Several works also address the different factors that affect the performance of a system with mobile nodes [9, 39], mostly based on mobile clients. A proposal to create self-configurable and adaptive peer-to-peer architecture for implementing content-based publish/subscribe communications on top of structured overlay networks has also been made [5, 4].

Another possible solution to support mobility is the use of information-centric networks [21, 53, 55]. Since this kind of network supports mobility natively, authors propose exploiting this property instead of using traditional TCP/IP communications.

Internet of Things (IoT) and Wireless Sensor Networks (WSN) also constitute an area that is still pushing research towards new topics [35, 54]. A number of recent contributions have also been made in the area [13, 27, 22, 33]. Most of the approaches support mobility through the inclusion of gateway nodes and the separation of the publish/subscribe system from the WSN. The gateway nodes receive messages from any number of sensors and act as a publisher to the publish/subscribe system. This allows for the sensors to be mobile devices that send events to the gateway they are connected to, but does not fully compose a mobile publish/subscribe system.

In this section we will explore several solutions for mobility support in a publish/subscribe system. We have separated them into two groups: those that support publisher or subscriber mobility and those that support mobility inside the event delivery notification service.

2.1.1 Mobile clients

Most of the research carried out to support the mobility of nodes in a publish/subscribe system has been done with regard to supporting mobile clients, be they publishers or subscribers.

The first system to support client mobility was called JEDI [19], named for Java Event-Based Distributed Infrastructure. In JEDI a node must notify of its intention to migrate to the broker to which it is connected, before the migration happens. This is done by the use of a *moveOut* message in which the subscriber can also specify if the broker should store all events it has subscribed to that are received while it is disconnected. When the subscriber connects to another broker it will send a *moveIn* message that will start the reconfiguration of the network. The new event dispatcher will exchange the required information with the old broker for that subscriber, obtaining all the subscriptions and undelivered messages. The new broker will also need to communicate with its parents in the topological tree to change the delivery route for the subscriber.

SIENA [14, 13] was a system developed at the same time as JEDI that also allowed for client mobility. Similar to JEDI, it also requires explicit *moveOut* and *moveIn* messages. The mobility is handled by a mobility service that creates proxy nodes in charge of storing events for clients that are disconnected. SIENA uses flooding, that has been found to be excessive [50], in order to find the source and destination brokers for the migration of a subscriber.

The REBECA [37, 38, 25] publish/subscribe system was also extended to support client mobility. In this case the moving node does not need to send an explicit *moveOut* message, a broker will detect when one of its connected subscribers has disconnected. The broker will then create a virtual counterpart of the roaming subscriber, and once the client is at its new location it will be merged with the virtual representation. The migration is done by the brokers themselves without further interaction by the subscriber. But due to the nature of its advertisement semantics it does not support publisher mobility.

Mobile XSiena [45] is a publish/subscribe platform which seeks to extend the XSiena [34] content-based publish/subscribe system in order to support user mobility. The key mobility-related features of Mobile XSiena are mobile device integration, seamless networking, reconnection support, location-based matching, and persistent events. This was later integrated into the Phoenix framework [44, 46, 47]. Phoenix solves the two tasks that must occur when a subscriber migrates [32]: updating the routing tables of the corresponding brokers such that new events are properly routed, and delivering the events published during the migration. The framework does this in a communication-efficient manner, i.e., without flooding the network.

MQTT is a commonly used protocol that also has received improvements in order to support client mobility. Though MQTT offers the support for subscriber mobility by allowing a subscriber to be connected to a subset of brokers, creating backups in case of a link failure, it does not allow for network reconfiguration in the case of a new connection, the subscriber will have to issue the subscriptions again. In [36] authors extend the protocol to support publisher mobility, by detecting a disconnection in the publisher node, and storing undelivered messages while the system is reconfigured.

This approach guarantees the delivery order of the messages to be the same as that of the creation.

$\mathcal{PSVR}$ [49] is a routing algorithm for a publish/subscribe system in a WSN. Siegemund et. al. mention the cost of maintaining a communication overlay in a dynamic environment, that is often really high or is omitted [18, 17], where systems usually recreate the overlay completely. The proposed algorithm is designed for systems with highly dynamic subscribers and publishers. The middleware also provides the guaranteed delivery of all published messages to all subscribers and the correct handling of subscriptions and unsubscriptions.

2.1.2 Mobile brokers

The scenario of mobility inside the event notification service is the most difficult to handle [30]. In this case the algorithms need to be able to handle the migration of not only clients but also reconfiguration on the subscription delivery path. There are few solutions that support full mobility on publish/subscribe systems.

In [4] an extension to SIENA is introduced where a self-organizing algorithm executed by brokers will try to optimize message delivery. Mechanisms are introduced to allow the reconfiguration caused by changes in topology, mostly to minimize the notification cost, but that could also be a first step towards supporting mobile nodes. Though the complexity of the algorithm, together with the need for a human administrator in case of a broker failure during the topology change procedure, makes it unsuitable for a highly mobile environment where a broker might start the topology change, but be disconnected by the time it finishes.

EMMA [41] is an extension to MQTT that not only handles client and broker migration in a transparent way, it also uses its migration mechanism in order to optimize QoS. It uses a controller node that is constantly monitoring the network and is informed of any change in device connectivity. The controller will then try to optimize event delivery and issue migrations to both clients and brokers to load balance the system. The requirement of a device that needs to know the connectivity of each node in the system prevent this solution to be used in a fully mobile environment where it might sometimes be unreachable.

2.2 Model and definitions

In this section we will describe the nomenclature used in the rest of the dissertation. We have already mentioned that in a publish/subscribe system we might find two different components. *Clients* will produce and consume events while the *notification service* handles the subscriptions issued by the clients and assures the correct delivery of events to the interested clients.

We can further divide the clients into two subsets: *subscribers* that will register their interests and consume events, and *publishers* that will produce those events. We will use $s \in S$ to refer to a subscriber belonging to the set of subscribers S and $p \in P$ to refer to a publisher that belongs to the set of publishers P. Any clients in the system may behave as a subscriber, publisher or even both at the same time. We will also use the nomenclature $f \in F$ when referring to a *filter* that belongs to the set of filters F. Subscribers are able to emit subscriptions by sending the corresponding filter to the broker they are connected to.

The notification service is composed of a set of brokers which we will call B and refer to individually as $b \in B$. The brokers will be connected at the logical level by an acyclic graph or a spanning tree. The brokers are the ones responsible for storing the subscriptions issued by the subscribers and routing the published events to the matching subscribers. At any moment a broker will have a set of neighboring brokers, in the graph, that it can communicate with. We will refer to this set as N_i for broker b_i. A broker will also be able to communicate with clients that are connected to it. For this reason we will refer to the set of interfaces, be it other brokers or clients, that a broker b_i can communicate with at any moment as I_i. Each publisher or subscriber will be connected to just a single broker which will be used as the entry point to the event notification service. Brokers are able to be connected to as many neighbors as they deem necessary. If a device decides to take on the role of a broker and a subscriber or a publisher, the client role will be considered to be connected to the broker in the same device. Thereby allowing the broker role to be connected to different neighboring devices but with the client connected to a single broker. We will refer to each individual role taken by a device as a process.

All communications are by point-to-point message passing over FIFO channels. In a static approach these links will be defined at the creation of the system and will not change over time. Whereas, if the participants are mobile, the set of channels linking them, as well as the neighbor set evolves. For our use there is no need to have previous knowledge of the sets, i.e., initially each participant knows only itself, and the number of participants in each set might change as time passes. This means that sets P, S and B are dynamic and will change depending on participants joining or leaving the network.

We represent any message sent by a process in the system with the following tuple:

$$(message\_type, payload)$$

The *message_type* field will inform the receiver of the message how it has to handle the containing *payload*. The amount of data in the *payload* field is variable and depends on the *message_type*. Each message will also have a sender and a destination.

A process is able to send a message to any other connected process using a `send()` primitive. We will use the following expression to refer to a message sent to a process reachable by a broker b_i:

$$send(message\_type, payload)\ to\ b \in I_i$$

Whenever a process receives a message it will block the reception of further messages until it has processed the contents. These messages will be stored in a buffer and will not be ignored.

Every subscriber will have a set of active filters that brokers will know of, we will denote this as $F_s \in F$ for subscriber s. The filters have a mechanism to test the events propagated through the network to see if they need to be delivered to the subscriber that issued them.

$$f(e) \rightarrow \{TRUE \mid FALSE\}$$

2.3 Simple Routing

The Simple Routing [6] approach assumes a static system where brokers are connected in an acyclic graph, and clients are permanently bound to a single broker. This routing strategy is based on the propagation of subscription (SUB) and unsubscription (UNS) messages to all of the brokers in the system.

Table 2.1 shows the three types of messages used in the Simple Routing protocol, for subscribing to a filter, unsubscribing from a filter and publishing an event respectively.

The routing table R_i at every broker b_i contains, 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 $z \in S$). The routing table is updated each time a subscriber issues a SUB or UNS message.

Lets use the topology found in Figure 2.1 as an example. We have a set of three brokers $B = \{b_1, b_2, b_3\}$ and six subscribers $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$. Each of the subscribers has registered an interest in the following filters: $F_{s_1} = \{f_1\}$, $F_{s_2} = \{f_1, f_2\}$, $F_{s_3} = \{f_3\}$, $F_{s_4} = \{f_4, f_6\}$, $F_{s_5} = \{f_1\}$ and $F_{s_6} = \{f_6\}$. Each of the brokers will keep track of the subscriptions of each subscriber and the interface a message needs to be sent through in the case

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

Table 2.1: Simple Routing message description.

that the filter matches. In the case of b_2 receiving an event matching f_1 it will directly deliver it to s_2 and send it to b_1 so it can be routed towards s_1.

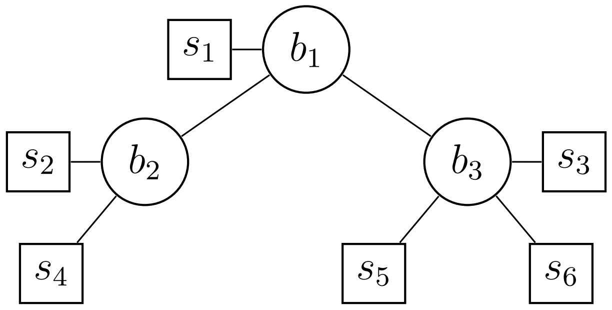


Figure 2.1: Sample publish/subscribe topology.

Filter	Interface
f_1	s_1
f_1	b_2
f_1	b_3
f_2	b_2
f_3	b_3
f_4	b_2
f_6	b_3
f_6	b_3

Table 2.2: Routing table for broker b_1 in a simple routing strategy.

In Table 2.2 we show the what the routing table of broker b_1 would look like. As mentioned previously this routing table contains the filters and the interface to send the event towards in the case that they match. We can also observe that the last two entries of the table have the same filter and the same interface. This is used in this example to show the subscriptions for all subscribers, though in a real implementation those two entries can be merged into one, optimizing the delivery.

2.3.1 Routing mechanism

In this section we will describe the routing mechanism used by simple routing. Algorithm 2.1 shows the code that is executed by every broker in the event notification service. This mechanism provides optimal routing treating event delivery as multicast messages. Events are transmitted through a link only once without duplication and subscribers will receive no duplicate messages. Using the example in Table 2.2 any event that matches f_6 will be routed towards b_3 in a single message and b_3 will deliver that event to the subscribers.

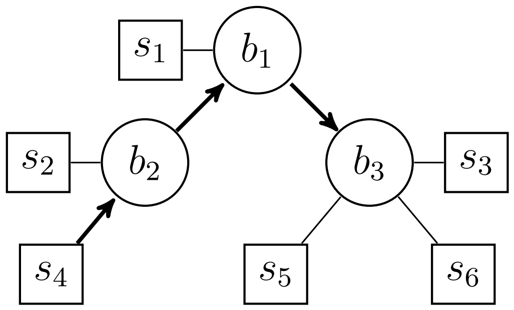
```
1  when receive(SUB, f) from $z \in I_i$ do
2      add one $(f, z)$ entry to $R_i$
3      foreach $b \in N_i$ where $b \neq z$ do
4          send(SUB, f) to $b$

5  when receive(UNS, f) from $z \in I_i$ do
6      remove one $(f, z)$ entry from $R_i$
7      foreach $b \in N_i$ where $b \neq z$ do
8          send(UNS, f) to $b$

9  when receive(PUB, e) from $z \in I_i$ do
10     $X \leftarrow \varnothing$
11     foreach $(f, y) \in R_i$ where $y \notin X \wedge y \neq z$ do
12         if $f(e) = true$ then
13             $X \leftarrow X \cup \{y\}$
14     foreach $y \in X$ do
15         send(PUB, e) to $y$
```

Algorithm 2.1: Simple Routing (code executed by broker b_i)

Whenever a broker receives a *SUB* message it does two tasks, lines 1-4. First it adds the subscription to the routing table, storing the filter, f, and the sender z. Then it propagates that message to all the brokers that belong to the set of neighboring processes, with the exception of the sender of the message. This way the subscription message is propagated to all the brokers in the network and since they are connected through an acyclic tree there will be a time when the last broker receives this message and has no one else to send it to. Note that as shown in Table 2.2, the algorithm does not check for duplicate entries in its routing table. If there already exists an entry with the same combination of (f, z), this entry will be duplicated. Figure 2.2 shows how a *PUB* message sent by subscriber s_4 reaches all the brokers in the network and updates the routing tables.

The *UNS* message, lines 5-8 is handled in a similar way to the *SUB* message. In this case, instead of adding the filter to the routing table we are removing it, since the subscriber has shown its interest in unsubscribing from the filter. The message is also propagated through the network in the same manner as the previous one, allowing all the brokers in the network to remove the subscription. In the case of duplicate entries only one is removed. Due to the decoupling of elements within the system we do not know which subscriber issued the subscription. If we were to optimize the routing table and only store unique combinations of (f, z), we would have a problem at this point. If we allowed no duplicate entries in the routing table, we would not know how many subscribers were interested in a given filter, and when we receive this *UNS* message we would not be able to remove it from the table. Allowing the duplicate entries, we can remove one of the entries at this time and we will still be able to keep routing messages to other subscribers interested in the same filter and connected through the same interface.

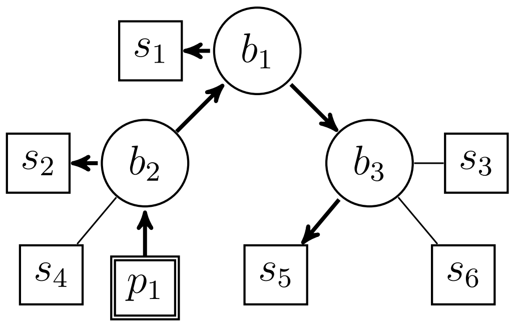
Lines 9-15 show the processing of the last message, *PUB*. The handling of this message is based on iterating through the routing table of the broker to find filters that match the event and their corresponding interfaces. Line 11 ensures that once an interface has been chosen it will not be tested again, since the message will already be propagated through that link. This interface matching is what makes it possible for simple routing to optimally deliver events. Publications towards different subscribers will be sent in a single message until they reach another broker that will have separate interface entries for the same filter. Figure 2.3 shows the path a *PUB* message sent



Filter	Interface
f_1	s_1
f_1	b_2
f_1	b_3
f_2	b_2
f_3	b_3
f_4	b_2
f_6	b_3
f_6	b_3
$\boldsymbol{f_7}$	$\boldsymbol{b_2}$

Figure 2.2: Subscriber s_4 issues a new subscription and the SUB message is propagated. Routing table of broker b_1 with the new entry on the right.

by a publisher connected to b_2 would follow. When this message reaches first b_2, and later b_1, in both cases the broker resends it to two of its interfaces, the first one is the subscriber that is directly connected to it, and the second is another broker.



Filter	Interface
f_1	s_1
f_1	b_2
f_1	b_3
f_2	b_2
f_3	b_3
f_4	b_2
f_6	b_3
f_6	b_3
f_7	b_2

Figure 2.3: Publisher p_1 sends a PUB message that matches filter f_1. Routing table of broker b_2 with the new entry on the right.

2.4 Phoenix

Phoenix [47] 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 simple routing brokers do not care where a published event is coming from. At any step of the delivery process the broker will check its routing table and send the message towards the interfaces that are related to the matching filters. For this reason and since brokers do not store any information on publishers, a publisher can easily change the router it is connected to.

Any kind of temporary loss of connectivity can be characterized as client mobility, not only those caused by physical mobility. A link failure in a static system could also be described as mobility, if the process, instead of recovering the lost link tries to reconnect to the network using a secondary link. When this happens the newly connected process might have to be informed of this change. But client mobility often happens as a result of signal degradation on a wireless network due to the increase of distance between two connected devices. In those cases the client is temporarily disconnected to the network in a manner that it is not able to fully predict and does not know when it will be connected again. In the case of subscriber mobility, a subscriber cannot be sure that in case of a disconnection it will be able to connect to the same broker again.

Phoenix tries to solve the problem of client mobility while trying to make this mobility as transparent to the subscribers as possible. Not only keeping the subscriptions the client already has on the system, but also storing undelivered events. This way, in the case of a temporary disconnection the subscriber will not lose any messages. In order to support subscriber mobility Phoenix requires a method to notify the broker network that it has changed the connection and that it is the same broker that was connected somewhere else before. The messages defined for simple routing are kept, but their contents are changed. Subscriber identity is added to *SUB* or *UNS* messages and this identity is also stored in the routing table of the brokers.

Filter	Interface	Subscriber
f_1	s_1	s_1
f_1	b_2	s_2
f_1	b_3	s_5
f_2	b_2	s_2
f_3	b_3	s_3
f_4	b_2	s_4
f_6	b_3	s_4
f_6	b_3	s_6

Table 2.3: Routing table of broker b_1 using Phoenix.

The modified routing table, with the subscriber identities added, is shown in Table 2.3 for comparison with that shown in Table 2.2 for the network in Figure 2.1. And in order to support sending undelivered events during client migration all messages will include a timestamp. This way subscribers can request the replaying of events that have happened after a specific time. The replaying of events also requires the creation of queues on the brokers, one for each subscriber they know of, in order to store messages that can be sent at a later date.

For the support of subscriber mobility, Phoenix introduces two new message types that can be seen in Table 2.4. The first, called *MIG*, is the one a subscriber will use in the case it detects a disconnection, or connects to a different broker. This *MIG* message will reach the last broker the subscriber is connected to following the same path a *PUB* message would follow for that same subscriber. The second message it adds is called *REP*, and is the one that is used to resend events that where lost during the disconnection.

Message	Payload	Client/Broker	Meaning
MIG	—	$s \in S$	Notify the migration of s
REP	$e \in E$	$s \in S$	Replay event e towards s

Table 2.4: Phoenix message description

Algorithm 2.2 shows the changes made to simple routing to begin supporting mobile subscribers. We can see how the messages also contain the subscriber identifier which is also added to the extended routing tables. Besides that, the broker also maintains a set of local clients C_i. This set of local clients is updated each time a broker receives a message from a subscriber, as seen in lines 5 and 12. The matching of events is similar to the one in simple routing. The main difference, in line 22, is that broker will now store all events that are directed to a subscriber belonging to its local set. Events are stored with a local timestamp by the broker. Since a subscriber receives all events from the same broker they can specify the replaying of messages after the last correct reception of a *PUB* message.

The main contribution of Phoenix is shown in Algorithm 2.3. Here the subscriber migration process is described. There are two main parts in the handling of the *MIG* message. On the one hand we have the brokers that receive this message without the subscriber being in their local sets. In this case the broker will check if the subscriber is now local and will redirect the message towards the broker that was last connected to that subscriber. Using the routing table we have information not only about the filters and interfaces they belong to, but also about subscribers that issued those same filters. Searching the routing table, as seen in line 5, we can obtain the interface from which the *SUB* message of that subscriber was received. Following this path at each broker we reach the one that was connected directly to the broker. Brokers will have to update their routing tables to point towards the new and correct interface to route messages.

On the other hand, if the broker that receives such a message has the subscriber in its local set, it means that the subscriber has moved and it is no longer local. In this case the message does not need to be redirected anymore, but the broker has other duties. It needs to send back the events that the subscriber has requested.

Lines 17-24 show how a replayed message is routed. The message is routed in a similar way to a published event, but instead of matching the filter, the broker simply matches the subscriber in the routing table. The last broker on the delivery process will be the one that has this subscriber in its local set, and has to store the messages in a queue again, in case the subscriber has migrated during this process.

```
when receive(SUB, f, s) from $z \in I_i$ do
    if $\nexists(f, \_, s) \in R_i$ then
        $R_i \leftarrow R_i \cup \{(f, z, s)\}$
    if $z \notin N_i \wedge s \notin C_i$ then
        $C_i \leftarrow C_i \cup \{s\}$
    foreach $b \in N_i$ where $b \neq z$ do
        send(SUB, f, s) to $b$

when receive(UNS, f, s) from $z \in I_i$ do
    if $\exists(f, \_, s) \in R_i$ then
        $R_i \leftarrow R_i \setminus \{(f, \_, s, t)\}$
    if $z \notin N_i \wedge s \nexists(-, -, s) \in R_i$ then
        $C_i \leftarrow C_i \setminus \{s\}$
    foreach $b \in N_i$ where $b \neq z$ do
        send(UNS, f, s) to $b$

when receive(PUB, e, ts) from $z \in I_i$ do
    $ts \leftarrow$ clock()
    $X \leftarrow \varnothing$
    foreach $(f, y, \_) \in R_i$ where $y \notin X \wedge y \neq z$ do
        if $f(e) = true$ then
            $X \leftarrow X \cup \{y\}$
            if $y \in C_i$ then
                enqueue(e, ts) in $Q_i(s)$
    foreach $y \in X$ do
        send(PUB, e, ts) to $y$
```

Algorithm 2.2: Simple Routing with dynamic clients

```
when receive(MIG, s, b, ts_last) from $z \in I_i$ do
    if $s \notin C_i$ then
        if $z = s$ then
            $C_i \leftarrow c_i \cup \{s\}$
        $b_j \leftarrow b \in N_i$ where $(-, b, s) \in R_i$
        foreach $(-, -, s) \in R_i$ do
            replace $(\_, \_, s)$ with $(\_, z, s)$ in $R_i$
        send(MIG, s, b, ts_last) to $b_j$
    else
        $C_i \leftarrow C_i \setminus \{s\}$
        foreach $(-, , -, s) \in R_i$ do
            replace $(\_, \_, s)$ with $(\_, z, s)$ in $R_i$
        while $Q_i(s)$ is not empty do
            dequeue(e, ts) from $Q_i(s)$
            if $ts > ts_{last}$ then
                send(REP, e, s, _) to $z$

when receive(REP, e, s, ts) from $z \in I_i$ do
    if $s \notin C_i$ then
        $b_j \leftarrow b \in N_i$ where $(-, b, s) \in R_i$
        send(REP, e, s) to $b_j$
    else
        $ts \leftarrow$ clock()
        enqueue(e, ts) in $Q_i(s)$
        send(REP, e, ts) to $s$
```

Algorithm 2.3: Client mobility in Phoenix

2.5 Network topology

Unlike Phoenix in which the broker topology was static and predefined before starting the publish/subscribe service, this dissertation introduces a fully mobile protocol. Before handling the communication required to support that service, we must create a network topology with the nodes available to us. Being fully mobile we have a variable amount of nodes on the system, that will change their neighbors as time passes, so we cannot preconfigure the network.

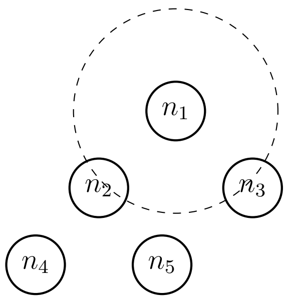
In order for the devices on the network to communicate efficiently we must create a logical overlay. One of the most common examples is the creation of an acyclic graph also called a spanning tree that helps in correctly routing messages towards their recipient. These spanning trees are usually created using information about the network, be it the number of nodes connected or cost of sending a message through one link. On a static network we can easily obtain that information, and once we have it in our hands we are able to calculate a single graph for optimum connectivity which requires few reconfigurations. However when we add mobility into the mix we have an increased number of lost links between two nodes. And each time we have to notify the devices on the network of the need to reconfigure. For this reason we have a need for a protocol that inherently supports mobility in the creation of the network topology.

We also need a mechanism that detects when a change in the topology has occurred so a new link will be created when an old one disappears, so we can react accordingly. Another problem to take into account is that due to the mobility of the nodes we might end up with a partitioned network. Whenever this happens we have to keep providing a service to all the nodes on all the partitions. Lastly we want to minimize the number of nodes in the network that needs to be reconfigured whenever a partition occurs. This way we reduce the number of migrations that might take place, reducing the number of messages sent, and reducing the load on the network.

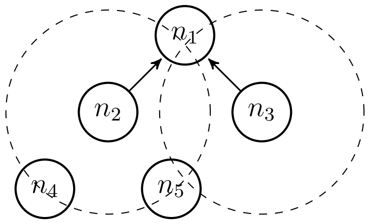
In our case, we have chosen a leader election algorithm that has a heartbeat mechanism in order to keep the leader stable [26]. Once a leader has been elected, this node will keep sending messages so that all the other nodes will have this one as their leader. When a node receives one of these messages it will know the path to the leader [10], and it will broadcast it so the message spreads to all nodes within communication range. With this we create the overlay we need for constructing the publish/subscribe system.

In Figure 2.4 we show how this heartbeat mechanism works. Suppose a network of five nodes where n_1 has been chosen as the leader by the algorithm, the leader will keep sending a heartbeat mechanism so the rest of the nodes know it is still alive and reachable. Using this message the rest of the nodes will know the path towards the leader. If each time this message is reset we store the link used to receive the message in the node we can point towards the leader as seen in Figures 2.4b and 2.4c. Each node will only resend the heartbeat message once for each round and only store the path towards the leader on the first reception of the message. After the round finishes we are left with a connectivity graph that does not have any cycles as shown in Figure 2.4d.

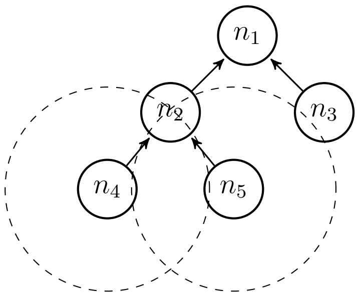
Using this algorithm, in the event that the network is partitioned, each of the partitions will choose a leader. And eventually when the network becomes connected again both partitions will merge, choosing a single leader and maintaining a single graph. Furthermore, with this heartbeat message, when a node first receives the message of a new round it will store the sender as the next hop to the leader. This next hop might be modified by any physical change in the location of a node or by a failure, since the heartbeat message will arrive via another node. Using this we can detect when the topology has changed, whenever the next hop to the leader changes, and notify the publish/subscribe system so that it can migrate accordingly.



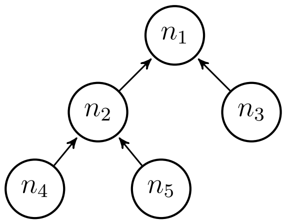
(a) n_1 sends heartbeat message.



(b) n_2 and n_3 resend the heartbeat.



(c) n_4 and n_5 resend the heartbeat.



(d) The connection tree is created.

Figure 2.4: Hearbeat mechanism of the leader election used to create an acyclic connection tree.

Chapter 3

First approach to mobile Publish/Subscribe

3.1 Introduction

This is the first approach to supporting full mobility on publish/subscribe. It follows a simple and modular approach to handling node migration. Whenever a node creates a change in the network topology, be it from physically moving from one place to another, or due to losing a connection, it causes a migration to happen in the pub/sub service. This migration can range from a simple one, a publisher changing its connection to a broker losing a link and migrating. In the first case we do not need to update the network at all; the base protocol is able to handle that on its own. For the second case we need to update the routing tables for the subscribers attached to that broker. Depending on the complexity of the network, several migrations can happen at the same time, complicating things further.

For this first approach whenever a broker migrates it will only handle the migration for the subscribers that are directly attached to it. Any other subscriber is left as it was before the migration and the rest of the brokers are notified that they also need to migrate. Creating a cascading, or recursive migration until all the brokers involved have finished updating their routing tables.

3.2 Messages

Table 3.1 shows the additional messages used in order to support broker mobility, for sending the set of filters a subscriber has issued, for notifying the migration of a broker, for updating the routing table based on the information of the primary partition (i.e., the partition containing the leader), and for forcing the migration of a broker respectively.

We have created four new messages for the nodes to support broker mobility, and to be able to communicate information referring to a migration in the network. The definition for these messages can be seen in Table 3.1. These new messages are used together with the previous ones, defined on Section 2.4 for the Phoenix protocol.

Message	Payload	Client/Broker	Meaning
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
FILTERS	$f : f \in F$	$b \in B$	Send active subscriptions

Table 3.1: Broker mobility message description

The first message, which we call *BMIG*, is the one used to notify the migration of a broker. This is the message that starts the migration process and contains information corresponding to the subscribers that are directly connected to the broker that sends it. This message is routed using the routing path for the subscribers it contains until it reaches the last broker that was connected to the migrating broker. The *BTAB* message contains the full routing table of the broker that sends it, and is used to send updated information to the brokers involved in a migration. As mentioned earlier we create a cascading migration with the use of the *FMIG* message that forces any broker that receives it to send a *BMIG* message to the sender. Finally due to inconsistencies that might happen in a partitioned network we use the *FILTERS* message to notify a broker of all the active subscriptions of a given subscriber.

In order to further explain this *FILTERS* message, let us imagine a network with two different partitions (p_1 and p_2). A subscriber (s) might connect to p_1 and subscribe to whatever topics it desires, but after some time it loses connection with the brokers on p_1 and connects to a broker on p_2. Since the network is partitioned, the brokers on p_2 have no information on the subscriptions of s. This can be complicated further since s might be jumping between p_1 and p_2, and subscribing to different topics in each one. In this case whenever s migrates, the first broker that receives the migration message will answer back with a *FILTERS* message, and the subscriber can fix inconsistencies by sending *SUB* or *UNS* messages.

To summarize the migration process, whenever a broker detects a migration, it will send a *BMIG* to its new parent in the topology, and receive a *BTAB* message from it. The migrating broker will also send *FMIG* messages to all its connected brokers causing a cascade of migrations.

3.3 The protocol

This section contains an in depth explanation of this first approach. For this version of the protocol we have simplified the code of Phoenix, for example by removing the timestamp values from messages. In Phoenix, brokers store undelivered *PUB* messages, and when a subscriber migrates it can request for the events that it has lost to be resent by specifying the timestamp of the last received message. When the broker to which it was connected previously receives this message, it will resend all the messages that are newer than that specified timestamp. But, in our case, the subscribers are not the only ones that are migrating, brokers will also migrate. As a broker migrates it has no knowledge of the last received message by a subscriber. Furthermore since the broker network is also changing, we cannot designate a single broker as the one responsible for storing the events. We can have a situation where the broker that stores the messages for a subscriber is in a different partition from that subscriber. For this reason we do away with the set of local subscribers that Phoenix uses to handle subscriber migration. In the case that we need to know if a subscriber is local for a broker, we can simply look at the routing table. Remember that in the routing table we store a tuple containing the filter, subscriber and interface. The interface in this case is designated as the identifier of the device that is the next hop towards the subscriber. If the interface has the same identifier as the subscriber, we know that the subscriber is one hop away and that it is local to the broker.

For this reason in this protocol, brokers will store all undelivered messages to be resent at a later date. We need a mechanism that tells us if a message has been delivered. A simple acknowledgment by the subscriber whenever it receives a message is enough. With this, if an error occurs, the broker will store the message as undelivered. When a broker receives a migration message, from a subscriber or another broker, it will send all messages stored for the subscribers that migrate. These changes are reflected in Algorithms 3.1 and 3.2. Due to the way migrations can occur in the system we cannot be sure that a subscriber has received all the undelivered messages whenever it migrates. If the broker that contains some of those messages is not connected to the same partition as the subscriber, the subscriber will have to wait until another migration occurs and they end up in the same partition.

```
when receive(SUB, f, s) from $z \in I_i$ do
    if $\nexists (f, \_, s) \in R_i$ then
        $R_i \leftarrow R_i \cup \{(f, z, s)\}$
    foreach $b \in N_i$ where $b \neq z$ do
        send(SUB, f, s) to $b$

when receive(UNS, f, s) from $z \in I_i$ do
    if $\exists (f, \_, s) \in R_i$ then
        $R_i \leftarrow R_i \setminus \{(f, \_, s)\}$
    foreach $b \in N_i$ where $b \neq z$ do
        send(UNS, f, s) to $b$

when receive(PUB, e) from $z \in I_i$ do
    $X \leftarrow \varnothing$
    foreach $(f, y, \_) \in R_i$ where $y \notin X \wedge y \neq z$ do
        if $f(e) = true$ then
            $X \leftarrow X \cup \{y\}$
    foreach $y \in X$ do
        if $y \notin I_i$ then
            foreach $(f, y, s) \in R_i$ where $f(e) = true$ do
                enqueue $\{e\}$ in $Q_i(s)$
        else
            send(PUB, e) to $y$
```

Algorithm 3.1: First approach to mobile clients

```
when receive(MIG, s, b) from $z \in I_i$ do
    if $z = s$ then
        $X \leftarrow \varnothing$
        foreach $(f, \_, s) \in R_i$ do
            $X \leftarrow X \cup \{f\}$
        send(FILTERS, X) to $s$
    if $b \neq b_i$ then
        if $\exists(\_, \_, s) \in R_i$ then
            $b_j \leftarrow y \in N_i$ where $(\_, y, s) \in R_i$
            send(MIG, s, b) to $b_j$
    foreach $(\_, \_, s) \in R_i$ do
        replace $(\_, \_, s)$ with $(\_, z, s)$ in $R_i$
    sendQueuedMessages($s, z$)

function sendQueuedMessages($s$, $y$)
    while $Q_i(s)$ is not empty do
        dequeue $\{e\}$ from $Q_i(s)$
        if $y \notin I_i$ then
            enqueue $\{e\}$ in $Q_i(s)$
            return
        else
            send(REP, e, s) to $y$

when receive(REP, e, s) from $z \in I_i$ do
    $y \leftarrow x \in I_i$ where $(\_, x, s) \in R_i$
    if $y \notin I_i$ then
        enqueue $\{e\}$ in $Q_i(s)$
    else
        send(REP, e, s) to $y$
```

Algorithm 3.2: Simple Routing with mobile clients - Message replay

To complicate things more, as explained in Section 2.5, we may have a subscriber migrating from one partition of the network to another, and since both partitions function individually the subscriber will have different subscriptions in each of them. The $FILTERS$ message is designed to fix this issue. We can see how this message is sent in lines 23-27 of Algorithm 3.1. Whenever a subscribers sends a MIG message, the broker it migrates to will answer with a $FILTERS$ message. This message contains all the subscriptions of that subscriber the broker has in its routing table. Using this information the subscriber may decide that the subscriptions are outdated and issue SUB or UNS messages to fix and update the routing tables of the brokers on that partition. These SUB and UNS messages will be propagated normally to the rest of the brokers.

The main function responsible for how brokers handle the migration is shown in Algorithm 3.3. Whenever a broker migrates it will send a $BMIG$ message with two parameters: a list of the subscribers connected to it (C_j) and its own identifier (b_j). Any broker that receives this message will first replace the next hop of those subscribers in the routing table to the sender of the message, while storing the old value. The it will resend the $BMIG$ message to the stored values so that they are notified of the change. This behavior can be seen in lines 2-8 in Algorithm 3.3 and it is similar to what happens when a subscriber migrates, but in this case the migration happens by proxy, using the broker. Then, if the broker is the first one to receive the message it will answer with a $BTAB$ message containing its whole routing table. Finally the broker will send any stored messages for those subscribers. The process is the same as if those subscribers were to individually migrate, but the broker is able to group the migrations together.

Whenever a broker migrates it can only trust the subscribers that are connected directly. A subscriber that is lower on the network topology might have migrated and the broker has not yet received that information, or as explained before that subscriber might have connected to another partition. For this reason when, after sending a $BMIG$ message, it receives a $BTAB$ message back, it will then empty its routing table except for the entries of the local subscribers. Subsequently it will store all the information that comes with the message as shown in lines 14-17. In lines 18-21 the broker checks for any information that is missing on the other broker's routing table and sends the necessary SUB and UNS messages to fix it as if it were a subscriber that received a $FILTERS$ message. After this, the broker will force the migration of all the brokers that are connected to it with an $FMIG$ message. As soon as a broker receives an $FMIG$ message it will answer back with a $BMIG$ message starting the process again. And finally, as before, the broker will send any stored messages for all subscribers.

```
when receive(BMIG, $C_j$, $b_j$) from $z \in N_i$ do
    $X \leftarrow \varnothing$
    foreach $s \in C_j$ do
        $X \leftarrow X \cup \{b \in N_i$ where $(\_, b, s) \in R_i \wedge b \neq z\}$
        foreach $(\_, \_, s) \in R_i$ do
            replace $(\_, \_, s)$ with $(\_, z, s)$ in $R_i$
    foreach $y \in X$ do
        send(BMIG, $C_j$, $b_j$) to $y$
    if $z = b_j$ then
        send(BTAB, $R_i$) to $b_j$
    foreach $s \in C_j$ do
        sendQueuedMessages $(s, z)$

when receive(BTAB, $R_j$) from $b_j \in N_i$ do
    foreach $(\_, \_, s) \in R_i$ where $s \notin C_i$ do
        $R_i \leftarrow R_i \setminus \{(\_, \_, s)\}$
    foreach $(\_, \_, s) \in R_j$ where $s \notin C_i$ do
        $R_i \leftarrow R_i \cup \{(\_, b_j, s)\}$
    foreach $(f, \_, s) \in (R_i - R_j)$ do
        send(SUB, f, s) to $b_j$
    foreach $(f, \_, s) \in (R_j - R_i)$ do
        send(UNS, f, s) to $b_j$
    foreach $b \in N_i$ where $b \neq b_j$ do
        send(FMIG) to $b$
    foreach $(\_, \_, s) \in R_j$ where $s \notin C_i$ do
        sendQueuedMessages $(s, b_j)$

when receive(FMIG) from $z \in N_i$ do
    send(BMIG, $C_i$, $b_i$) to $z$
```

Algorithm 3.3: Simple Routing with mobile brokers

3.4 Migration example

Let us analyze the protocol with two examples of migrations that might occur. The first one, shown in Figure 3.1 shows a straightforward migration. Due to physical mobility or a link failure a connection has been lost, and the broker decides to connect to a different one.

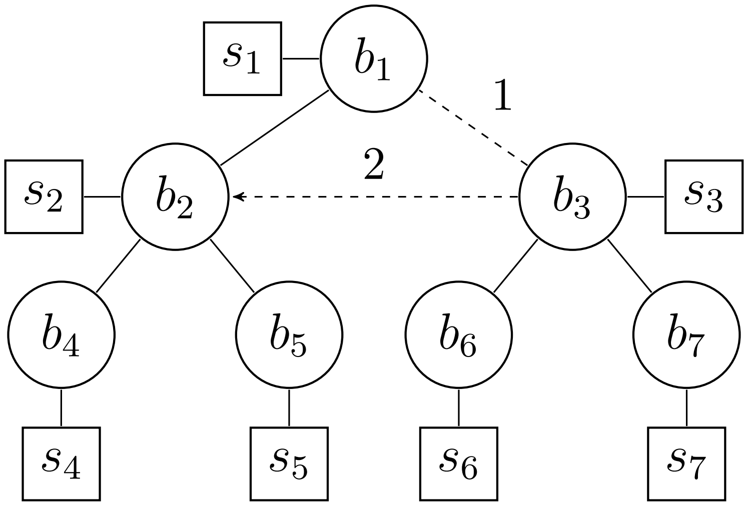


Figure 3.1: Simple migration example. $b_3 \leftrightarrow b_1$ link is lost and $b_3 \leftrightarrow b_2$ is created.

If we were to look at the messages exchanged between the brokers we would notice that b_3 will start with a *BMIG* message. And b_3, b_6 and b_7 will have to somehow inform $b1$ of the new location of their subscribers. The following list enumerates the sequence of messages sent due to that migration:

1. $b_3 \rightarrow b_2 \Rightarrow BMIG : \{C_j = [s_3], b_j = b_3\}$
2. $b_2 \rightarrow b_3 \Rightarrow BTAB : \{R_j = R_2\}$
3. $b_2 \rightarrow b_1 \Rightarrow BMIG : \{C_j = [s_3], b_j = b_3\}$
4. $b_3 \rightarrow b_6 \Rightarrow FMIG$
5. $b_3 \rightarrow b_7 \Rightarrow FMIG$
6. $b_6 \rightarrow b_3 \Rightarrow BMIG : \{C_j = [s_6], b_j = b_6\}$

7. $b_7 \to b_3 \Rightarrow BMIG : \{C_j = [s_7], b_j = b_7\}$
8. $b_3 \to b_6 \Rightarrow BTAB : \{R_j = R_3\}$
9. $b_3 \to b_7 \Rightarrow BTAB : \{R_j = R_3\}$
10. $b_3 \to b_2 \Rightarrow BMIG : \{C_j = [s_6], b_j = b_6\}$
11. $b_3 \to b_2 \Rightarrow BMIG : \{C_j = [s_7], b_j = b_7\}$
12. $b_2 \to b_1 \Rightarrow BMIG : \{C_j = [s_6], b_j = b_6\}$
13. $b_2 \to b_1 \Rightarrow BMIG : \{C_j = [s_7], b_j = b_7\}$

Even thought the change seems small, only one link changes and no other change has been made in the network, the protocol forces the migration of all the brokers that are below the migrating one. This causes the cascade of *BMIG* and *BTAB* messages we see in the list. We can further complicate this example by adding another migration at the same time as shown in Figure 3.2. In this case b_3 loses its link to b_1, and while that link is down b_6 decides to migrate to b_5.

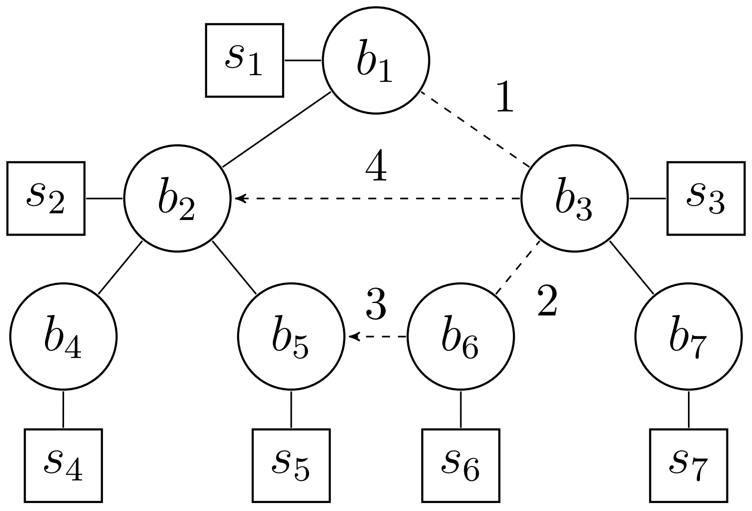


Figure 3.2: Complex migration example. First $b_3 \leftrightarrow b_1$ link is lost and after $b_6 \leftrightarrow b_3$ is also lost. $b_6 \leftrightarrow b_5$ is connected and allowed to completelly migrate before $b_3 \leftrightarrow b_2$ is created.

The sequence of messages in this example is similar to the previous one. First we have the migration of b_6 and then b_3. b_6 has no broker connected to it during the migration, for this reason it does not need to force migration. b_3, on the other hand will force the migrations as in the previous example.

1. $b_6 \rightarrow b_5 \Rightarrow BMIG : \{C_j = [s_6], b_j = b_6\}$
2. $b_5 \rightarrow b_6 \Rightarrow BTAB : \{R_j = R_5\}$
3. $b_5 \rightarrow b_2 \Rightarrow BMIG : \{C_j = [s_6], b_j = b_6\}$
4. $b_2 \rightarrow b_1 \Rightarrow BMIG : \{C_j = [s_6], b_j = b_6\}$
5. $b_3 \rightarrow b_2 \Rightarrow BMIG : \{C_j = [s_3], b_j = b_3\}$
6. $b_2 \rightarrow b_3 \Rightarrow BTAB : \{R_j = R_2\}$
7. $b_2 \rightarrow b_1 \Rightarrow BMIG : \{C_j = [s_3], b_j = b_3\}$
8. $b_3 \rightarrow b_7 \Rightarrow FMIG$
9. $b_7 \rightarrow b_3 \Rightarrow BMIG : \{C_j = [s_7], b_j = b_7\}$
10. $b_3 \rightarrow b_7 \Rightarrow BTAB : \{R_j = R_3\}$
11. $b_3 \rightarrow b_2 \Rightarrow BMIG : \{C_j = [s_7], b_j = b_7\}$
12. $b_2 \rightarrow b_1 \Rightarrow BMIG : \{C_j = [s_7], b_j = b_7\}$

We have to take into account that while a migration is taking place, until the last *BMIG* is sent brokers will have the wrong destination for some subscribers. Let us use the first example. Before any migration occurs for b_3, s_6 is connected through b_6. At step 2 when b_3 receives a $BTAB$ message from b_2 it will change the next hop to s_6 to show b_2, since it prefers to trust b_2. After sending an *FMIG* to b_6 at step 4 and receiving its corresponding *BMIG* on step 6 it will correct the next hop to show b_6. If we take a look at the second example we see that there is no need to correct the direction of s_6. When b_3 receives a $BTAB$ from b_2 on step 6 it is already able to correctly route messages to s_6.

If we did not force the migration of the lower brokers on a branch b_3 would not be able to differentiate between the two examples. And when migrating, it would add incorrect information to the routing table of the rest of the brokers. This is the reason a broker only migrates with the subscribers directly connected to it and forces the migration of the rest so they can inform the network of their subscribers.

3.5 Summary

In this chapter we have shown a first approach to supporting broker mobility on a Publish/Subscribe system. The protocol is simple and modular. It treats broker migrations simply as an aggregate of subscriber migrations making the broker act as a proxy for the subscribers. The protocol is able to support the creation of partitions on the network and the publish/subscribe service will keep working correctly on each partition. Whenever two partitions merge, thanks to the leader election algorithm that creates the network topology, one of the two leaders of the partitions will be chosen as a leader for both. And one partition will migrate into the other.

Though simple, there are also some problems. The number of messages required on each migration is pretty high, and some of those messages contain too much information. A *BTAB* message contains the full routing table of the broker, in a small network as in the previous examples it will only have a few lines of data. But on a larger network, and with several subscriptions by each subscriber, that message will have too much information to be easily sent and resent on each migration.

The way migrations are handled can also lead to the inability of removing a subscriber's subscriptions from the system. Let us say a subscriber s is connected to the network and has some subscriptions. The network is partitioned between p_1 and p_2, and s is connected to p_1. After some time s decides to leave the network and sends an *UNS* message for all its subscriptions, a message that is correctly routed to all the brokers on p_1. At that time p_1 has deleted all subscriptions of s, but on p_2 the brokers have not received any *UNS* message and they still keep all the subscriptions as active. Later if p_1 merges into p_2, meaning that between both partitions the leader of p_2 is the one chosen, none of the brokers of p_1 will have s as a local subscriber. And when they receive a $BTAB$ message, that message will contain the subscriptions of s, and they will store it in their routing table. From the point of view of a broker in p_1 we do not know if after s has left it has subscribed again or if they are still the old subscriptions. The brokers will keep the subscriptions of s active even when there is no s anymore.

Chapter 4

Mobile Fault Tolerant Publish/Subscribe: MFT-PubSub

4.1 Introduction

This chapter describes the main contribution of this dissertation. After developing the first iteration of a mobile publish/subscribe protocol, there were some drawbacks that made it quite complicated to use in an environment where the resources are limited. Sending the whole routing table each time a broker migrates and furthermore, causing a cascade of migrations can completely saturate the network in devices with limited connectivity capabilities, such as the ones found in a wireless sensor network. The inability to completely remove a subscriber's subscriptions in some cases can also cause a broker to keep storing messages forever for a subscriber that might never return.

With the idea of eliminating these drawbacks we decided to introduce a timestamp value to any message sent by a subscriber. This timestamp is a simple sequence number that increases in value each time a subscriber sends a message. Using this information whenever a broker migrates it can share the timestamp values for the subscribers with the rest of the brokers and see if any subscriber has sent new messages. If the new partition has a newer timestamp for a subscriber, the broker it is able to deduce that the subscriber has somehow already migrated and that message has not yet been received. This way we can make a broker responsible not only for the migration of the subscribers directly connected to it, but also for all the subscribers that are connected to other brokers lower on the logical connectivity branch. And we do not have to force the migration of those brokers as before. With this change we solve the main problems of the previous approach.

We completely eliminate the need to force the migration of brokers, and at the same time reduce the size of the messages exchanged by the brokers. Where before we were sending the full routing table of the brokers each time one migrated, now we only exchange the timestamps of the subscribers. Using this timestamp we can also determine if a subscriber has completely left the service. Whenever two brokers exchange the timestamp values, if one of them has a higher number but no active subscriptions for any given subscriber, the brokers are able to deduce that the last message of the subscriber has been an unsubscription. And consequently they disable its active subscription on the partition this information exchange takes place.

But this timestamp value alone is not enough to determine the location of a subscriber when part of the network migrates. We mentioned that the timestamp value is increased each time a subscriber sends a message, be it a subscription, unsubscription or migration. But if a broker migrates we change the position of the subscriber in the logical topology, and we do not increase the timestamp values for that subscriber. This can lead to conflicts if several migrations take place at the same time.

Let us use the example provided in the previous chapter in Figure 3.2. Here first b_3 disconects from b_1 and creates a network with 2 partitions. Those two partitions will keep working independently thanks to the leader elections algorithm that creates the network topology. After some time b_6 decides to migrate to b_5 with its subscriber still connected. With this we have changed the logical location of s_6, but b_3 will not receive that migration message yet. So for b_3 the subscriber is still at a lower branch in the topology map. When b_3 chooses to migrate to b_2 it will do so as a proxy for s_3, s_6 and s_7. Without knowing that it has the wrong routing information for s_6. And since the timestamps for s_6 that both b_2 and b_3 have are the same value, b_2 and the rest of the brokers are forced to trust b_3. This erroneous information will be distributed to all the network with s_6 becoming unreachable.

Taking this into account we need more information whenever a migration happens in order to prevent this from happening. The information we are missing is the number of hops between a subscriber and a broker. This value can be easily extracted from the messages sent by a subscriber. With this each broker will store, besides the subscriptions for each subscriber, the timestamp and the number of hops to each one. With both these values whenever a migration happens the broker will check first the timestamp for the subscriber and if the value is the same the number of hops will be compared. Any broker with a higher sequence number will be deemed to have the latest information and correct path on that subscriber, if the timestamps are equal the one that reports being the closest will have a higher probability of being correct.

Due to the way we are able to handle migrations, from the point of view of any node on the system a link failure is indistinguishable from a real physical migration. Furthermore a crash on any broker will cause the migration of the other nodes that are connected to it as if that broker, instead of crashing, had changed its physical location. Moreover, after a broker crash, if the node is able to reboot and start again, the protocol will handle it as a new broker migrating into the network and the crashed broker will be able to resume working normally. For these reasons we decided to call this protocol Mobile Fault Tolerant Publish/Subscribe, or **MFT-PubSub** for short.

4.2 Messages

As with the protocol described in the previous chapter we have defined new messages to be sent in order to support broker migrations. These new messages are shown in Table 4.1. We keep the *FILTERS* message described for the first approach for the same reasons. Whenever a subscriber migrates from one partition to another we need a mechanism for that subscriber to get the active subscriptions on that partition and fix them if needed.

Message	Payload	Client/Broker	Meaning
FILTERS	$f : f \in F$	$b \in B$	Send active subscriptions
BMIG	$\{C_b, O_b\} \in S_b$	$b \in B$	Notify the migration of b
BQUERY	$s \in S$	$b \in B$	Ask for the subscriptions of s
BSUB	$f : f \in F$	$b \in B$	Send active subscriptions

Table 4.1: Message description

The *BMIG* message keeps the name and is also sent whenever a broker migrates to notify the network of its migration, but its behavior has been completely changed. Unlike the previous approach where it contained only information on subscribers that are directly connected to it, this time it contains information on all the subscribers it knows of. Only the timestamp and hop count will be sent for each subscriber, and the broker divides the list into two groups. The first group is composed by the subscribers that are considered child nodes of the broker, in this case all the subscribers that are lower on the topology map than itself. This is the group of subscribers that the broker is representing when it migrates. We call this group C_b for broker b The other group contains the rest of the subscribers the broker knows of, called O_b for broker b. Using these two lists together the broker that receives the *BMIG* message can determine how many subscribers the sender knows of and how updated that information is. If there is any missing information the brokers will try to fix that by themselves. Another parameter that is also important in this message is the number of hops it needs to reach each broker. We have mentioned previously that we use the number of hops to each subscriber as a tie breaker, and whenever a broker migrates, the information on the subscribers that move with it will also need to be updated. Taking the hop count that the broker declares on list C_b and adding the hop count of the *BMIG* message we obtain the new hop count for that subscriber. As with the previous case the *BMIG* message will be routed using the routing path for the subscribers in C_b until it reaches the last broker that was connected to the migrating broker.

The next messages are the ones used to update any inconsistent information detected during the migration process. If a broker, after receiving a *BMIG* message, realizes that it is missing some information on a subscriber due to having a lower timestamp, it will ask the sender of the *BMIG* to send the mising subscriptions using a *BQUERY* message. This message simply contains the identifier of the subscriber the broker is asking about. The *BQUERY* message will be answered with a *BSUB* message. In a *BSUB* a broker sends all the active subscriptions it has on a given subscriber. This message is treated as a grouped *SUB* message from a subscriber and is propagated through the network as such. A broker that has received a *BMIG* message might also decide to send a *BSUB* message to the sender if it realizes that it has a higher timestamp or lower hop count for a subscriber.

This interchange of *BQUERY* and *BSUB* messages is what allows us to prevent the cascading migration of the previous approach. In the best case a broker migration will cause only a single *BMIG* message to be sent into the network, and this message will follow a single path to the last broker. In a system where there are several partitions working independently, and, brokers and subscribers are moving freely between them, the migration will be more complicated. The *BMIG* message might have to take several paths to reach different brokers and any amount of *BQUERY* and *BSUB* messages will have to be sent to fix any erroneous or inconsistent information in the routing tables.

4.3 The protocol

This section will have an in depth explanation of the protocol. In order to introduce the timestamp and hop count values explained in the previous section we have to modify the message reception behavior of the basic messages defined on Phoenix. Algorithm 4.1 shows these changes. Each message has two new parameters t and h,, containing the timestamp and hop count respectively. The hop count is increased each time the message is resent. And since we have these values we can also use them to check if the message we are receiving is newer than the previous one. It might happen, due to the way migrations are handled that a broker receives the same message from a subscriber twice, using this condition we remove the duplicity. Let us imagine a broker that is connected to a lower branch on the network topology with a subscriber at 2 hops of distance. That subscriber sends a *SUB* message and since the broker is close it receives it quickly. But, while that message is being propagated to the network the broker decides to migrate. It ends up in a similar position, on one of the lower branches, but on the other side of the network. Once it finishes migrating, since the *SUB* message is still being propagated into the network it will receive the message again. In this case we ignore the message.

```
when receive(SUB, f, s, t, h) from $z \in I_i$ where $(t, h) > T_i(s)$ do
    $T_i(s) \leftarrow (t, h + 1)$
    if $\nexists(f, \_, s) \in R_i$ then
        $R_i \leftarrow R_i \cup \{(f, z, s)\}$
    foreach $b \in N_i$ where $b \neq z$ do
        send(SUB, f, s, t, h+1) to $b$

when receive(UNS, f, s, t, h) from $z \in I_i$ where $(t, h) > T_i(s)$ do
    $T_i(s) \leftarrow (t, h + 1)$
    if $\exists(f, \_, s) \in R_i$ then
        $R_i \leftarrow R_i \setminus \{(f, \_, s)\}$
    foreach $b \in N_i$ where $b \neq z$ do
        send(UNS, f, s, t, h+1) to $b$

when receive(MIG, s, b, t, h) from $z \in I_i$ where $(t, h) > T_i(s)$ do
    $T_i(s) \leftarrow (t, h + 1)$
    if $z = s$ then
        $X \leftarrow \varnothing$
        foreach $(f, \_, s) \in R_i$ do
            $X \leftarrow X \cup \{f\}$
        send(FILTERS, X) to $s$
    if $b \neq b_i$ then
        if $\exists(\_, \_, s) \in R_i$ then
            $b_j \leftarrow y \in N_i$ where $(\_, y, s) \in R_i$
            send(MIG, s, b, t, h+1) to $b_j$
    foreach $(\_, \_, s) \in R_i$ do
        replace $(\_, \_, s)$ with $(\_, z, s)$ in $R_i$
    sendQueuedMessages($s, z$)
```

Algorithm 4.1: Simple Routing for MFT-PubSub

As explained previously, when a broker migrates it has to send a *BMIG* message. This message contains two groups of subscribers, the ones that are considered children for the broker and the remaining subscribers it knows of. The creation of these two groups is pretty straightforward and is shown in Algorithm 4.2. A broker is able to differentiate the two groups taking a look into its routing table. All the subscribers for which their next hop is defined as the old link towards the leader will be considered as outside of the control of the broker and will integrate the O_b list. The rest of the subscribers will create the C_b list. A broker, at any time, if it cannot deliver a *PUB* message to a subscriber, be it by directly sending it to the subscriber or via another broker, will store it in a queue for that subscriber. When a migration occurs the broker supposes that the brokers belonging in the O_b list it has just created are connected through the newly created connection. In order to replay lost events, the broker will send all queued messages belonging to the subscribers on the O_b list at this time. It may happen that some of those messages cannot be delivered, due to the subscriber not being connected to that partition at that time. However, they will tend to arrive to brokers that are closer to where the subscriber was previously. And with time a single broker will store the messages to be delivered to a subscriber.

Since we might end up moving messages from a broker to another, instead of directly delivering them to a subscriber, we are increasing the network load. This can be lessened by instead of sending individual *REP* messages as in Phoenix, grouping those same messages by subscriber.

As mentioned before, a *BMIG* message has three parameters; two lists of subscribers with their timestamps, separating what the sending broker believes that are children nodes C_j, and the rest O_j, and a hop count for the message. In the algorithms b_i will be the broker running the code and b_j the broker that sent the message. Whenever any broker receives such a message it knows that the sending broker has moved and some corrections might have to be made to the routing tables.

```
function migrate($b_o$, $b_n$)
    $C \leftarrow \varnothing$
    $O \leftarrow \varnothing$
    foreach $(s, z, \_) \in R_i$ do
        if $z = b_o$ then
            $O \leftarrow O \cup \{(s, T_i(s).t, T_i(s).h)\}$
            replace (s, _, _) with (s, $b_n$, _)
        else
            $C \leftarrow C \cup \{(s, T_i(s).t, T_i(s).h)\}$
    send(BMIG, C, O, 0) to $b_n$
    foreach $((s, \_, \_) \in O)$ do
        sendQueuedMessages $(s, b_n)$
```

Algorithm 4.2: Migration of broker b_i

The first step is to check the list of subscribers the migrating broker believes are its children. This is done in lines 3-18 of Algorithm 4.3. Depending on the timestamps of the subscribers there are three different outcomes.

b_j **has newer information on the subscriber:** When b_i checks the timestamps, if it sees that the timestamp it received for a given subscriber is higher than the one it has stored it knows that it is missing information. In this case b_i will ask b_j to send the subscriptions of that subscriber to update its routing table. b_i will also create another C_i list that will be composed of those subscribers that b_i considers are still children of b_j. In this case b_i will add the subscriber to C_i since b_j has newer information on it.

b_j **has older information on the subscriber:** This is the opposite situation to the previous outcome. This might happen if the subscriber migrated without b_j knowing about it. In this case b_i will send the updated information to b_j and will not store the subscriber in C_i since it is able to determine that, in this instance, b_j is wrong.

b_j has the same timestamp as b_i: In this case we have a tie and we need to take the hop counts into account. If b_j declares a lower or equal hop value for the subscriber than what b_i has, b_i supposes that the other broker is correct. b_i will store the subscriber in its C_i list and update the hop count to that subscriber. On the other hand, if b_j is further away from the subscriber than b_i, the broker determines that the other one is wrong and sends the subscriptions of the subscriber to fix this inconsistency.

```
when receive(BMIG, $C_j$, $O_j$, $h$) from $b_j \in N_i$ do
    $C_i \leftarrow \varnothing$
    foreach $(s, (t, h_j)) \in C_j$ do
        if $t > T_i(s).ts$ then
            //If j is newer than i
            send(BQUERY, $s$, $T_i(s).ts$, $T_i(s).h$) to $b_j$
            $C_i \leftarrow C_i \cup \{(s, (T_i(s).ts, h_j))\}$
        else if $t < T_i(s).ts$ then
            //If i is newer than j
            sendSubscriptions(s, $b_j$)
        else if $t = T_i(s).ts$ then
            if $h \leq T_i(s).hops$ then
                //If j is closer than i
                $C_i \leftarrow C_i \cup \{(s, (T_i(s).ts, h_j))\}$
                $T_i(s) \leftarrow (t, h + h_j + 1)$
            else if $h > T_i(s).hops$ then
                //If i is closer than j
                sendSubscriptions(s, $b_j$)
    foreach $(s, (t, \_)) \in O_j$ do
        if $t > T_i(s).ts$ then
            send(BQUERY, $s$, $T_i(s).ts$, $T_i(s).h$) to $b_j$
        else if $t < T_i(s).ts$ then
            sendSubscriptions(s, $b_j$)
    if $h = 0$ then
        foreach $(s, \_, \_) \in R_i$ where $s \notin (C_j \cup O_j)$ do
            sendSubscriptions(s, $b_j$)
    $X \leftarrow \varnothing$
    foreach $s \in C_i$ do
        $X \leftarrow X \cup \{b \in N_i$ where $(\_, b, s, \_) \in R_i \wedge b \neq b_j\}$
        foreach $(\_, \_, s) \in R_i$ do
            replace $(\_, \_, s)$ with $(\_, b_j, s)$ in $R_i$
    foreach $y \in X$ do
        send(BMIG, $C_i$, $\varnothing$, $h$+1) to $y$
    foreach $s \in C_i$ do
        sendQueuedMessages $(s, b_j)$
```

Algorithm 4.3: Mobile brokers with timestamps

After checking the subscriber on C_j, b_i will take a look at the remaining subscribers b_j knows on O_j, in lines 19-23. In this case the treatment is simpler than before, the broker will only look at the timestamps. If b_j has newer information on a subscriber, it will ask for it with a *BQUERY* message. Otherwise, if b_j has a lower timestamp b_i will send directly the subscriptions for that subscriber.

The next step, shown in lines 24-26 is only carried out on the first broker that receives a *BMIG* message. It might happen that the broker b_i has subscriptions by subscribers in its routing table that b_j does not know. In a partitioned network a subscriber may join one of the partitions, and when a broker migrates into it, the broker needs to be informed of that subscriber. In this case the broker b_i will send the subscriptions of all the subscribers that are not in either C_j or O_j.

Finally broker b_i has to resend the *BMIG* message it received in lines 28-33. In order to correctly route this message b_i has to check for the next hop for all the subscribers in its newly created C_i list. In this way we direct a *BMIG* message to the oldest known brokers for those subscribers. In line 31 we also change the next hop for the subscribers to show the newly migrated broker. The list O_j is only used by the first broker that receives that message and there is no need to send it anymore. The *BMIG* is handled as if it were a *PUB* message for a subscriber, and each broker that receives it will update the next hop accordingly. This way we can correctly route messages to the new location of the subscriber by updating a few brokers.

The last step of the migration process is to send the queued messages for the subscribers on C_i back to b_j so that the broker can route them to their recipients.

If in any case when checking for the timestamps of subscribers the broker does not know about that specific subscriber, it will create a new entry with 0 for the value of the timestamp. This way it will always ask for information concerning that specific subscriber.

The two messages shown in Algorithm 4.4 are the ones responsible for fixing any difference in the routing tables of the brokers. The first message,

```
when receive(BSUB, s, $S_j$, $T_j(s)$, h) from $z \in N_i$ where
  $T_j(s) > T_i(s)$ do
    foreach $(\_, \_, s) \in R_i$ do
        $R_i \leftarrow R_i \setminus \{(\_, \_, s)\}$
    foreach $f \in S_j$ do
        $R_i \leftarrow R_i \cup \{(f, z, s)\}$
    $T_i(s) \leftarrow \{(T_j(s).ts, T_j(s).h + h + 1)\}$
    foreach $b \in N_i$ where $b \neq z$ do
        send(BSUB, s, $S_j$, $T_j(s)$, $h + 1$) to $b$
    sendQueuedMessages $(s, z)$

when receive(BQUERY, s, t, h) from $z \in N_i$ where $(t, h) < T_i(s)$ do
    sendSubscriptions(s, z)

function sendSubscriptions(s, z)
    $S_i \leftarrow \varnothing$
    foreach $(f, \_, s) \in R_i$ do
        $S_i \leftarrow S_i \cup \{f\}$
    send(BSUB, s, $S_i$, $T_i(s)$, 0) to $z$
```

Algorithm 4.4: Messages to fix inconsistent routing tables

BSUB, is treated as if it were a *SUB* message from a subscriber. For this reason the broker will ignore it if it already has a higher valued timestamp. In order to process the *BSUB* message, since we do not know how many active subscriptions that subscriber has, a broker will first delete all the subscription for that subscriber. subsequently, in the next step it will store all the subscriptions included in the message. It will also update the timestamp and hop count for that subscriber. Since this message is treated as a *SUB* message the *BSUB* is resent to all the neighbors the broker has. Finally, since this message can be received in order to fix an error in the routing table and at this moment the broker believes it has a path to the subscriber it will send its queued messages.

The handling of a *BQUERY* message is shown in lines 10-11. When a broker receives a *BQUERY* message, with a timestamp older than what it has, it will directly answer back with the subscriptions of the subscriber the message is asking for. The subscriptions will be grouped in a *BSUB* message that will be propagated to the network.

4.4 Migration example

As in the previous chapter we will analyze the behavior of the system with two examples. We will use the same graphs, but for ease of reading they are also copied here as Figures 4.1 and 4.2.

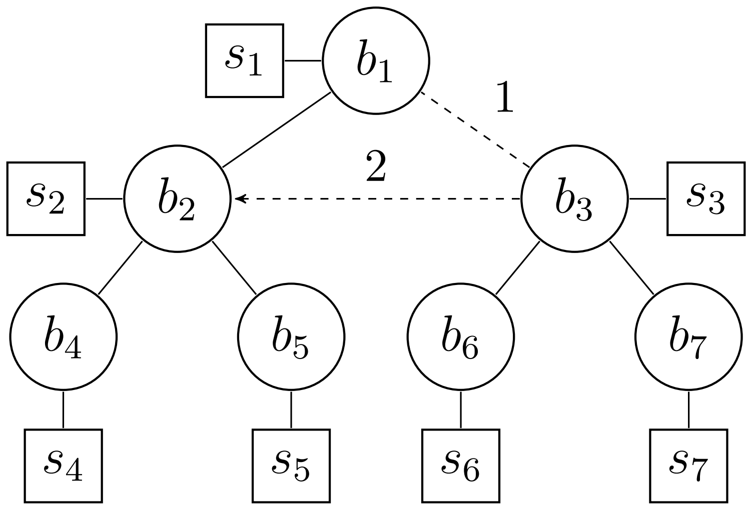


Figure 4.1: Simple migration example. $b_3 \leftrightarrow b_1$ link is lost and $b_3 \leftrightarrow b_2$ is created.

The messages sent in order to complete the migrations in both examples were quite similar in the first approach we explained in the previous chapter. Using MFT-PubSub we are able to reduce the messages to a minimum in the best case scenario. For the migration shown in Figure 4.1 we have the following sequence of messages:

1. $$b_3 \rightarrow b_2 \Rightarrow BMIG : \left\{ \begin{array}{l} C_j = [(s_3, 1, 1), (s_6, 1, 2), (s_7, 1, 2)] \\ O_j = [(s_1, 1, 2), (s_2, 1, 3), (s_4, 1, 4), (s_5, 1, 4)] \\ h = 0 \end{array} \right\}$$

2. $$b_2 \rightarrow b_1 \Rightarrow BMIG : \left\{ \begin{array}{l} C_j = [(s_3, 1, 1), (s_6, 1, 2), (s_7, 1, 2)] \\ O_j = \varnothing \\ h = 1 \end{array} \right\}$$

Here we have also included the full content of the messages. The entire migration is comprised of just two *BMIG* messages that reach b_1. In the first message we can see how C_j and O_j are filled, with subscriber identifiers, timestamp values (in this case set to 1), and the number of hops to reach each subscriber. b_2 checks the subscribers on both groups and since everything is correct it sends it towards the broker it had stored as the next hop for subscribers s_3, s_6 and s_7, in this case b_1. Once b_1 receives the message from b_2 all the required changes in the routing tables are done. We do not need to notify b_4 or b_5 since they are connected to the rest of the network through b_2 and we have already corrected it. Neither do b_6 or b_7 need to be notified of this migration since they connect through b_3. Note that if we look into b_6, before the migration it had five hops to reach s_5, and after the migration it is at four hops, but we have not changed that value. The hop count is more of a tie breaker in the case of having the same timestamp and at worst the *BMIG* message will be sent until it reaches the broker with the hop value of one. Once it reaches this last broker, it will issue a *BSUB* to fix the incorrect routing. We will analyze this situation with the next example.

Figure 4.2 shows a more complicated migration. In this case the protocol makes use of the *BQUERY* and *BSUB* messages in order to fix the inconsistencies after a migration. The sequence of messages is quite a bit more complicated than before and it looks as follows:

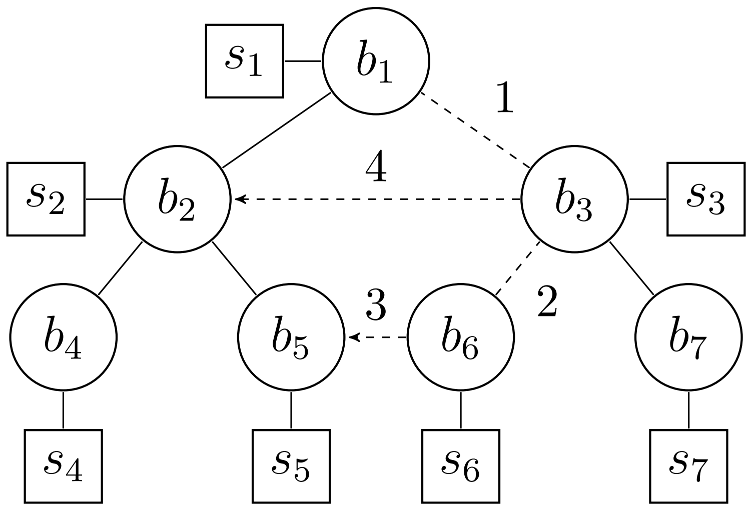


Figure 4.2: Complex migration example. First $b_3 \leftrightarrow b_1$ link is lost and after $b_6 \leftrightarrow b_3$ is also lost. $b_6 \leftrightarrow b_5$ is connected and allowed to completelly migrate before $b_3 \leftrightarrow b_2$ is created.

1. $b_6 \rightarrow b_5 \Rightarrow BMIG : \left\{ \begin{array}{l} C_j = [(s_6,1,1)] \\ O_j = \begin{bmatrix} (s_1,1,3),(s_2,1,4),(s_3,1,2), \\ (s_4,1,5),(s_5,1,5),(s_7,1,3) \end{bmatrix} \\ h = 0 \end{array} \right\}$

2. $b_5 \rightarrow b_2 \Rightarrow BMIG : \left\{ \begin{array}{l} C_j = [(s_6,1,1)] \\ O_j = \varnothing \\ h = 1 \end{array} \right\}$

3. $b_2 \rightarrow b_1 \Rightarrow BMIG : \left\{ \begin{array}{l} C_j = [(s_6,1,1)] \\ O_j = \varnothing \\ h = 2 \end{array} \right\}$

4. $b_3 \rightarrow b_2 \Rightarrow BMIG : \left\{ \begin{array}{l} C_j = [(s_3,1,1),(s_6,1,2),(s_7,1,2)] \\ O_j = [(s_1,1,2),(s_2,1,3),(s_4,1,4),(s_5,1,4)] \\ h = 0 \end{array} \right\}$

5. $b_2 \to b_1 \Rightarrow BMIG : \left\{ \begin{array}{l} C_j = [(s_3,1,1),(s_6,1,2),(s_7,1,2)] \\ O_j = \varnothing \\ h = 1 \end{array} \right\}$

6. $b_2 \to b_5 \Rightarrow BMIG : \left\{ \begin{array}{l} C_j = [(s_3,1,1),(s_6,1,2),(s_7,1,2)] \\ O_j = \varnothing \\ h = 1 \end{array} \right\}$

7. $b_5 \to b_6 \Rightarrow BMIG : \left\{ \begin{array}{l} C_j = [(s_3,1,1),(s_6,1,2),(s_7,1,2)] \\ O_j = \varnothing \\ h = 2 \end{array} \right\}$

8. $b_6 \to b_5 \Rightarrow BSUB : \{s = s_6, S_j = [f_6], T_j(s) = (1,1), 0\}$

9. $b_5 \to b_2 \Rightarrow BSUB : \{s = s_6, S_j = [f_6], T_j(s) = (1,1), 1\}$

10. $b_2 \to b_4 \Rightarrow BSUB : \{s = s_6, S_j = [f_6], T_j(s) = (1,1), 2\}$

11. $b_2 \to b_1 \Rightarrow BSUB : \{s = s_6, S_j = [f_6], T_j(s) = (1,1), 2\}$

12. $b_2 \to b_3 \Rightarrow BSUB : \{s = s_6, S_j = [f_6], T_j(s) = (1,1), 2\}$

13. $b_3 \to b_7 \Rightarrow BSUB : \{s = s_6, S_j = [f_6], T_j(s) = (1,1), 3\}$

Once both migrations hava taken place we have a similar number of messages to those shown in the previous chapter. Though we have to categorize this as a worst case scenario. The first three messages are for b_6 to complete its migration and are similar to the previous example. The *BMIG* message is routed towards b_1 were it cannot continue since the link towards b_3 is broken. At this time b_5, b_2 and b_1 have the correct routing for s_6.

However, once b_3 begins its migration we start having some complications. Note that the content of b_3's *BMIG* message still contains s_6 as a child node, since it has not received any new message notifying it of the migration. Once this message reaches b_2 it will check the timestamps and hop counts against its own. b_2 has a timestamp value of 1 for s_6, the same as b_3, so we have to use the hop value. b_3 claims to be at two hops from s_6 and b_2 knows that it is at three itself, so it trusts b_3's claim that s_6 is its child. At this time an incorrect routing path is set for s_6 on b_2, it will point towards b_3 instead of b_5. But, as we see on steps 5 and 6 b_2 will redirect the *BMIG* message not only to b_1 as previously, but it will also send a copy to b_5 since this was the last next hop towards b_6. b_1 will receive the message and act accordingly by changing the next hop for s_3 and s_7. The moment b_6 receives the *BMIG* message sent by b_5 in step 7 it will realize that its hop count is one whereas the hop count claimed by b_3 is two. Since it has a lower hop count it corrects the routing tables of the rest of the brokers by sending a *BSUB* message that is propagated to the rest of the network as seen in steps 8-13.

Using the hop count to break the ties created by having the same timestamp can cause temporary inconsistencies in the network as seen in this example. However, as the *BMIG* message is routed towards the old connection of the subscribers, it also means that the brokers that receive the message have a lower hop count to reach that subscriber. Eventually it will reach a broker with a lower value than the one the message contains and that broker will stop including the subscriber in the child list and correct the inconsistency with a *BSUB* message. Any *PUB* message sent during this inconsistency will be routed toward the wrong broker. In the example shown in Figure 4.2 b_3 might receive *PUB* messages sent to s_6, but since b_6 is unreachable it will store those messages. When it receives the *BSUB* message at step 12 it will answer back with all the queued messages it has for s_6 as shown in line 9 of Algorithm 4.4. Thus in the end s_6 or any other subscriber in the same situation will not loose any message due to the inconsistency created during the migration.

4.5 Performance evaluation

In this section we will analyze the performance of the MFT-PubSub protocol in a fully mobile environment. Before obtaining performance metricks we have to check if the protocol works as intended and is able to support the migrations of the brokers. The first step in validating the protocol was carried out using JBotSim [15].

JBotSim is a library written in Java that allows us to easily test code for distributed networks. It does this by allowing us to define function for when something happens in the network, be it the reception of a message or the creation of a new link. Using this tool we can easily and quickly deploy the protocol in a distributed network and simulate the migration of brokers in the conditions that we specify. Thought it is not as extensive as other tools, it allows us to test specific use cases in minutes and run tests where we can see how the system behaves in real time. Using JBotSim to test the protocol is as simple as implementing the system behavior whenever a message is received. Then we can create the network by dropping the nodes onto a 2D map of the environment and allowing them to create connections using a predefined wireless connection range. Once the connectivity tree is created we can manually move any node to any point in the map and check the messages it sends to notify the migration. We can also check the routing tables, or the queued messages of the rest of the brokers and see if they are behaving correctly. Once we had validated the correctness of the protocol using this tool we decided to implement it in a simulation environment where we could obtain more information on the protocol behavior. Even though JBotSim is a useful tool to validate a distributed algorithm it lacks some features that would make it a complete wireless network simulator. The main disadvantage is that by default all communications are carried out through channels that are perfect. If two nodes are in communication range of each other, a message sent by one will always be received by the other, there is no calculation for signal strength or interference caused by other nodes.

For this reason we decided to use the OMNeT++ [52] simulation environment, and more specifically the Castalia [8] simulator. OMNeT++ is a general network simulation tool that is extensively used to test the validity of distributed algorithms or even specific deployments. This is quite a powerful tool and if it lacks any feature it can be expanded by the use of the multiple frameworks that use OMNeT++ as a base.

This is the case of Castalia which was developed as a simulator specifically for wireless sensor networks. It was developed following a scheme comprised of layers, separated in modules. The user can develop the protocol in different layers, such as the application, MAC, radio or even physical movement. In order to create a node the user has to define the set of layers required for that node. Changing the radio model is as easy as changing a line in a configuration file. This tool also simulates the common problems found on a real deployment of a wireless sensor network, such as network connectivity degradation due to distance or saturation of the radio spectrum. The most important feature it has, and the reason we choose it, is that it support the simulation of a wireless node that is physically moving through a set of coordinates in a 2D map. Whenever a wireless message is sent, Castalia will calculate the signal strength for that message in the rest of the nodes in the simulation. If that signal is above some threshold in order to be correctly received, the message will be delivered to the node. Using this we can simulate a network where all the nodes are moving in predefined patterns and the framework will calculate each time the set of nodes that are able to receive the message. This subsequently might cause a migration following the protocol presented in this dissertation.

Castalia also offers tools specific to wireless sensor networks, such as the ability to calculate energy consumption of the nodes. On this occasion we do not make use of this feature. In our simulations we want to test the validity of our proposed protocol. We want the system to be as responsive as possible, and to this end the messages must be sent as quickly as possible the moment any change is detected. The change detection rate is also related to the timeout the leader election algorithm uses for its heartbeat mechanism. As we explained before we use this heartbeat to check when a broker or a subscriber has to migrate. If this message is sent once a minute we will be able to act in the case of a migration at most at the frequency of once a minute. A slower heartbeat rate will also increase the time a broker or subscriber is disconnected if they loose one of their links, since they have to wait until they receive the heartbeat to migrate. For these reasons we choose to have the antenna module active all the time. Mostly waiting to react whenever a message is received. And, since this the one module that consumes the most energy, with several orders of magnitude above the rest, we decided to ignore energy consumption for these tests. In the future it would be interesting to try and implement some synchronization mechanism to try to save energy. For example we could use the heartbeat message to piggyback other messages, this way even though we might cause some delays in the delivery of the messages, we would be able to have a known timeout for the communication. And with this we might be able to turn off the antenna when we know no messages are going to be sent.

4.5.1 AODV

To test the validity of the proposed protocol we chose to compare it to a similar algorithm that would allow for communication in a fully mobile environment. Due to the difficulty of finding a publish/subscribe algorithm that would allow for full mobility of all the nodes at any time we chose to implement a more general communication protocol. For this we chose to use AODV [40].

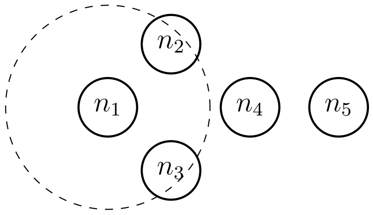
AODV is short for Ad hoc On-Demand Distance Vector, and is intended to be used for mobile nodes in an ad hoc network. The algorithm is designed to find unicast routes between a sender and the destination. To compensate for the dynamic nature of the network it uses a reactive approach to route creation. Routes are created only whenever a node wants to send a message.

The algorithm works as follows. Nodes will be continually exchanging hello messages. These messages are used to determine the neighboring nodes and whether the links are still alive. Each node will keep a list of neighboring nodes they can communicate with, and in the case that a neighbor fails to send the periodic hello message a broken link is detected. The nodes also store a routing table for all the destinations they tried to communicate with. This routing table is simply the next hop from the set of neighbors it has, with each node only knowing the next hop and not the full route. Whenever a node wants to send a message to a destination that is not in the routing table it will issue a Route Request message (*RREQ*). This message will be propagated via broadcast to the rest of the network. Any node that receives such a message will first check if it already has a route to the destination. In the case that a route exists, or it is the destination, it will answer back with a Route Reply (*RREP*), if it does not have any route it will also broadcast the *RREQ*. This behavior is described in Figure 4.3 where a node n_1 wants to find a route towards n_5. It might happen that several routes are created towards the destination, in this case the route with the shortest hop count is used. Once a route is created the originator can send the message they want. Each route created also has a predefined timeout, once no messages have been sent through that route on some time the node invalidates the route. If this happens or a broken link is detected a Route Error (*RERR*) message is sent back towards the originator of the route.

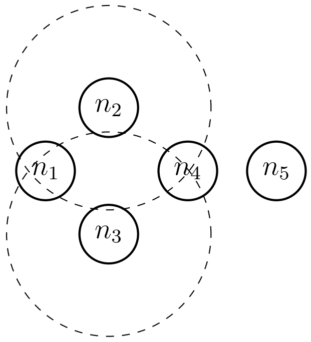
Due to its ability to handle route creation and send messages without the need to stop and synchronize the nodes of the network, while every node if moving, we chose AODV to compare with our protocol. Even though MFT-PubSub is designed for publish/subscribe and is able to handle publication as if they were multicast messages, whereas AODV only handles unicast, we believe this is a fair comparison with the parameters described in the next subsection. Another interesting feature of MFT-PubSub is that we can also create routes in a similar way to what AODV does. If each subscriber creates a subscription that refers to itself we can have a system where, instead of multicast messages, any node is able to send a message directly to a subscriber. These routes will be created when the subscriber issues the subscription and will be updated with each subsequent migration. Thus, instead of having to create a route whenever we want to send a message the system already has one ready for us. For simplicity we also define roles for each node such as broker, subscriber or publisher. However, MFT-PubSub supports the ability of a node to take any of the three roles at any time, even all three at the same time.

4.5.2 Simulation parameters

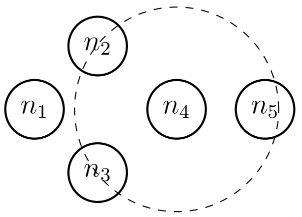
Before starting the simulations we have to define the parameters these simulations are going to be run at. We want to test several situations, from



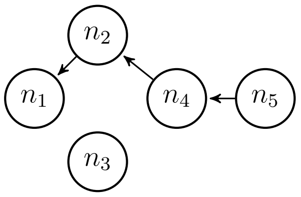
(a) n_1 sends a RREQ message for n_5 that is received by n_2 and n_3.



(b) n_2 and n_3 do not have any route to n_5 so they also broadcast the message.



(c) n_4 also broadcasts the message.



(d) Finally n_5 answers back with a RREP message and the route is created.

Figure 4.3: AODV route creation.

having a few nodes to creating a bigger network. In some preliminary tests we realized that having a few nodes in a big area will not allow the correct communication of the nodes. Most of the time the nodes will be too far away from each other to be able to communicate with each other and this will only permit brief moments when the messages can be sent. For this reason we choose to link the simulation area with the amount of nodes we are simulating, by defining a node density. This density has been defined as 0,005 nodes per square meter, giving each node an area of 200 square meters. The five different configurations used can be seen in Table 4.2. We have given each node a specific role, even those on AODV. If a node is defined as a broker or subscriber in AODV it will not initiate any RREQ messages, though subscribers and publishers are also allowed to route messages to another node if needed. For all simulations we keep the amount of publishers equal, and low, since we are mostly interested in checking the ability of the brokers to handle their routing table as the network size increases.

Configuration	#publishers	#susbscribers	#brokers	area
C2	2	2	2	35x35 m^2
C4	2	4	4	45x45 m^2
C8	2	8	8	60x60 m^2
C16	2	16	16	80x80 m^2
C32	2	32	32	110x110 m^2

Table 4.2: Simulation configurations.

The duration of the simulation is set at 700 seconds, with the last 200 of those being a period where no new messages are created. This gives time for the messages that are in transit or in the queues to be delivered to their recipients. AODV has no mechanism to store a message to be delivered at a later date. But, in MFT-PubSub we store all *PUB* messages that could not be correctly routed to its destination. When a message is not delivered, it means that it is still stored in the memory of some of the brokers waiting for the subscriber to be reachable again. This means that even though the messages have not been delivered by the time the simulation ends if we left it to run enough time, eventually they would be delivered. The nodes defined as publishers will send messages to the network every second for the first 500 seconds of the simulation.

One of the most important parameters to check the validity of the protocol is the mobility model chosen. After starting with a random walk model where for each tick of a clock a random direction and random distance is chosen for the new location(within some predefined parameters), we realized that this created a few clusters of nodes, but did now allow the nodes to cover the whole area. We choose the random waypoint model [12], at speeds of 2-4-6-8-10 meters per second, that go from walking quite fast to the top speed of a sprinter. In this mobility model the nodes choose a random point in the simulation area and go towards that point at a predefined speed. Once it reaches that point it will choose another one and repeat the procedure. This means that nodes will walk around the whole simulation area more often and prevents the creation of clusters.

For the radio module we choose to use one that is already configured in the Castalia framework, the CC2420 chip [1]. This chip is designed for low power wireless communication applications such as a wireless sensor network that uses the 2.4GHz frequency band. The transmission power is set to 0dbm in order to maximize the communication range of the devices which, as this chip is designed for low power use cases, is quite low. One of the most interesting points of Castalia is its ability to calculate if a node should receive a message. This is done by calculating a Signal to Interference Ratio (SINR) for each message reception. We set the radio module to use the additive collision model that will allow the delivery of the message in the case of a concurrent transmission if one of the signals is strong enough. We also use the normal mode for the chip.

The last module we have to configure is the Media Access Control (MAC) module. For the simulations we have used the Carrier-Sense Multiple Access (CSMA) configuration that comes with the Castalia installation. CSMA is a common MAC protocol used in both wired or wireless computer networks. Using this protocol a node will check if any other node in the network is transmitting in the shared medium, the 2.4GHz band in our case, before it begins the transmission. If a transmission is detected the sender will wait a random interval before trying to send the message again.

All possible combinations of size and speed are repeated 10 times during the simulation with different random seeds each time, to be able to repeat the simulation conditions. This way we can obtain an average of the results to better reflect the performance. We also want to mention at this point that, where MFT-PubSub only needed a few minutes to run the biggest simulations, the implementation AODV required between six and up to 16 hours for the biggest ones. This long simulation time made it difficult to test

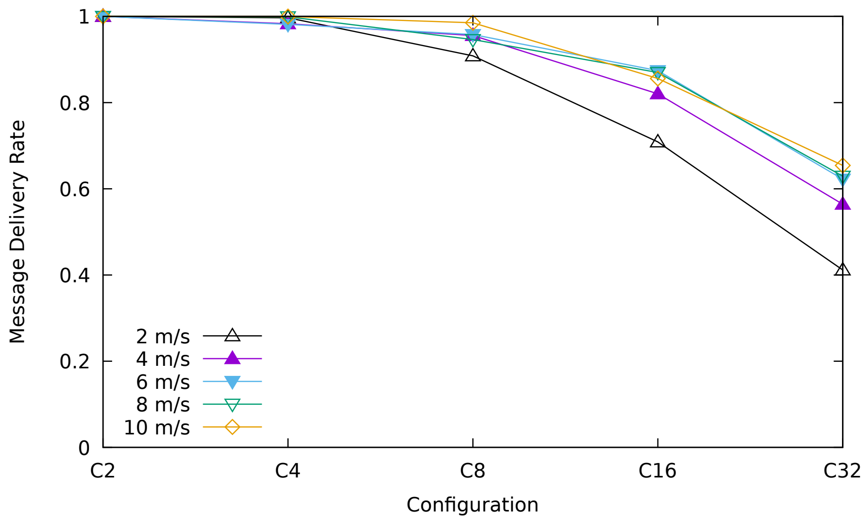
AODV and was mostly caused by the number of messages that it needs to send. Each time a message is sent Castalia has to calculate and deliver those messages to nodes in range, and AODV being an algorithm prone to sending broadcast messages flooding the network creates a huge amount of messages.

4.5.3 Delivery rate

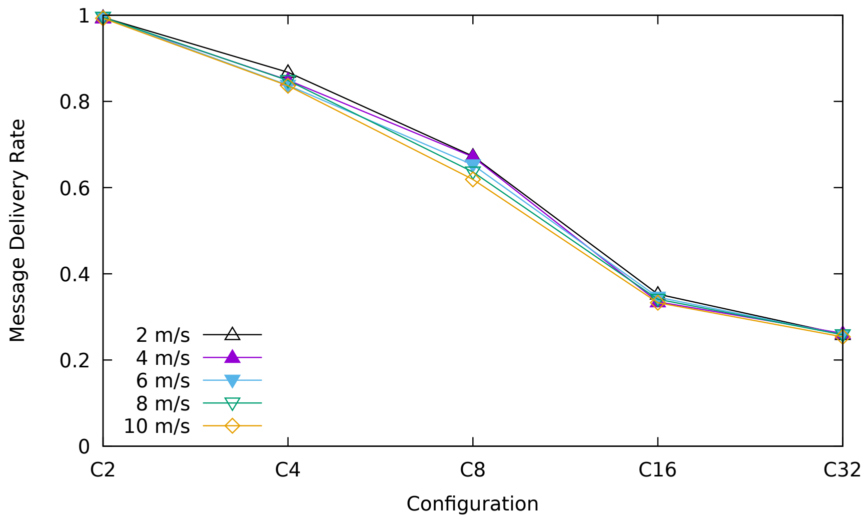
The first and most obvious metric to measure is if the messages are being delivered correctly to the subscribers. We call this the delivery rate. We consider the delivery rate as the number of messages a subscriber receives with respect to the ones that were originally sent to it. This metric does not take into account messages that are stored in the routers, where hopefully, with time, they will reach the subscribers.

In Figure 4.4 we see a comparison of the delivery rates for both MFT-PubSub and AODV. MFT-PubSub seems to have better resilience to speed, even improving the delivery rate as the speed goes up. We will later analyze this behavior. Both protocols are strongly affected by the network size, the bigger the network, the harder it is to correctly deliver a message.

In order to find the reason behind the strange behavior of MFT-PubSub where faster mobility improves the delivery rate of messages we have to look at the actual number of messages delivered. Figure 4.5 shows the comparison for delivered messages both with respect to the network size and the speed. The improvement of delivery rate for higher speeds can be further analyzed with Figure 4.5b. This figure shows the results of the C16 configuration using the different predefined speeds. Here we can see a slight increase on the total number of messages delivered as the speed increases, but as the speed reaches 6 m/s it starts to drop. This happens due to the way MFT-PubSub stores the undelivered messages. Whenever a broker cannot find the path to a subscriber it will store it and wait for new information on that subscriber, as the speed increases there are more opportunities for a subscriber both to pass by a broker that has a message for it and to be in range to communicate. If we look at the amount of messages delivered in Figure 4.5a, we observe that for C32 AODV has delivered more messages than MFT-PubSub, whereas in Figure 4.4 we show that MFT-PubSub has a better delivery rate for that same configuration. This difference is mainly due to how a publish/subscribe system works; in order to be able to route a message to a subscriber, that subscriber has to actually subscribe to the content, and the subscription is what creates the route. In these simulations we only take into account the messages that are routed to a subscriber as having to be delivered to that subscriber. If a subscriber is in another partition and a broker does not know it needs to route a PUB message to that subscriber, since it is not in the routing table, the message will not be considered a loss. AODV works in a more reactive approach, it will always try to create a route to the subscriber and retry several times until it fails. If we take a look at medium sized networks MFT-PubSub is able to correctly deliver more messages as shown in Figure 4.5b, where MFT-PubSub outperforms AODV at all speeds.

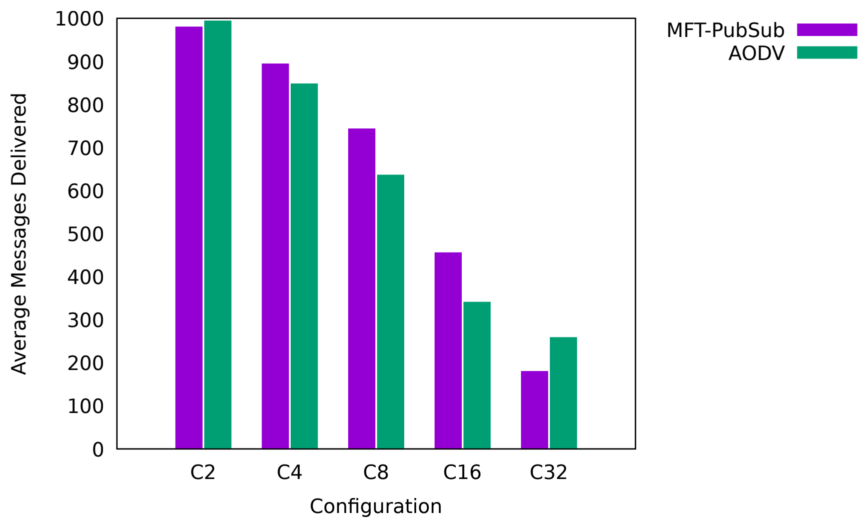


(a) MFT-PubSub.

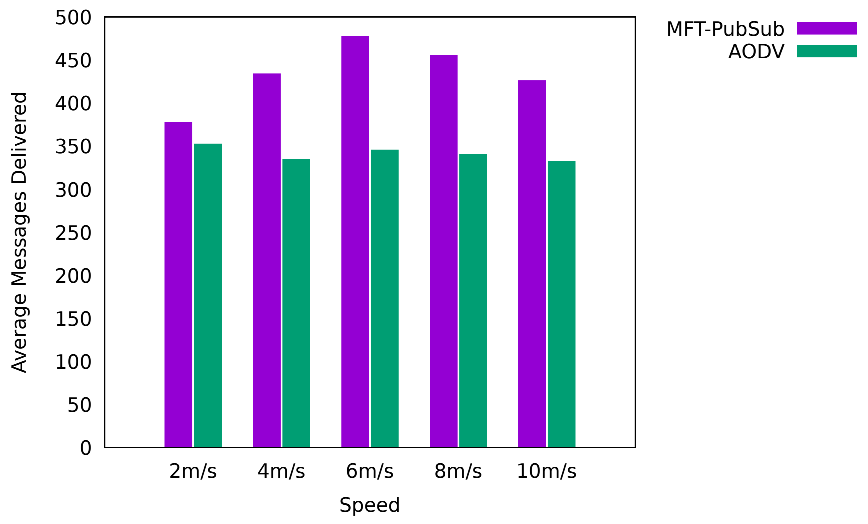


(b) AODV.

Figure 4.4: Message delivery rate comparison of both algorithms depending on the size of the simulation.



(a) Node speed of 8 m/s.



(b) C16 configuration.

Figure 4.5: Average number of messages correctly delivered. In Figure 4.5a we show the results for a node speed of 8 m/s on different configurations. And, in Figure 4.5b we show the results of all speeds for the C16 configuration.

4.5.4 End-to-end delay

The next metric we will take a look into is the delay the protocol causes in order for a message to reach its destination. We call this metric the end-to-end delay. Figure 4.6 shows the comparison between both protocols. It is interesting to note that MFT-PubSub is able to keep up with AODV in this metric in spite of the differences in how message delivery works. AODV will try to deliver the message as soon as possible, by creating a route if it does not have one already and sending the mesage afterwards. This metric measures this delay, caused by the need to create the route, since, once the route is created the message is sent directly towards the destination. In MFT-PubSub we already have predefined routes so it should need less time to deliver the message since half the work is already done. Nevertheless, in this metric we also have to take into account those messages that had been stored by the brokers due to it being not possible to deliver them. Here we are measuring the time from the message creation to its delivery. Any message that is stored to be delivered later will greatly increase this value.

We also observed that some of these values in the case of AODV are higher than they should be, mostly those pertaining to C4. In Figure 4.7 we show the number of hops a message needs to be correctly delivered. In contrast to the average delivery time, here we are mostly interested in knowing how many hops a message needs in order to be delivered from source to destination. Whenever a broker has to replay undelivered events we reset the hop count for that message. We can see that in the case of C4, AODV needs more hops that the other configurations in order to deliver a message. We ran this configuration with different random seeds and we always obtained the same behavior. This behavior means that the message is being sent from a node to another without being able to be delivered due to the changing nature of the network.

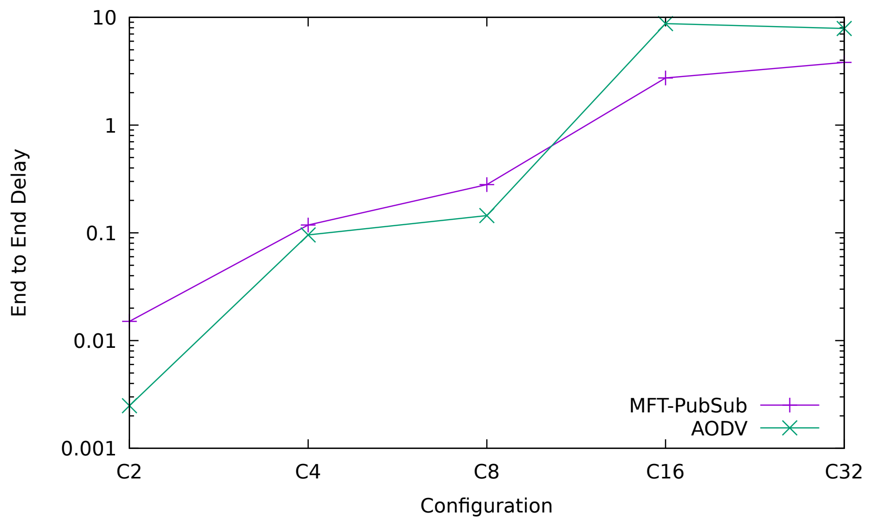
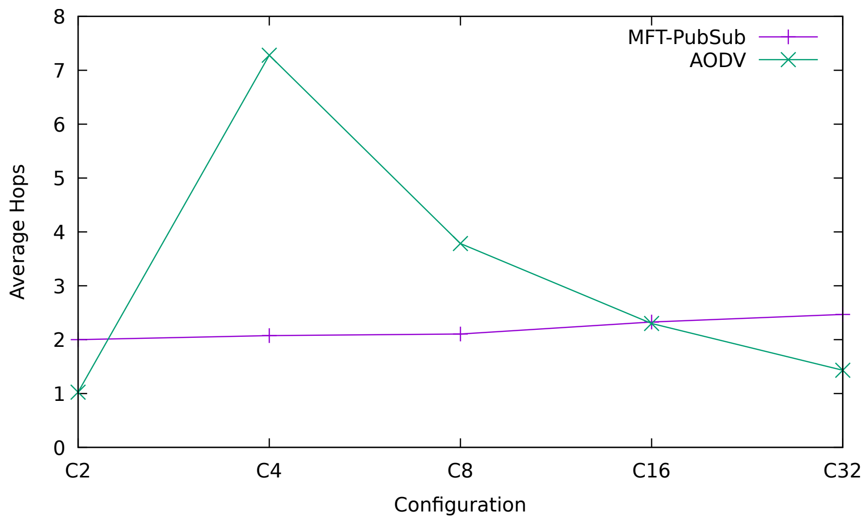
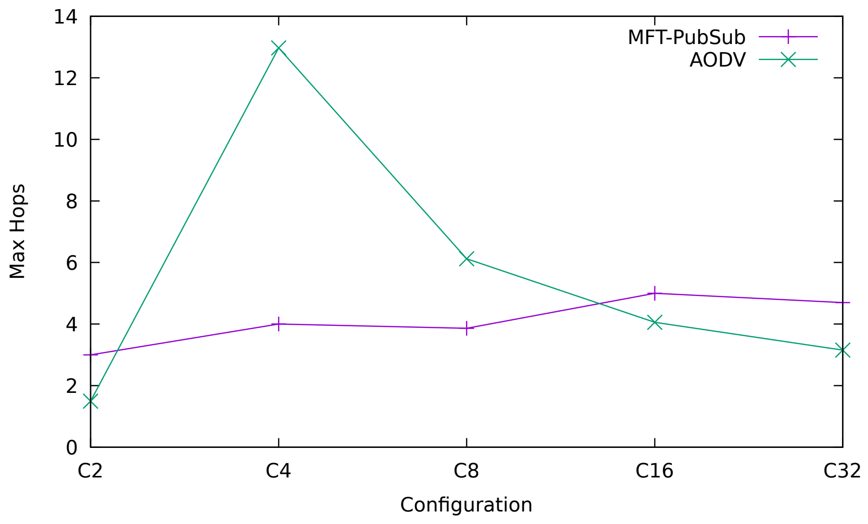


Figure 4.6: Comparison of end-to-end delay, in seconds, for data messages for a node speed of 8 m/s. Note the logarithmic scale on the y axis.



(a) Average number of hops.



(b) Maximum number of hops.

Figure 4.7: Number of hops a message makes on the network before it is delivered for a node speed of 8 m/s.

4.5.5 Number of messages exchanged

Finally, since we decided not to analyze the energy consumption of the protocol, we take a look at another metric that might tell us about the efficiency of it. In this case we choose the total number of messages by all the nodes exchanged in the network. Due to the mobility of the nodes messages have to be sent to inform neighboring nodes or to recreate broken routes. Looking at the total number of messages gives us an insight into how much overhead the protocol needs to route the messages it has delivered. We show this comparison in Figure 4.8. We have to emphasize that this is the average number of messages sent by each node, to obtain the total we have to multiply this by the number of nodes. For the smallest configuration AODV needs fewer messages than MFT-PubSub, since the latter has to create the whole publish/subscribe structure in order to function correctly. However, as the number of nodes increase we see an explosion of messages by AODV, whereas MFT-PubSub barely changes in number. With this we conclude that even though AODV works fine it is highly inefficient due to the amount of broadcast messages it needs to create the routes, with mobility complicating things more. While nodes are moving an already created route might become unreachable, and require another round of route request messages.

If we take a look to the number of messages sent by individual nodes we also notice that on MFT-PubSub most of the work is done by brokers that are responsible for routing messages and keeping up with the changes caused by migrations. Publishers and subscribers send fewer messages.

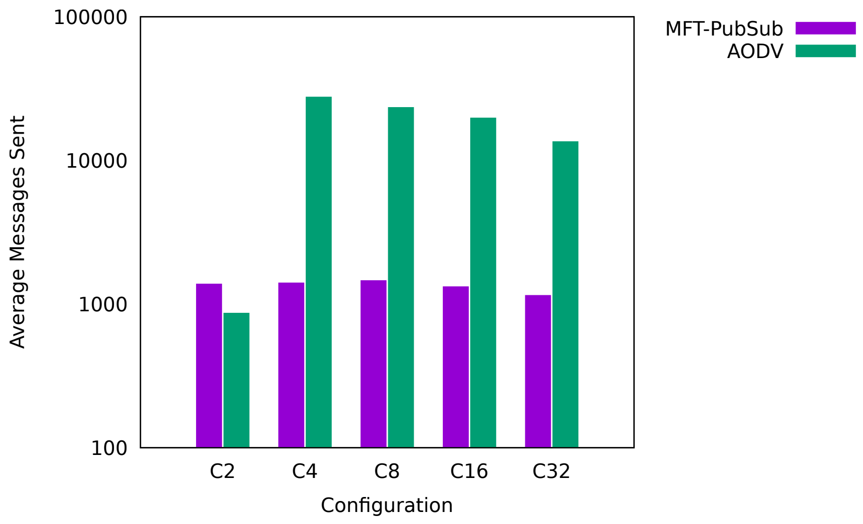


Figure 4.8: Comparison of the average number of messages sent by each node in order to correctly route messages for a node speed of 8 m/s. Note the logarithmic scale on the y axis.

4.6 Summary

In this chapter we have presented a publish/subscribe protocol that supports full mobility of nodes, called MFT-PubSub. We have improved on the previous protocol by reducing the number of messages a migration requires to stabilize the network, and reducing the size of those messages. The protocol is more complex to implement than the one presented in the previous chapter due to the handling of timestamps and hop counts.

Thanks to the timestamps a broker can notify the migration of not only the subscribers that are directly connected to it, but also those that are in a lower branch in the communication topology. Using those timestamps we also prevent the creation of phantom subscriptions where a subscriber has unsubscribed from one partition, but not the other. Whenever the migration occurs brokers can compare their timestamps and if the one with the newer timestamp has no subscriptions for a subscriber, it means that the subscriber has left the network.

We have implemented the protocol in Castalia in order to obtain performance metrics and compare it with similar solutions. In our case we compare MFT-PubSub with AODV which is an established algorithm for communication in a mobile network. We observe that the protocol proposed in this dissertation improves the performance of AODV, though it suffers in bigger sized networks. We have also shown that the number of messages increases linearly with the number of nodes on the network, with each node suffering a similar load, though nodes closer to the leader will have a higher load.

One of the main improvements that could be made is to try to optimize the algorithm that creates the communication tree. Using the leader election mechanism we often end up with trees that are severely unbalanced, with most of the nodes on a single branch from the leader. This behavior leads to an increased amount of migrations that could be reduced if the tree were more balanced.

The nodes can also take on any of the three roles found in typical publish/subscribe networks, even at the same time. This allows for the creation of a network that instead of behaving like a common multicast scenario. Thus, we can create a network where all the nodes are able to communicate with each other if each node has the three roles at the same time and a node creates a subscription with its own identifier.

Chapter 5

Conclusions

5.1 Conclusions

In this dissertation we have presented two protocols that allow broker migration in a fully mobile publish/subscribe system. The second one being an improved iteration of the first one. Both have been designed by extending the Phoenix protocol that allows subscriber and publisher migration. We have achieved the objective of having a system where network changes caused by mobility will not massively impair the performance of the network. As in Phoenix, we also try to optimize the messages caused by a migration and we are able to replay events to subscribers that took place while that subscriber was disconnected. However, due to the mobile and changing nature of the network we cannot make any promises on the order of those messages or even regarding whether a subscriber has received all of them or not.

Both protocols presented support the creation of partitions, each one composing an independent event delivery network that works for the subscribers and publishers connected to them. Thanks to the leader election algorithm used, in those partitions that merge together one of the two previous leaders will be chosen as the new leader designating a main partition and a secondary one. The main partition will be the one that keeps the leader and the secondary will effectively migrate into it. Thus we always have one direction for migrations. Neither protocol is dependent on the leader election algorithm we have chosen, any other algorithm that has similar properties can be used. Any algorithm that is able to create an acyclic connection graph with the nodes in the system, and keep it updated while the nodes are moving, can also be used. Different algorithms might also be used depending on the expected mobility of the system. A deployment in which we expect to have a backbone of static brokers while a few others are mobile could use a more traditional approach of trying to optimize or load balance the links used.

The first protocol, presented in Chapter 3 is a direct extension to Phoenix. It simplifies a few things, such as not storing messages with timestamps for event replay, but it offers broker mobility by creating a migration for the group of subscribers that are local to that broker. This protocol though simple, also has some drawbacks. The messages it sends can be very large since they might contain the full routing table of a broker. And, due to the cascading nature of the forced migrations, the protocol needs more messages than the optimum for notifying the migration. In some cases it might not be possible to completely eliminate a subscriber from the system, leaving behind phantom subscriptions.

The second approach, which we have decided to call Mobile Fault Tolerant Publish/Subscribe, or MFT-PubSub for short, is the main contribution of this dissertation. MFT-PubSub maintains the concept of using the brokers as a proxy for subscriber migration whenever a broker migrates, by simulating the migration of the subscribers connected to it. However, it improves its efficiency with respect to the first approach. We have added a timestamp to each message sent by a subscriber and this allows us to know which one of any two brokers has received the latest message from a subscriber. Using this timestamp, together with the number of hops a broker needs to reach the subscriber, we can optimize the migration of the brokers. Each time a broker migrates, it compares those values with the new connected broker and they exchange messages until both have an updated routing table. Furthermore this protocol allows any device on the network to take on any combination of the roles of a broker, subscriber or publisher. We can also create a filter that matches messages sent directly to an specific subscriber, this way we can create a network in which we keep delivery routes between any two nodes. And those same routes will be updated each time a topology change occurs.

MFT-PubSub has been validated by simulation in the Castalia framework. We have compared the performance of our protocol, with respect to mobility and number of devices it supports, against AODV. We improve on the message delivery rate that AODV offers though the performance is reduced on bigger networks. The number of messages needed also increases linearly with the network size allowing better scalability than AODV.

5.2 Future work

The protocols described in this dissertation can be extended and improved in several ways. As mentioned before, different mobility patters might reduce the requirements of migrations, allowing to optimize the number of messages needed for the migration.

Small changes to the leader election used might also have an impact in the way migrations are detected. By using the communication tree we are ignoring better delivery paths. If we introduce the concept of mobility to the creation of the spanning tree, we might have a network where devices are aware of the speed at which they are moving. Using this information, we might automatically create a backbone of slower nodes where links are more stable and try to prevent migrations from happening inside that backbone. Subscribers might also be more interested in joining those stable brokers than simply having the shortest path to the leader which is the one used in the current implementation. Trying to balance the spanning tree is also another point of improvement.

Also, since a device might have all three roles of a publish/subscribe network at the same time we can create a solution for multicast communication in a mobile environment.

References

[1] CC2420 chip datasheet. `https://www.ti.com/lit/ds/symlink/cc2420.pdf?ts=1611489151106`. [Online; accessed 22-January-2021].

[2] S. Akkermans, R. Bachiller, N. Matthys, W. Joosen, D. Hughes, and M. Vucinic. Towards efficient publish-subscribe middleware in the IoT with IPv6 multicast. In *2016 IEEE International Conference on Communications, ICC 2016, Kuala Lumpur, Malaysia, May 22-27, 2016*, pages 1–6. IEEE, 2016.

[3] B. Badrinath, A. Acharya, and T. Imieliński. Impact of mobility on distributed computations. *ACM SIGOPS Operating Systems Review*, 27(2):15–20, 1993.

[4] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. Efficient Publish/Subscribe Through a Self-Organizing Broker Overlay and its Application to SIENA. *Comput. J.*, 50(4):444–459, 2007.

[5] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg. Content-Based Publish-Subscribe over Structured Overlay Networks. In *25th International Conference on Distributed Computing Systems (ICDCS 2005), 6-10 June 2005, Columbus, OH, USA*, pages 437–446. IEEE Computer Society, 2005.

[6] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. 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.

[7] K. Beckmann and M. Thoss. A wireless sensor network protocol for the OMG Data Distribution Service. In *Proceedings of the 10th International Workshop on Intelligent Solutions in Embedded Systems, WISES 2012, Klagenfurt, Carinthia, Austria, July 5-6, 2012*, pages 45–50. IEEE, 2012.

[8] A. Boulis. Castalia: revealing pitfalls in designing distributed algorithms in wsn. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 407–408, 2007.

[9] I. Burcea, H. Jacobsen, E. de Lara, V. Muthusamy, and M. 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.

[10] U. Burgos, U. Amozarrain, C. Gómez-Calzado, and A. Lafuente. Routing in Mobile Wireless Sensor Networks: A Leader-Based Approach. *Sensors*, 17(7):1587, 2017.

[11] H. Cam, O. K. Sahingoz, and A. C. Sonmez. Wireless Sensor Networks Based on Publish/Subscribe Messaging Paradigms. In J. Riekki, M. Ylianttila, and M. Guo, editors, *Advances in Grid and Pervasive Computing - 6th International Conference, GPC 2011, Oulu, Finland, May 11-13, 2011. Proceedings*, volume 6646 of *Lecture Notes in Computer Science*, pages 233–242. Springer, 2011.

[12] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. *Wireless Communications and Mobile Computing*, 2(5):483–502, 2002.

[13] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications. *IEEE Trans. Software Eng.*, 29(12):1059–1071, 2003.

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

[15] A. Casteigts. JBotSim: a tool for fast prototyping of distributed algorithms in dynamic networks. In *Proceedings of the 8th International Conference on Simulation Tools and Techniques*, pages 290–292. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2015.

[16] M. Castro, P. Druschel, A. Kermarrec, and A. I. T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, 2002.

[17] C. Chen, H.-A. Jacobsen, and R. Vitenberg. Reinforce your overlay with shadows: Efficient dynamic maintenance of robust low fan-out overlays for topic-based publish/subscribe under churn. Technical report, Tech. rep., University of Toronto, University of Oslo, 2012.

[18] C. Chen, R. Vitenberg, and H.-A. Jacobsen. Omen: Overlay mending for topic-based publish/subscribe systems under churn. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 105–116, 2016.

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

[20] J. P. de Araujo, L. Arantes, E. P. Duarte, L. A. Rodrigues, and P. Sens. VCube-PS: A causal broadcast topic-based publish/subscribe system. *Journal of Parallel and Distributed Computing*, 125:18–30, 2019.

[21] A. Detti, D. Tassetto, N. B. Melazzi, and F. Fedi. Exploiting content centric networking to develop topic-based, publish–subscribe MANET systems. *Ad hoc networks*, 24:115–133, 2015.

[22] A. M. Dominguez, T. Robles, R. Alcarria, and E. Cedeño. A Hot-topic based Distribution and Notification of Events in Pub/Sub Mobile Brokers. *Network Protocols & Algorithms*, 5(1):90–110, 2013.

[23] C. Esposito, M. Platania, and R. Beraldi. Reliable and Timely Event Notification for Publish/Subscribe Services Over the Internet. *IEEE/ACM Transactions on Networking*, 22(1):230–243, Feb 2014.

[24] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[25] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. Supporting Mobility in Content-Based Publish/Subscribe Middleware. In M. Endler and D. C. Schmidt, editors, *Middleware 2003, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003, Proceedings*, volume 2672 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2003.

[26] C. Gómez-Calzado, A. Lafuente, M. Larrea, and M. 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.

[27] C. Gündogan, P. Kietzmann, T. C. Schmidt, and M. Wählisch. HoPP: Robust and Resilient Publish-Subscribe for an Information-Centric Internet of Things. In *43rd IEEE Conference on Local Computer Networks, LCN 2018, Chicago, IL, USA, October 1-4, 2018*, pages 331–334. IEEE, 2018.

[28] A. Hakiri, P. Berthou, A. S. Gokhale, and S. Abdellatif. Publish/subscribe-enabled software defined networking for efficient and scalable IoT communications. *IEEE Communications Magazine*, 53(9):48–54, 2015.

[29] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz. Meeting IoT platform requirements with open pub/sub solutions. *Annales des Télécommunications*, 72(1-2):41–52, 2017.

[30] Y. Huang and H. 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.

[31] Y. Huang and H. Garcia-Molina. Publish/Subscribe Tree Construction in Wireless Ad-Hoc Networks. In M. Chen, P. K. Chrysanthis, M. Sloman, and A. B. Zaslavsky, editors, *Mobile Data Management, 4th International Conference, MDM 2003, Melbourne, Australia, January 21-24, 2003, Proceedings*, volume 2574 of *Lecture Notes in Computer Science*, pages 122–140. Springer, 2003.

[32] Y. Huang and H. Garcia-Molina. Publish/Subscribe in a Mobile Environment. *Wireless Networks*, 10(6):643–652, 2004.

[33] U. Hunkeler, H. L. Truong, and A. J. Stanford-Clark. MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks. In S. Choi, J. Kurose, and K. Ramamritham, editors, *Proceedings of the Third International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE 2008), January 5-10, 2008, Bangalore, India*, pages 791–798. IEEE, 2008.

[34] Z. Jerzak. *XSiena: The Content-Based Publish/Subscribe System*. PhD thesis, Dresden University of Technology, 2009.

[35] S. Li, L. Da Xu, and S. Zhao. The internet of things: a survey. *Information Systems Frontiers*, 17(2):243–259, 2015.

[36] J. E. Luzuriaga, J. C. Cano, C. Calafate, P. Manzoni, M. Perez, and P. Boronat. Handling mobility in IoT applications using the MQTT protocol. In *2015 Internet Technologies and Applications (ITA)*, pages 245–250. IEEE, 2015.

[37] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, 2002.

[38] G. Mühl, A. Ulbrich, K. Herrmann, and T. Weis. Disseminating Information to Mobile Clients Using Publish-Subscribe. *IEEE Internet Computing*, 8(3):46–53, 2004.

[39] V. Muthusamy, M. Petrovic, and H. Jacobsen. Effects of routing computations in content-based routing networks with mobile data sources. In T. F. L. Porta, C. Lindemann, E. M. Belding-Royer, and S. 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.

[40] C. Perkins, E. Belding-Royer, and S. Das. Rfc3561: Ad hoc on-demand distance vector (aodv) routing, 2003.

[41] T. Rausch, S. Nastic, and S. Dustdar. Emma: Distributed qos-aware mqtt middleware for edge computing applications. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 191–197. IEEE, 2018.

[42] D. S. Rosenblum and A. L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In M. Jazayeri and H. 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.

[43] P. Salehi, C. Doblander, and H.-A. Jacobsen. Highly-available Content-based Publish/Subscribe via Gossiping. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS 2016, pages 93–104, New York, NY, USA, 2016. ACM.

[44] Z. Salvador. *Client Mobility Support and Communication Efficiency in Distributed Publish/Subscribe.* PhD thesis, University of the Basque Country UPV/EHU, Spain, 2012.

[45] Z. Salvador, A. Alzua, M. Larrea, and A. Lafuente. Mobile XSiena: towards mobile publish/subscribe. In J. Bacon, P. R. Pietzuch, J. Sventek, and U. Çetintemel, editors, *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, pages 91–92. ACM, 2010.

[46] Z. Salvador, A. Lafuente, and M. Larrea. Design and Evaluation of a Publish/Subscribe Framework for Ubiquitous Systems. In K. Zheng, M. Li, and H. Jiang, editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services - 9th International Conference, MobiQuitous 2012, Beijing, China, December 12-14, 2012. Revised Selected Papers*, volume 120 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 50–63. Springer, 2012.

[47] Z. Salvador, M. Larrea, and A. 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.

[48] T. R. Sheltami, A. A. Al-Roubaiey, and A. S. H. Mahmoud. A survey on developing publish/subscribe middleware over wireless sensor/actuator networks. *Wireless Networks*, 22(6):2049–2070, 2016.

[49] G. Siegemund and V. Turau. A Self-Stabilizing Publish/Subscribe Middleware for IoT Applications. *ACM Transactions on Cyber-Physical Systems*, 2(2):12:1–12:26, 2018.

[50] S. Tarkoma. *Efficient content-based routing, mobility-aware topologies, and temporal subspace matching.* PhD thesis, University of Helsinki, Finland, 2006.

[51] Y. Tekin and O. K. Sahingoz. A Publish/Subscribe messaging system for wireless sensor networks. In *Sixth International Conference on Digital Information and Communication Technology and its Applications, DICTAP 2016, Konya, Turkey, July 21-23, 2016*, pages 171–176. IEEE, 2016.

[52] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and , 2008.

[53] A. V. Ventrella, G. Piro, and L. A. Grieco. Publish-subscribe in mobile information centric networks: Modeling and performance evaluation. *Computer Networks*, 127:317–339, 2017.

[54] A. Whitmore, A. Agarwal, and L. Da Xu. The internet of thingsa survey of topics and trends. *Information systems frontiers*, 17(2):261–274, 2015.

[55] G. Xylomenos, X. Vasilakos, C. Tsilopoulos, V. A. Siris, and G. C. Polyzos. Caching and Mobility Support in a Publish-Subscribe Internet Architecture. *IEEE Communications Magazine*, 50(7):52–58, 2012.

[56] Y. Zhao and J. Wu. Building a reliable and high-performance content-based publish/subscribe system. *Journal of Parallel and Distributed Computing*, 73(4):371–382, 2013.