

INF-3320: Middleware

Supporting Mobility in Content-Based
Publish/Subscribe Middleware

Christer A. Hansen

Abstract

Publish/Subscribe (pub/sub) is considered to be a scalable loosely coupled middleware architecture. The model with including support for mobility fits well for mobile devices since they often are on the move and disconnected. Unfortunately most of the pub/sub systems do not provide enough support for mobility. The main reason for this is because they are developed for static wired environments where mobility is not an issue. Mobility becomes important when we want the system to handle physical relocation and disconnections of the clients. We distinguish between two orthogonal types of mobility: the system-centric physical mobility, and the application-centric logical mobility. In this paper we discuss various solutions for mobility. We separate the good solutions from the bad, and address the limitations and weaknesses.

1 Introduction

The main concept in a publish/subscribe (pub/sub) system is to let the clients define their own interests in form of creating subscriptions. The producer does not need to know any information about the consumers (e.g. IP address and port number), everything is handled by the middleware itself. This loose coupling between the producers and the consumers is the main advantage in the pub/sub model.

Mobility is an important topic in pub/sub systems. We split mobility into two different types. The first type is called physical mobility. This type is about handling temporary disconnections. The second type is called logical mobility and is about location-awareness. In this paper we will focus on these two types of mobility. We will take a closer look on how real pub/sub systems provide mobility, and whether they support logical or physical mobility.

This paper is structured as follows: First we will describe the issues concerning mobility in Section 2. We provide some basic terminology on content-based pub/sub systems in Section 3. Then we will introduce some real-life systems that are concerning mobility in Section 4. Section 5 concludes the paper.

2 Mobility issues

In this section we briefly introduce some issues pub/sub systems should concern and support. As we already know there are two types of mobility, physical and logical.

We address some physical mobility issues:

- The system must be able to deliver notifications that are published while a client is disconnected.
- The system must avoid letting the clients get duplicates of notifications.
- The system must also let the client connect through different access points so that there are no geographical restrictions in the system.
- The client must be able to not worry about manually getting hold of lost notifications that was published while it was disconnected.

Here are some common logical mobility issues:

- The system must be able to deliver different notifications based on where the client is located (e.g. in a building).
- The system should be able to predict where the next location will be, and the system should not let the client wait a long time at a new location before getting its notifications.

3 Terminology

A pub/sub system consists of producers (publishers), consumers (subscribers), and the underlying middleware system. A process can act as both a producer and a consumer. When it acts as a producer it generates messages called notifications. A notification is a message containing information about an event that has occurred. The notifications are handled by the underlying notification service (middleware) which propagates the notifications to those consumers that have registered a matching subscription. The middleware is responsible for storing the subscriptions and forwarding the notifications when they are published.

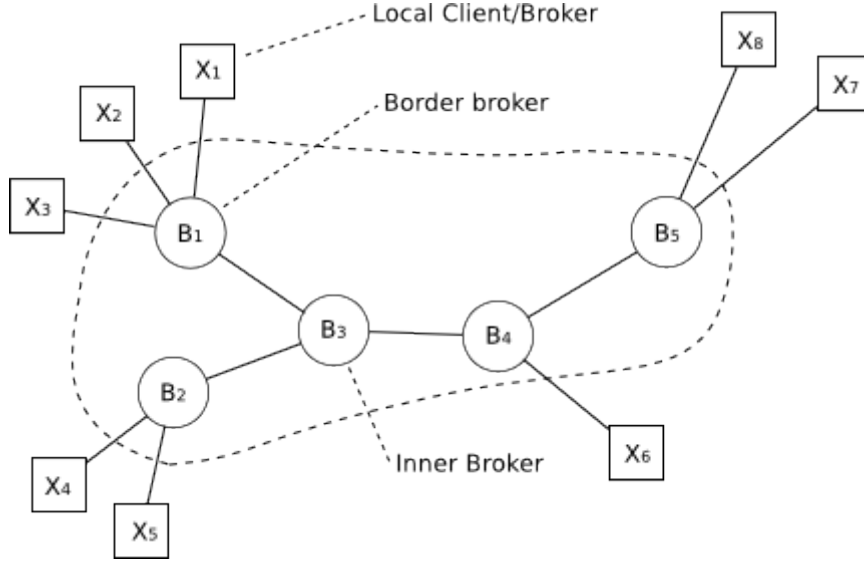


Figure 1: Network overview of a pub/sub system.

The communication interface consists of four primitives; *pub*, *sub*, *unsub* and *notify* [4]. When a client wants to subscribe it uses the *sub* primitive (e.g *sub(weather="oslo")* tells the system that weather forecasts concerning Oslo must be forwarded to the subscribing client). The *pub* primitive is used to publish notifications. The *unsub* primitive is used for removing earlier subscriptions. The last primitive, *notify*, is used by the middleware system to deliver notifications to the consumers.

Different languages can be used for specifying the subscriptions in a pub/sub system, such as subject- and type-based languages [8, 1, 3]. But the most flexible scheme is offered by content-based filtering [7]. In the content-based approach the whole notification is used in the filter process. This makes the system more flexible because the content is extracted from the notifications and compared to the subscriptions by using special algorithms. Filters are used for defining subscriptions. They are stateless boolean functions that are applied to the notification. If a notification matches a filter the boolean function returns true, and the notification is forwarded to the client.

The pub/sub middleware systems we describe in this paper are distributed. Scalability and better support for mobility are two advantages by having the system distributed. The communication overview of a pub/sub system is given by a graph,

see Figure 1, which is assumed to be acyclic and connected. The system consists of brokers and clients. The edges are communication links (e.g. TCP connections). Brokers are processes that route notifications from the publisher to the subscribers. There are three different types of brokers: Local brokers are the clients access point to the middleware, and are part of the communication library loaded into the clients. A local broker can only be connected to one border broker. The border brokers maintain connections to the local brokers (i.e. the clients). Inner brokers are connected to other inner or border brokers, and can not be connected to any clients [4].

Each broker maintains a routing table which determines where the notifications should be forwarded. Each tuple in the table is a pair (F, L) containing a filter and a link to where the subscription was sent from [4]. The link may be to a local, inner, or border broker. The routing tables are maintained to hold information about active consumers and their subscriptions. When a notification reaches a broker it parses through its routing table looking for matching filters. If matching filters are found the notification will be forwarded to another broker.

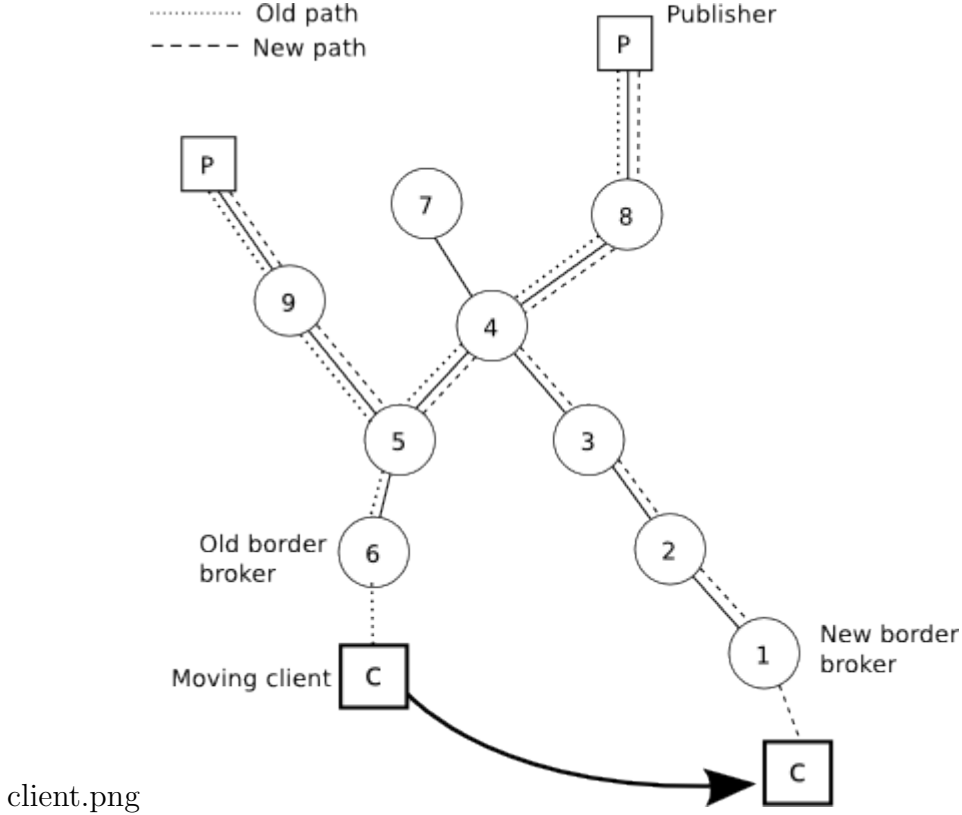
4 Mobility

There are two types of mobility, physical and logical [4]. Physical mobility concerns about temporary disconnections and when clients change border brokers. Logical mobility is about the middleware's ability to send location-dependent notifications to their clients.

4.1 Physical mobility

Mobile clients often have the need to disconnect from the network for various reasons. The cause may be administrative, a change in geographical location, or power saving restrictions. E.g. when a client move to a new geographical location it will most likely not be served by the same border broker, but instead a new one, and while the client was disconnected it probably missed some notifications. The system must be able to deliver the same quality of service even though the client has been disconnected [4]. The same interface must be offered if the client changes to another access point. The pub/sub system is also responsible for delivering all the notifications in the correct order to the client even if intermittent disconnections appear.

A solution that handles lost notifications while disconnected is suggested by [4],



client.png
 Figure 2: Network overview of a pub/sub system.

and used on top of the Rebeca system. They introduce a “virtual counterpart” that stays at the last known location until a broker at a new location claims responsibility for the client. The notifications that are published while the client is disconnected are collected by the “virtual counterpart”, and then transferred to the client at its new location. A more detailed example is illustrated in Figure 2. In the figure we assume that client C is moving from the location at broker B_6 to B_1 . When the client finds out about the new location and new broker it automatically re-issues a subscription with the last received sequence number for this subscription. When broker B_1 notices that the client has moved from another broker it starts the relocation process. The goal of the relocation process is to renew the delivery paths from the producers to the consumers. When the client subscribes at the new location the brokers propagate the subscription through B_2 and B_3 to broker B_4 . At B_4 the old and the new path from the producers to the consumers meet. The broker knows this by inspecting

the routing table. B_4 notices that it has an old entry for the same subscription just received, therefore it sends a fetch request along the path to B_6 (the old broker the client was connected to). Notifications published after this subscription will be routed along the new path (dashed line). After B_6 received the fetch request it will start emptying the buffer inside the “virtual counterpart” by sending a message with a replay of all notifications into the direction of B_4 . B_4 will propagate the replay towards the client. Eventually the client will then receive all the notifications that was published while it was disconnected.

Another pub/sub middleware system called STEAM (Scalable Timed Events And Mobility) was built for supporting the mobile computing domain [6]. It was developed to fit IEEE 802.11-based wireless local area networks (WLAN). The system was also developed specifically with the traffic management application domain in mind. The STEAM system supports three types of event filters, which are subject, proximity and content filter. The proximity filter is the characteristic one in this system, it is used for propagating notifications to consumers within a given geographical area. The subject and proximity filters are stored at the producer while the content filter is located at the consumer and will be evaluated there.

The JEDI system supports mobility by the functions `moveIn` and `moveOut` [2]. The system is distributed and messages are propagated through dispatchers or buses. When a client, or a so called active object, uses the `moveOut` function it asks the dispatcher to save all the events it should have received if it stayed connected. When the active object calls the `moveIn` function it will receive all the events generated during the disconnection that matches its subscriptions.

4.2 Logical mobility

Mobile clients have the need to receive notifications based on their location. In logical mobility we assume that mobile clients do not change from one broker to another, but instead they stay connected to the same one. The middleware must be able to deliver notifications based on the clients location and its subscriptions. Several designs are suggested and most of them concern predicting where the client may be located in the future, and how to cache notifications close to the client to avoid starvation and maintain high responsiveness.

[4] suggests a way to implement location awareness in a pub/sub middleware by changing the filters in the broker network. E.g if a client moves from an old location to a new location it must declare the new location by sending a message to its broker.

The broker will then change the location-dependent part of the filter by updating its routing table. Broker B_i will then send a message with the new location to B_{i+1} instructing it to change its filter too. This will be done until every broker knows about the client's new location.

They also suggest an approach for caching future locations the client may be located at by using a movement graph. The graph illustrates which locations the client can be located at after one movement step. E.g. if the client is located at location a it can be located at a , b , or c after one movement step. It is assumed that changing one subscription at a broker takes about as long as a consumer stays at one particular location, this means that brokers farther away from the subscriber need to subscribe to more locations. This method of propagating notifications is some sort of "restricted flooding", e.g. all notifications may reach a certain broker but from there only a subset of all the notifications will be forwarded.

An approach for pre-subscription and better responsiveness of the Rebeca system is suggested by [5]. This approach uses both physical and logical mobility extensions. The main idea behind this approach is to set up virtual clients at different brokers the client may connect to in the future. Having virtual clients running at brokers around the physical client at any given time reduces the time it takes to bootstrap at a new broker. The virtual clients are subscribing for location dependent notifications and for the client's interests. By doing this the virtual client builds up a buffer over time. When a client moves from an old broker to a new one, the virtual client will send all its buffered notifications to the client. A lot of time is saved for the client because it does not need to start subscribing when it reaches the new broker, this is already handled by the virtual client. Rarely published notifications that are received and buffered by the virtual client may also save the client for a lot of waiting.

5 Conclusion

Pub/sub systems are often developed to serve in static environments. What we have seen is that systems often only support a portion of mobility. In some systems mobility may not even be considered at all because it's not needed. In this paper we have looked at the dynamic and applicable system called Rebeca. We have taken a closer look at how the Rebeca system works and how physical and logical mobility is implemented and handled by the middleware system.

5.1 Physical mobility

A good solution for handling disconnections is developed for the Rebeca system [4]. In their approach a virtual counterpart is introduced to buffer the notifications that are published while the client is disconnected. When the client re-connects to the network the virtual counterpart will send the buffered notifications to the client. This solution is built into the middleware system so the client does not have to save its own state or notify the middleware of its departure. In the JEDI system another approach is used, mobility is controlled by the application itself [2]. When an application is to disconnect from the system it must call a `moveOut` function to save its own state. If the network connection breaks down, or the application machine fails, the client will not be able to tell the middleware that its about to disconnect and that it wants buffering on its subscriptions. This is a major drawback of the JEDI system.

STEAM is a static system built for traffic management [6]. The system relies on proximity-based group communication. Mobility is an important issue that needs to be handled with care, specially since the system is based on wireless communication. Failed connections and unavailable entities are not handled by the middleware system. This makes the system vulnerable and insecure.

There are several approaches to handle physical mobility, but the most promising approach is the one designed for the Rebeca middleware system. The middleware system finds out about disconnected clients and automatically caches the notifications while the clients are disconnected.

5.2 Logical mobility

Logical mobility is implemented to support location-awareness in a pub/sub middleware. When a client moves to a new location and is connected to the same broker, it needs to notify the broker network about the new location so that the subscriptions can be changed. If the filters in the broker network remains unchanged while the client moves to a new location, the client is unable to receive notifications based on where its currently located. A solution for this problem is implemented into the Rebeca middleware system [4].

Making the system responsive for the clients is also an important issue in logical mobility. For instance when a client moves from an old location to a new location starvation and waiting must be avoided. Solutions that handles this issue is suggested by [4, 5].

References

- [1] J. Bates, J. Bacon, K. Moody, and M. Spiteri. Using events for the scalable federation of heterogeneous components. In *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, pages 58–65, 1998.
- [2] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001.
- [3] P. T. Eugster, R. Guerraoui, and C. H. Damm. On objects and events. In *in Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, pages 254–269. ACM Press, 2001.
- [4] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. In *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 103–122, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
- [5] L. Fiege, A. Zeidler, F. C. Grtner, and S. B. Handurukande. Dealing with uncertainty in mobile publish/subscribe middleware. In *In 1st International Workshop on Middleware for Pervasive and Ad-Hoc Computing*, pages 60–67, 2003.
- [6] R. Meier and V. Cahill. Steam: Event-based middleware for wireless ad hoc network. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 639–644, Washington, DC, USA, 2002. IEEE Computer Society.
- [7] G. Mühl. Generic constraints for content-based publish/subscribe. In *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 211–225, London, UK, 2001. Springer-Verlag.
- [8] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. *SIGOPS Oper. Syst. Rev.*, 27(5):58–68, December 1993.