An Upgrade Mechanism Based on Publish/Subscribe Interaction[1]

M.R.V. Chaudron [a]
[a]*Eindhoven University of Technology*
The Netherlands
m.r.v.chaudron@tue.nl

F. van de Laar [a, b]
[b]*Philips Research Laboratories*
The Netherlands

Abstract

We advocate the use of publish/subscribe as an interaction style for upgradeable component-based systems. We present a software architecture based on this style. We describe some key design issues and their rationale.

1. Introduction

The world around us is constantly changing, and we want software systems to be as dynamical as the real world. Therefore, it is important that software systems can be changed easily. A major contribution in the development of changeable systems is component-based engineering of software [4]. Topic of our research is to investigate how components can be upgraded dynamically.

A well accepted design principle of component-based design is to minimize coupling between components. The emphasis/focus of this principle is often on the functionality of components. However, to facilitate upgrading of components in systems that are in operation, the coupling between non-functional aspects of components should also be minimized [1].

The dominant interaction style in current component models is request-response, e.g. (remote) procedure call. However, the publish/subscribe style (see [2] for an overview) induces less coupling than request-response by providing a decoupling in space (interacting components do not have to know each other) and time (publishers and subscribers do not have to interact simultaneously).

In this paper we describe a software architecture that is based on publish/subscribe as interaction style between components and investigate the issues that arise in replacing components in systems based on this model.

First, we present an overview of the system architecture. Next, we define our goals. After that we describe some key design issues and illustrate some of the upgrading-scenarios.

2. Goals

The aim of our research is to develop a software architecture that has the following properties:

- Transparent replaceability of parts of the system: upgrading of parts of the system should be transparent to other parts of the system
- Robustness of the system

3. System architecture

We define a system as a set of components together with a shared infrastructure. Our infrastructure consists of one configuration manager (CM) and zero or more brokers (see Figure 1). The key idea is that the components have a dependency on the infrastructure, but not on each other.

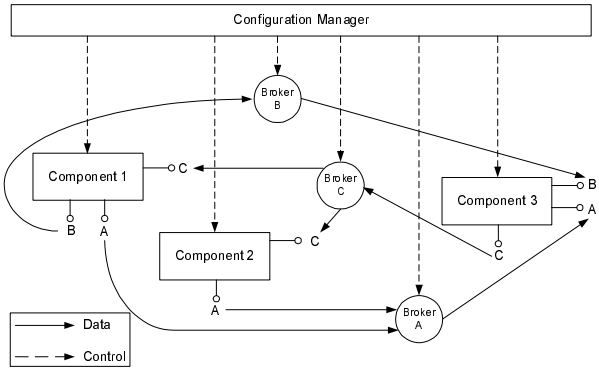

Figure 1: Example system

A component (see Figure 2) consists of some core code that implements the actual functionality of the component, a set of publish-ports and subscribe-ports, a heartbeat (HB) interface (explained below) and a configuration interface. The core code uses the

[1] This research is supported by the ITEA Project Robocop.

publishers and subscribers to interact with other components. The configuration and heartbeat interface are used for interaction with the configuration manager.

A broker is responsible for relaying messages received from publishers to subscribers, i.e. the components interact with each other via brokers.

The CM handles all issues regarding the configuration: starting and stopping of the system and the replacing of components and brokers. Furthermore, it plays a role in the robustness mechanism for the system (explained below).

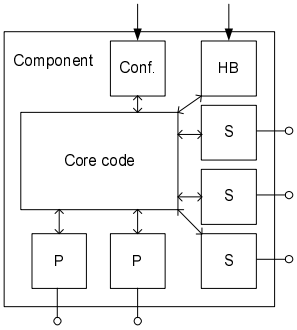

Figure 2: Example component

4. Design Issues

This section discusses key design issues. The main focus is on the binding method. The consequences for robustness and the way that brokers can be instantiated or replaced are discussed as an outcome of the binding choice.

4.1 Robustness

By robustness we mean that if one component or broker crashes or is removed from the system, the rest of the system keeps on running. Our architecture provides two mechanisms to achieve this. Firstly, we allow multiple brokers. These can be used to relay different types of data. This reduces the impact of replacing a broker to components that use this type. Secondly, the CM periodically checks all components and brokers if they are still alive (using the heartbeat interface). If a component or broker is down, the CM reinstantiates the crashed component or broker.

4.2 Brokers

We identified two possibilities for the moment in time when a broker is created. Firstly, whenever a publisher or subscriber is started. This means that a broker is always started. Secondly, a broker could be started when there are both subscribers and publishers. In this case the broker is only started if it is actually useful. E.g. if there are no publishers available, a broker is useless.

4.3 Binding

The only binding that we focus on in our system is the binding between components and brokers. This can be done in two ways. First, using first-party binding, in which the component itself takes the responsibility for binding to brokers. Second, using third-party binding, in which the CM takes care of binding the components to the brokers

4.3.1 First-party binding. If first-party binding is used, the coupling between the components and the brokers is strict. This could hinder the replacement of brokers by the CM. On the other hand, it relieves the CM of the broker creation task. It also means that the components should take care of crashing brokers themselves.

4.3.2 Third-party binding. If third-party binding is used, the CM binds the components to the brokers. This decreases the dependencies between the components and the brokers, hence simplifying the replacing procedure for the CM.

Third-party binding makes it possible for the CM to postpone the creation of the brokers until they are actually needed. If a new component is instantiated, the CM checks on what topics it wants to subscribe to and on what topics it will start publishing. On the basis of this information, the CM can decide whether or not a broker is needed: if there are subscribers but no publishers, no broker is needed. The CM could inform the components about this, so that they can anticipate on it.

4.3.3 Choice. We chose for third-party binding, because it provides more flexibility, especially when replacing a running broker. In addition, third-party binding makes it easier for the CM to replace crashed components and brokers.

5. Upgrading

Upgrading boils down to replacing a component or broker with another one. We only look at replacing components and brokers, not the CM.

We will discuss two examples: replacing a running component and replacing a running broker. First, we explain the replacement of a running component (see Figure 3). When the CM wishes to replace a component, it first creates a new component. Then it sends the UnSetBroker command (for every broker used by that component) to the to-be-replaced component. That

component then takes care of unsubscribing all of its subscribers (by issuing a UnSubscribe command to the broker) and unsetting all the brokers with its publishers. After that, the CM informs the new component about the presence of a broker with the SetBroker method (for all needed brokers).

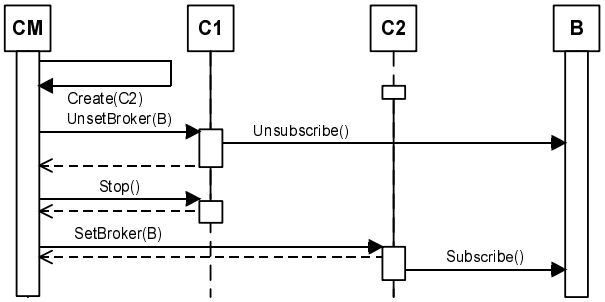

Figure 3: Replacing a component

The second example shows the replacement of a running broker (see Figure 4). First, the CM sends the UnSetBroker command to all components using that broker. The components themselves take care of unsubscribing their subscribers as well as stopping their publishers from publishing to that broker. Then the CM stops the broker. After that, the CM creates the new broker and binds all components that wish to use that broker to it (through the SetBroker command).

Note that there are two choices when replacing a broker or component. The new broker or component can be created before or after the replacing of the old broker or component is stopped. E.g. if there is a resource restriction, one can choose to create the new broker after the old one is deleted.

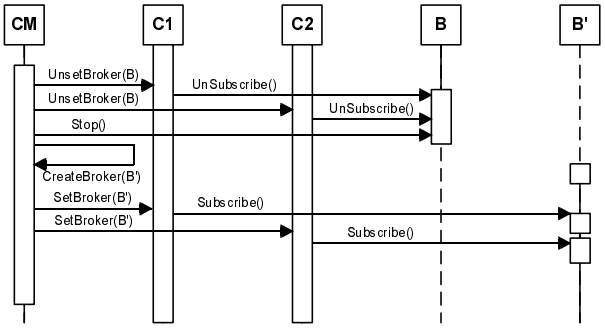

Figure 4: Replacing a broker

6. Concluding remarks and future work

We investigated publish/subscribe as an architectural style for component interaction. Conceptual considerations suggest that its looser coupling facilitates dynamic replaceability.

Currently, we are developing a prototype. Initial experiments using this prototype show the principal feasibility.

We did not consider scenarios involving components that have state and have to transfer that state to their replacement. Also, we do not consider a crashing CM. Both of these are future work.

Based on other experience described in [3] we will look in more detail into performance issues. The potential trade-off between performance and upgradability is subject of ongoing research.

In section 3 we defined the goals we wanted to achieve: transparent replacing of parts of the system (components and brokers) and robustness of the system. The latter is achieved by using the heartbeat interface to detect component and broker failure and by using multiple brokers. The transparency of replacement of components is established by using the decoupling provided by the brokers: a component does not notice the replacement of another component, other than it might observe that the publishing of data discontinues for a moment in time (if the component can notice this at all).

The replacing of a broker is less transparent: the involved components are notified that a broker is removed and added again (although they can not determine the difference between a broker being replaced or a broker that has crashed).

References

[1] M.R.V. Chaudron and E. de Jong, “Components are from Mars”, *Workshop on Parallel and Distributed Real Time Systems 2000*, LNCS 1800

[2] P. Th. Eugster, P. Felber, R. Gerraoui, A.-M. Kermarrec, *“The Many Faces of Publish/Subscribe”*, Technical Report MSR-TR-2001-104, Microsoft Research Laboratories, Cambridge, UK, Jan 2001

[3] R. Rajkumar, M. Gagliardi, L. Sha, “The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation”, *IEEE Proceedings of the Real-Time Technology and Applications Symposium*, 1995

[4] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, ACM Press and Addison-Wesley, New-York, 1998