Interaction systems design and the protocol- and middleware-centred paradigms in distributed application development

João Paulo Almeida, Marten van Sinderen, Dick Quartel, Luís Ferreira Pires
{almeida, sinderen, quartel, pires}@cs.utwente.nl
Centre for Telematics and Information Technology, University of Twente
PO Box 217, 7500 AE Enschede, The Netherlands

Abstract

This paper aims at demonstrating the benefits and importance of interaction systems design in the development of distributed applications. We position interaction systems design with respect to two paradigms that have influenced the design of distributed applications: the middleware-centred and the protocol-centred paradigm. We argue that interaction systems that support application-level interactions should be explicitly designed, using the externally observable behaviour of the interaction system as a starting point in interaction systems design. This practice has two main benefits: to promote a systematic design method, in which the correctness of the design of an interaction system can be assessed against its service specification; and, to shield the design of application parts that use the interaction system from choices in the design of the supporting interaction system.

1 Introduction

In recent years, there has been a predominant use of middleware platforms in the development of distributed applications. Typical design methods based on the re-use of middleware platforms consist of partitioning an application into application parts and defining the interconnection aspects by defining interfaces between parts, e.g., by using object-oriented techniques and abstracting from distribution aspects.

As a consequence of this practice, designers have neglected the role of interaction systems design in the development of distributed applications, focusing on application part design and failing to identify application interaction aspects explicitly. The objective of this paper is to show the benefits and importance of the explicit design of interaction systems in the development of distributed applications, both when reusing middleware platforms and when following a protocol-centred approach to development.

The service concept has an important role in our approach. A service is a design that defines the behaviour of a system from an external perspective. We propose a design trajectory that starts with the specification of the service of an application interaction system. This practice has two main benefits: to promote a systematic design method, in which the correctness of the design of an interaction system can be assessed against its service specification, and; to shield the design of application parts from choices in the design of the supporting interaction system.

This paper is further structured as follows: Section 2 defines interaction systems and presents the service concept; Sections 3 and 4 characterize the protocol-centred and the middleware-centred paradigms respectively; Section 5 discusses the role of interaction systems design in both paradigms, and Section 6 illustrates the use of an application interaction system and its service specification in a design trajectory. Finally, section 7 presents our conclusions and outlines some future work.

2 Interaction Systems

A distributed system can be considered from two different perspectives: an integrated and a distributed perspective. The integrated perspective considers a system as a whole or black box. This perspective only defines what function a system performs for its environment. The distributed perspective defines how this function is performed by an internal structure in terms of system parts (which are also systems) and their relationships.

We call the integrated perspective of a system a *service* [11]. A service is a design that defines the observable behaviour of a system in terms of the interactions that may occur at the interfaces between the system and the environment, and the relationships between these interactions. A service does not disclose details of an internal organization that may be given to implementations of the system.

Since the concept of system is recursive, in the sense that a system part is a system in itself, the service concept can be applied recursively in system design. The recursive application of the service concept allows a designer to consider the behaviour of a system at different related decomposition levels. In general, the number of decomposition levels and the particular choices for decomposition depend on particular system requirements and objectives of a designer.

When interactions between system parts have to be explicitly designed, the concept of interaction system is introduced. An *interaction system* supports the set of related interactions between two or more systems parts [7]. An interaction system consists of parts of system parts and their means of interaction, as depicted in Figure 2.

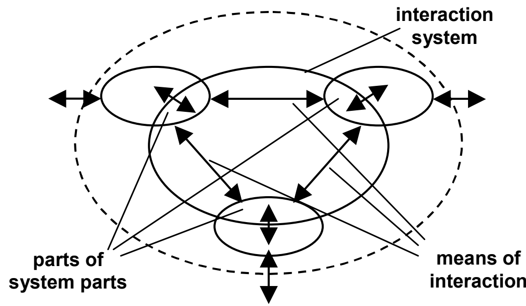

Figure 2 Interaction system from a distributed perspective

An interaction system is a system in itself, and therefore the external behaviour of an interaction system can be defined as a service, as depicted in Figure 3. The service specification serves as a starting point for the design of an interaction system that supports the service.

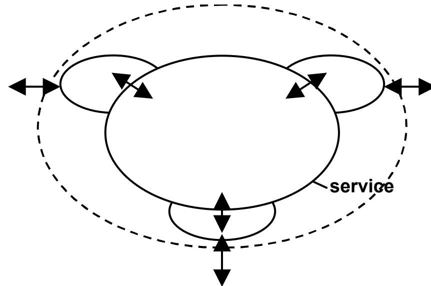

Figure 3 Interaction system from an integrated perspective

Interaction systems that satisfy basic communication needs between software components have been referred to as *connectors* in the software architecture literature [1].

3 Protocol-centred paradigm

In the protocol-centred paradigm, user parts interact locally with a *service provider*. A service provider consists of a composition of *protocol entities* and a *lower level service provider*, which interact in order to provide the required service to user parts. The model of the system to be built consists of user parts and, for each protocol layer, a collection of protocol entities and a lower level service provider. This model is depicted in Figure 4.

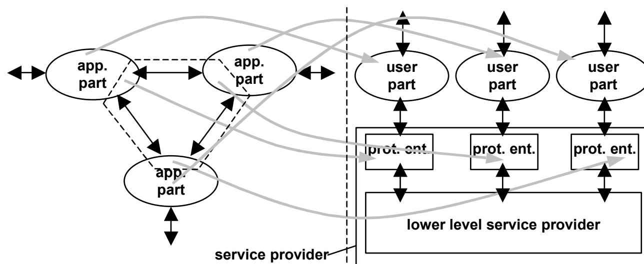

Figure 4 System in the protocol-centred paradigm

The lower level service provides physical interconnection and (reliable or unreliable) data transfer between protocol entities. Lower level services can support arbitrarily complex interaction patterns between the protocol entities, varying from connectionless data transfer (e.g., 'send and pray') to complex control facilities (e.g., handshaking with three-party negotiation).

Protocol entities communicate with each other by exchanging messages, often called Protocol Data Units (PDUs), through a lower level service. PDUs define the syntax and semantics for unambiguous understanding of the information exchanged between protocol entities. The behaviour of a protocol entity defines the service primitives between this entity and the service users, the service primitives between the protocol entity and the lower level service, and the relationships between these primitives. The protocol entities cooperate in order to provide the requested service [6].

Protocols can be defined at various layers, from the physical layer to the application layer. An application protocol defines distributed interactions that directly support the establishment of information values relevant to the application service users [7].

4 Middleware-centred Paradigm

In the middleware-centred paradigm, system parts interact through a limited set of interaction patterns offered by a middleware platform. The model of a distributed application to be built consists of the *middleware platform* and a collection of interacting parts, often called *objects* or *components*. This model is depicted in Figure 5.

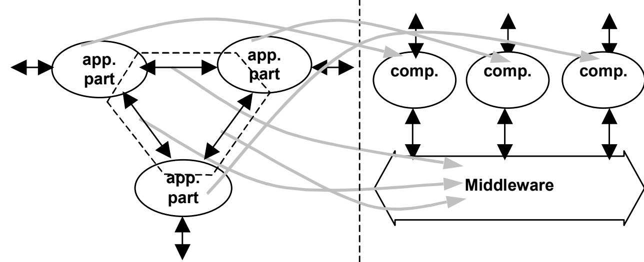

Figure 5 system in the middleware-centred paradigm

There are several different types of middleware platforms, each one offering different types of interaction patterns between objects or components. The middleware-centred paradigm can be further characterized according to the types of interaction patterns supported by the platform. Examples of these patterns are *request/response*, *message passing* and *message queues*. Examples of available middleware platforms are CORBA/CCM [4, 5], .NET [3] and Web Services.

The middleware-centred paradigm promotes the reuse of the middleware infrastructure, facilitating the development of distributed applications. Furthermore, middleware infrastructures provide facilities to define application-level information attributes and to exchange values of these attributes through the supported interaction patterns.

An interesting observation with respect to the middleware-centred paradigm is that it is somehow dependent on the protocol-centred paradigm: interactions between application parts are supported by the middleware, which 'transforms' the interactions into (implicit) protocols, provides generic services that are used to make the interactions distribution transparent and internally uses a network infrastructure to accomplish data transfer [8].

Design methods based on the re-use of middleware platforms often consist of partitioning the application into application parts and defining the interconnection aspects by defining interfaces between parts (e.g., by using object-oriented techniques and abstracting from distribution aspects). The available constructs to build interfaces are constrained by the interaction patterns supported by the targeted platform. Examples of these constructs are *operation invocation*, *event sources* and *sinks*, and *message queues*. This structuring strategy emphasizes a decomposition level in which the interaction systems provided by the middleware platform are emphasized.

The predominance of this view implies that the choice of interaction patterns provided by a particular middleware platform directly influence the application structure. The design of the application is therefore platform-specific, not only in the sense that the design depends on particular technological conventions adopted by the middleware platform, but in the sense that the structure of the application depends on the set of interaction patterns provided.

5 Interaction Systems Design

Instead of defining the interconnection of application parts directly in terms of a protocol or in terms of the interaction systems provided by a middleware platform, it is possible to identify *application interaction systems* that support application-level interactions between application parts. Figure 6 illustrates the view of an application where an application interaction system is identified.

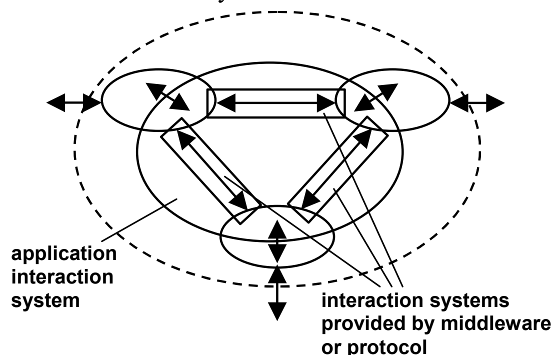

Figure 6 Application interaction systems

Whether or not the design of application interaction systems is considered explicitly depends on the application requirements and on the objectives of the designer [7]. In the following situations, interaction system design should be considered:

- if the relation between system parts is complex. In this case, proper attention should be given to the design of the relation between system parts. This is possible if this relation is made a separate object of design, i.e., if the interaction system of the system parts is considered separately. Consideration of the interaction system is possible at different abstraction levels in order to cope with the complexity of the relation. The interaction system provided by the middleware plays an important role at lower levels of abstraction.
- if it is easier to define a service than the behaviour of the system parts that interact. This may be the case if the functionality of the system parts is still in part unknown, or if the system parts are relatively complex because it must take account of the characteristics of the means of interconnection between the system parts.
- if it is more likely that interactions are changed than just the contributions to interactions by individual system parts. This is the case if several different middleware platforms are envisioned as alternatives to support the interactions. An interaction mechanism can only be replaced by another equivalent interaction mechanism if the relevant characteristics of the mechanism are clearly indicated in the design. This is naturally supported with interaction system design.
- if explicit attention to design choices that concern the effectiveness and efficiency of interactions is required. In this case, QoS aspects that are influenced by distribution aspects may be addressed separately with interaction system design.

A starting point in the design of an application interaction system is the specification of the application service, capturing the succinct description of the required application interaction system from an external perspective. The design of the application interaction system may, in principle, have any internal structure as long as it provides the required service. For example, it may make direct use of a data transport service as in a protocol approach. Nevertheless, we observe that the middleware leverages the reuse of a large building block that provides an interoperability architecture across programming languages, operating systems, network technologies and provides facilities to define application-level information attributes. Therefore, we argue that interaction systems provided by the middleware should also be considered for building application interaction systems.

A systematic interaction system design method based on the protocol-centred paradigm consists of: (i) defining the service to be supported in terms of the service primitives that occur at service access points, and the relationships between service primitives; and, (ii) decomposing this

service in terms of a structure of protocol entities and a lower level service. This resulting structure, which we call a *protocol*, has to be a correct implementation of the service. This can be assessed formally, if both the service and protocol are specified using some formal language.

A systematic interaction system design method based on the middleware-centred paradigm consists of defining (i) the service to be supported (as in the case for the protocol-centred paradigm) and, (ii) decomposing this service in terms of a structure of service components and the interaction systems provided by a middleware platform. This resulting structure has to be a correct implementation of the service. Again, this can be assessed formally, if both the service and its design (service components and platform) are specified using some formal language.

6 Example: Floor-control Service

In order to illustrate the use of an application interaction system and its service specification in a design trajectory, we introduce our running example, the *floor-control* problem. In this example, several application parts share a set of named resources. Each of these resources can only be used by a single application part at a time, and hence application parts have to coordinate their behaviours in order to ensure that there is no concurrent use of a resource. Subscribers are assumed to be cooperative, i.e., they will not use the resources indefinitely. In addition, no pre-emption of control over a resource is necessary.

6.1 Service Definition

We start with the definition of the *floor-control service*. The service relates the following interactions: `request`, `granted` and `free`. These interactions occur at the interfaces between the floor-control service and the subscribers. A result of the occurrence of each of these interactions is the establishment of the resource identification and the identification of the subscriber. The latter is implied by the location where the interaction occurs. The following relations between interactions are informally identified:

- Local constraint: the occurrence of `granted` follows the occurrence of `request` (for a given resource identification);
- Local constraint: the occurrence of `free` follows the occurrence of `granted` (for a given resource identification);
- Remote constraint: a resource is only granted to one subscriber at a time.

The floor-control service is illustrated in Figure 7.

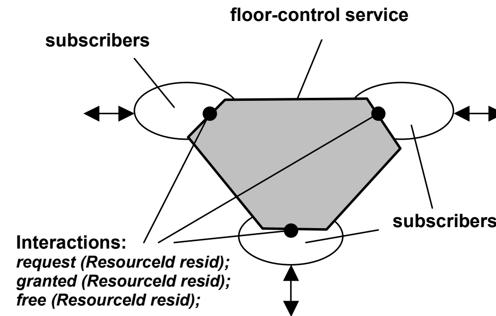

Figure 7 The floor-control service

The service is specified in such a way that interaction requirements between application parts are satisfied without unnecessarily constraining implementation freedom. This freedom includes the structure of the application interaction system (the system that eventually supports the floor-control service) and other technology aspects such as operating systems and programming languages.

6.2 Middleware-centred design

In order to demonstrate the benefits of the identification of the service of the application interaction system, it is useful to apply a typical middleware-centred design method to the same floor-control problem.

In a typical middleware-centred design method, we would have started by enumerating potential alternative solutions based on the identification of application parts and interfaces between these parts. The focus is on the design of application parts structured with constructs provided by the middleware platform.

This would lead to a number of alternative solutions for the floor-control problem, of which we consider a few. These solutions can be basically characterized as either asymmetric or symmetric. In asymmetric solutions, an application part plays the role of a controller, centralizing the coordination of access to shared resources. Some other application parts play the role of subscribers. In symmetric solutions, there is no controller, and all application parts have identical roles in the coordination.

In this example, we assume a component middleware that supports remote invocation. We identify the following asymmetric solutions:

(i)Callback-based. The controller is a singleton component that has an interface with a `request_permission` operation. The parameters of this operation are the identification of the requesting subscriber and the identification of the resource. Subscribers invoke this operation to register their intention to have access to a particular resource. Eventually, when the resource is to be granted to the subscriber, a `grant` operation on the subscriber's interface is invoked by the controller. When the subscriber wants to release the resource, a `free` operation of the controller's interface is invoked. This solution is

illustrated in Figure 8, where the arrows depict invocation dependencies.

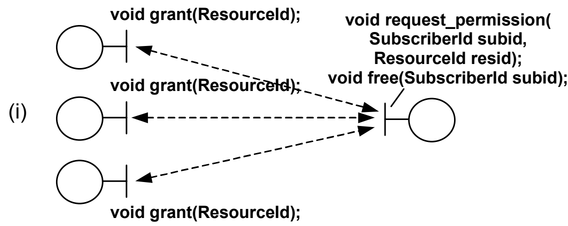

Figure 8 Callback-based floor-control

***(ii)Polling-based*.** The subscribers poll the controller for a certain resource by invoking the operation `is_available`, which returns the Boolean value `true` when the resource is available, and `false` otherwise. When the subscriber wants to release the resource, the operation `free` of the controller's interface is invoked. This solution is illustrated in Figure 9.

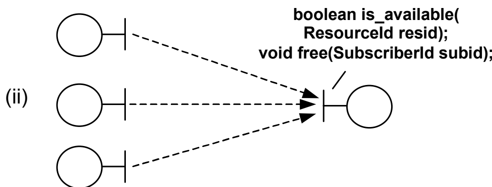

Figure 9 Polling-based floor-control

We identify the following symmetric solution:

***(iii)Token-based*.** A list with the set of available resources circulates among the subscribers. Each subscriber examines the list with the set of identifiers of available resources, removes the identifier of the resource desired and forwards the list invoking an operation on the interface of the following subscriber. When a subscriber wants to release a resource, it inserts the resource identifier to be released in the list. For the sake of simplicity, we assume the set of subscribers is known a priori, so that we can ignore ring management functionality. This solution is illustrated in Figure 10.

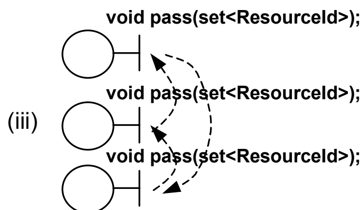

Figure 10 Token-based floor-control

6.3 Protocol-centred design

A protocol-centred design would be structured in terms of protocol entities and a lower level service. For the sake of this example, let us suppose we select a lower level service that offers reliable transfer of a sequence of octets. The protocol entities are responsible for encoding PDUs and delivering these to the lower level service.

Several alternative protocols are possible, such as:

- An asymmetric protocol similar to the *callback-based* solution, as illustrated in Figure 11 (i).
- An asymmetric protocol similar to the *polling-based* solution, as illustrated in Figure 11 (ii).
- A symmetric protocol similar to the *token-based* solution, as illustrated in Figure 11 (iii).

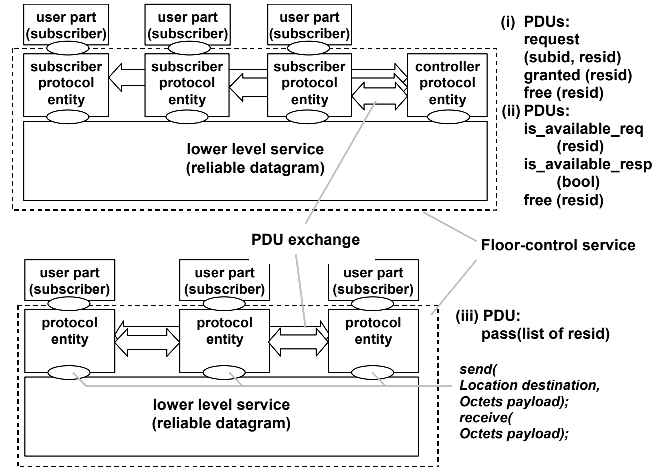

Figure 11 Alternative solutions in the protocol-centred paradigm

6.3 Discussion

The solutions we have presented for both the middleware- and protocol-centred paradigms could be used as particular implementations of the floor-control service, as shown in Figure 12. These solutions introduce abstractions that are bound to particular design solutions, such as the *controller*, an abstraction that is not identified in the symmetric design. In contrast, the floor-control service is a stable abstraction, and shields subscribers from the particular way in which the service is implemented; both with respect to *premature commitments to particular design solutions* (callback-, polling-, or token-based) and with respect to *premature commitments to a particular interaction pattern provided by the infrastructure* (either a middleware platform or a lower level service provider).

In analogy with the development of protocols, applying directly the middleware paradigm for applications with complex interaction requirements, yields similar results to designing a protocol *without* considering the required service explicitly. As has been pointed in [11], the definition of services should precede or accompany, but definitely not follow, the specification of protocols. The use of the service concept leads to careful consideration of the interaction problem being addressed. In terms of system structure, the use of the service concept promotes an appropriate application of the layering principle.

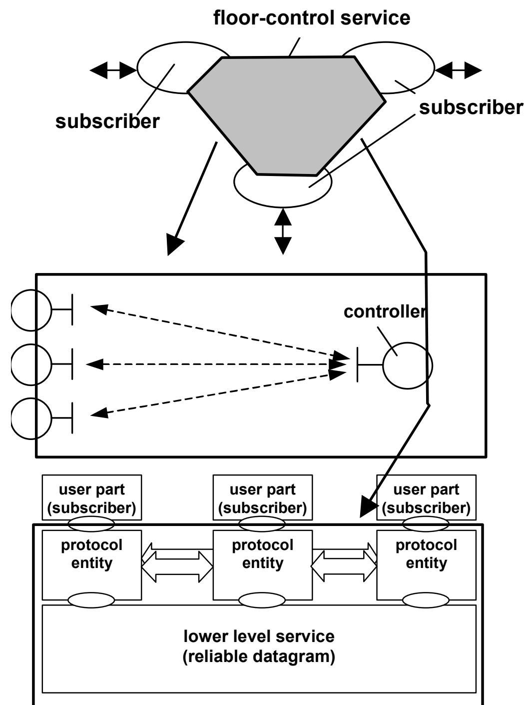

Figure 12 Floor-control service as stable abstraction

7 Conclusions

We have argued the case for an increased role of interaction system design in the development of distributed applications. The focus on bottom-up development, using as a starting point constructs provided by a middleware platform has neglected the role of interaction system design in the development of distributed applications. As a consequence of this practice, designers tend to focus on abstractions of particular design solutions, without recognizing interaction aspects that remain stable.

With the appropriate use of service specifications as a starting point for application interaction system design, it becomes irrelevant for the design of application parts whether the protocol-centred or middleware-centred paradigm is followed in the design of the supporting interaction system.

We presented our approach as a pure top-down design trajectory for interaction systems, starting from service definition to service design. However, this does not exclude the use of bottom-up knowledge. Bottom-up experience is what allows designers to re-use middleware infrastructures and lower level services, and to find appropriate service designs that implement the required service. Stable abstractions for service design should be derived from knowledge obtained from the solution space (as in a synthesis-based design method [9]).

Cariou et al. [2] have recently explored the notion of medium, which corresponds to our notion of *application interaction system*, focussing on the use of UML to represent such mediums. Our future research focuses on extending and/or complementing UML with respect to the representation of the service concept, in particular when specifying complex application interaction systems.

Acknowledgements

This work is partly supported by the Telematica Instituut in the context of the ArCo project (http://arco.ctit.utwente.nl/) and by the European Commission in context of the MODA-TEL IST project (http://www.modatel.org).

References

[1] R. J. Allen, and David Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, vol. 6, n. 3, July 1997, pp. 213-219.

[2] E. Cariou, A. Beugnard, J. M. Jézéquel: An Architecture and a Process for Implementing Distributed Collaborations. Proceedings *Sixth International Conference on Enterprise Distributed Object Computing*, September 2002, Lausanne, Switzerland, Sept. 2002, 132-143.

[3] Microsoft Corporation. *Microsoft .NET Remoting: A Technical Overview*, July 2001, available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/hawkremoting.asp

[4] Object Management Group. *Common Object Request Broker Architecture: Core Specification*, Version 3.0, Dec. 2002.

[5] Object Management Group. *CORBA Component Model*, v3.0. formal/02-06-65, July 2002.

[6] R. Sharp. *Principles of protocol design*. Prentice-Hall International Series in Computer Science, Prentice-Hall, Great Britain, 1994.

[7] M. van Sinderen. *On the Design of Application Protocols*. Ph.D. Thesis. University of Twente, The Netherlands, March, 1995, available at http://www.cs.utwente.nl/~sinderen/publications/thesis.html

[8] M. van Sinderen and L. Ferreira Pires. Protocols versus objects: can models for telecommunications and distributed processing coexist? In Proceedings *Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, October 1997, 8-13.

[9] B. Tekinerdogan. *Synthesis-Based Software Architecture Design*. Ph.D. Thesis. University of Twente. March, 2000.

[10] C. A. Vissers, L. Ferreira Pires, D. A. Quartel, M. van Sinderen. *The Architectural Design of Distributed Systems*, Lecture Notes, University of Twente, Enschede, The Netherlands, Nov. 2002.

[11] C.A. Vissers, L. Logrippo. The importance of the service concept in the design of data communications protocols. In Proceedings *Fifth IFIP WG6.1 International Conference on Protocol Specification, Testing and Verification*, June 1985, 3-17.