IEEE distributed systems ONLINE
Expert-authored articles and resources

**IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922 © 2005 Published by the IEEE Computer Society**
**Vol. 6, No. 3; March 2005**

# Designing Interaction Systems for Distributed Applications

**João Paulo Almeida**, *Centre for Telematics and Information Technology, University of Twente*
**Marten van Sinderen**, *Centre for Telematics and Information Technology, University of Twente*
**Dick A.C. Quartel**, *Centre for Telematics and Information Technology, University of Twente*
**Luís Ferreira Pires**, *Centre for Telematics and Information Technology, University of Twente*

**Application designers should explicitly design interaction systems that support application-level interactions. Designers can do this using middleware-centered and protocol-centered development approaches.**

In recent years, software infrastructures such as middleware platforms have dominated distributed applications development. Typical design methods based on the reuse of these infrastructures partition an application into parts and interconnect them through constructs provided directly by the infrastructure.

However, because application interaction requirements vary, a gap often exists between the interactions that software infrastructures provide and those required for interconnecting application parts. This argues for explicitly designing interactions between application parts.

Seminal work in systems and protocol design have acknowledged the importance of explicitly designing such interaction mechanisms in distributed systems.[1] More recent software architecture efforts[2] have identified the *connector* construct, which emphasizes the

importance of describing and analyzing the interaction aspects of software components.

In this article, we discuss the explicit design of interaction mechanisms for developing distributed applications. When interactions between system parts require an explicit design, the concept of an *interaction system* comes into play. We define criteria that designers can use to decide whether an interaction system requires an explicit design. We also show that designers can apply both middleware-centered and protocol-centered development approaches in designing an application-level interaction system.

## Interaction systems

An interaction system supports the set of related interactions between two or more system parts (see Figure 1). In general, interaction systems can support arbitrarily complex interaction needs. An interaction system's complexity can vary depending on the interactions it supports. For example, connectors are interaction systems that satisfy basic communication needs between software components.

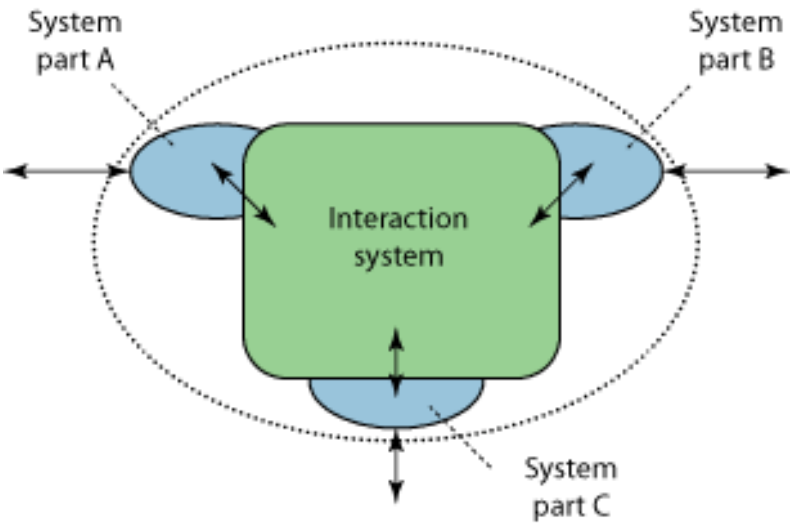


Figure 1. An interaction system.

We call a system's external perspective a *service.*[1] A service defines a system's observable behavior in terms of its interactions at the interfaces between the system and the environment, and these interactions' relationships. A service doesn't disclose internal organization details that designers can define in the system's implementations.

Because a system part is a system in itself, designers can apply the service concept recursively in system design. This recursive application lets a designer consider a system's behavior at different related decomposition levels. In general, the number of decomposition levels and the particular decomposition choices depend on system requirements and the designer's objectives. Because an interaction system is a system, designers can also describe it as a service.

## Protocol-centered paradigm

In the protocol-centered paradigm, user parts interact locally with a *service provider*. A service provider comprises a *lower-level service provider* and *protocol entities*, which interact to provide the required service to user parts. A distributed application's model comprises service users, a layer of protocol entities, and a lower-level service provider.[3] Protocols such as those standardized by the International Organization for Standardization and the Internet Engineering Taskforce use models that resemble this one.

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

Protocol entities communicate with each other by exchanging messages, often called *protocol data units*, through a lower-level service. PDUs define the syntax and semantics for unambiguous understanding of the information exchanged between protocol entities. A protocol entity's behavior 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 to provide the requested service.

Designers can define protocols 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.

## Middleware-centered paradigm

In the middleware-centered paradigm, system parts interact through a limited set of interaction patterns offered by a *middleware platform*. A distributed application model

comprises the middleware platform and a collection of interacting parts, often called *objects* or *components*.

Several different types of middleware platforms exist, each offering different types of interaction patterns between objects or components. We can further characterize the middleware-centered paradigm according to the types of interaction patterns that the platform supports, such as *request/response*, *message passing,* and *message queues*. Examples of available middleware platforms are CORBA , the CORBA Component Model, .NET, and Web Services.

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

Interestingly, the middleware-centered paradigm depends on the protocol-centered paradigm in that protocols eventually realize interactions between application parts.[4] For example, CORBA object request brokers interact through the General Inter-ORB Protocol.

Design methods based on middleware platform reuse often involve partitioning the application into parts and defining interconnection aspects by defining interfaces between these parts (for example, by using object-oriented techniques and abstracting from distribution aspects). The interaction patterns supported by the targeted platform constrain the available constructs for building interfaces. Example constructs include *operation invocation*, *event sources* and *sinks*, and *message queues*. This structuring strategy encourages a decomposition level that emphasizes interaction systems that the software infrastructure (middleware platforms) provides directly.

This structuring strategy implies that interaction patterns provided by a particular middleware platform directly influence the application structure. The application design is therefore platform-specific, in that the design depends on particular technological conventions adopted by the middleware platform and that the application's structure depends on the provided set of interaction patterns.

Using predefined software connector[5] implementations for developing distributed applications leads to results similar to those obtained by using middleware platforms. We can say the same about infrastructures that implement coordination models, such as Linda and its variants.[6] In both cases, the application's structure depends on the set of interaction patterns provided.

## Interaction system 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, we can identify *application interaction systems* that support application-level interactions between application parts (see Figure 2a). Figure 2b shows interaction systems that the middleware or protocols provide.

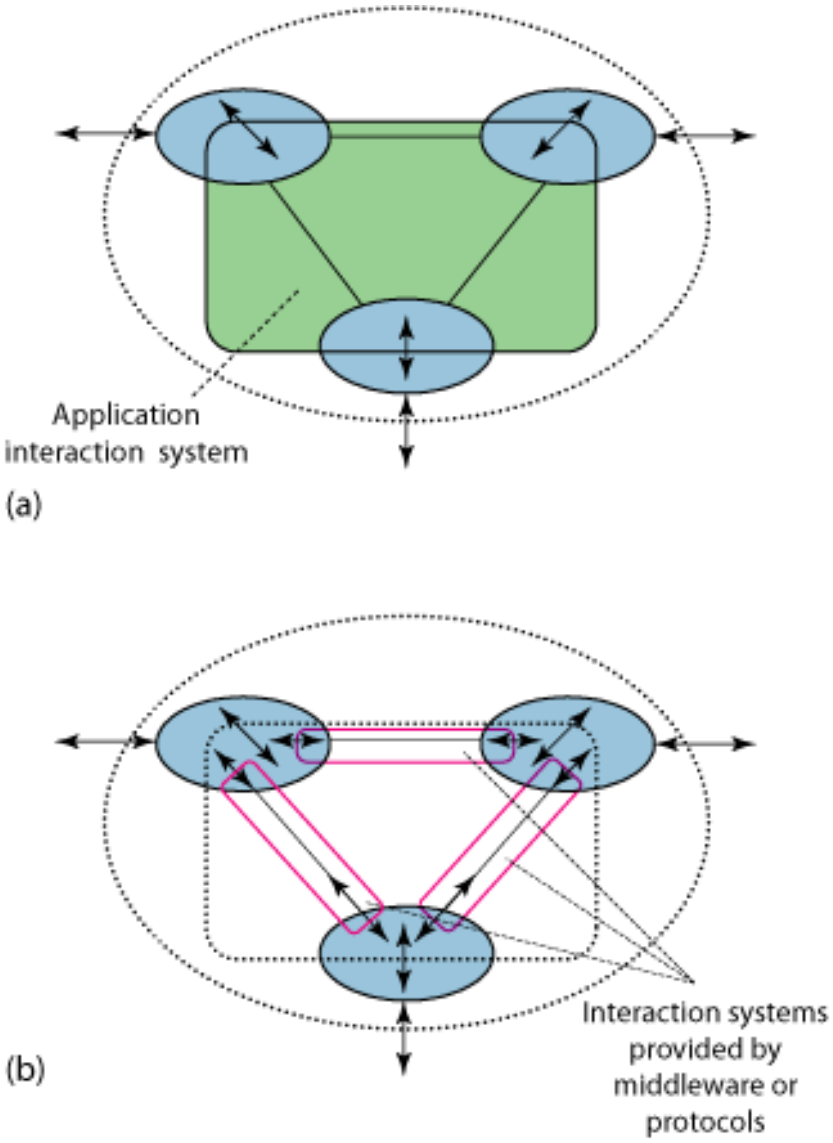


Figure 2. (a) Application interaction systems and (b) interaction systems provided by middleware or protocols.

Whether a designer should consider an explicit design of an application interaction system depends on the application's requirements and the designer's objectives. We define the following determining criteria:

- *The relation between system parts is complex.* In this case, designers should pay attention to the design of the relation between system parts. Designers can make this relation a separate design object—that is, considering the system parts' interaction system separately. Designers can consider the interaction system at different abstraction levels to cope with the relation's complexity. The middleware-provided interaction system plays an important role at lower abstraction levels.

- *Interactions are changed rather than just the contributions to interactions by individual system parts*. This occurs if a designer envisions several different middleware platforms as alternatives to support the interactions. A designer can only replace an interaction mechanism by another equivalent interaction mechanism if the design clearly indicates the mechanism's relevant characteristics. Interaction system design naturally supports this.

- *The interaction system is general-purpose, offering opportunity for reuse*.

- *Different design authorities are responsible for the process of designing the interaction system and system parts*. Specifying the interaction system's service serves as a contract for the communication between system part and interaction system designers.

- *Explicit attention to design choices that concern the effectiveness and efficiency of interactions is required.* In this case, designers can address quality of service aspects influenced by distribution aspects separately.

A starting point in designing an application interaction system is the application service specification, capturing a description of the required application interaction system from an external perspective. The design of the application interaction system could, in principle, have any internal structure as long as it provides the required service. For example, it could directly use a data transport service as in a protocol approach. Nevertheless, we observe that the middleware leverages the reuse of a large building block, providing an interoperability architecture across programming languages, operating systems, and network technologies and

offering facilities for defining application-level information attributes. So, designers should also consider the interaction systems provided by the middleware as alternatives for building application interaction systems.

A systematic interaction system design method based on the protocol-centered paradigm comprises two parts. First, a designer defines the service to be supported in terms of the service primitives that occur at service access points and the relationships between service primitives. Second, he or she decomposes this service in terms of a structure of protocol entities and a lower-level service. This resulting structure, which we call a *protocol*, must be a correct implementation of the service. Designers can access this formally if they specify both the service and protocol in some formal language.[3,7]

A systematic interaction system design method based on the middleware-centered paradigm also starts with the definition of the service to be supported (as in the case for the protocol-centered paradigm). After that, the designer decomposes this service in terms of a structure of service components and the interaction systems provided by a middleware platform. This resulting structure must correctly implement the service. Again, designers can assess this formally, if they specify both the service and its design (service components and platform) using some formal language.

## Example: Auction service

We use an auction application to illustrate the use of an application interaction system and its service specification in a design trajectory. In this example, a set of application parts participates in auctions in which products are sold to the highest bidder. Any application parts can auction off and bid for products. To simplify the example, we use a fixed and predefined number of participants.

## Service definition

We start with the definition of the *auction service*. The service relates the following interactions:

- `offer` and `offer_ind`, both with attributes `product_id` and `opening_value`. The `product_id` uniquely identifies a product being auctioned.

- `bid` and `bid_ind`, both with attributes `product_id` and `bid_value`.

- `outcome_ind`, with attributes `product_id`, `final_value`, and `participant_id`. The `participant_id` uniquely identifies an auction participant. In this interaction, it identifies the winning bidder.

These interactions occur at the interfaces between the auction interaction system and the application parts. An interaction's occurrence results the establishment of values for its attributes. In addition to the attributes just listed, for each interaction, the `participant_id` is implied by the location where the interaction occurs.

A useful technique for specifying a service is defining it as a conjunction of different interaction constraints.[7] Particularly , a useful structuring principle is to identify local and end-to-end (or remote) constraints. In this example, a local constraint exists for each participant: `bid` can only occur after `offer_ind` and before `outcome_ind` (for a given `product_id`). The remote constraints between participants are that

- an occurrence of `offer_ind` for each auction participant follows the occurrence of `offer;`

- an occurrence of `bid_ind` for each auction participant follows the occurrence of `bid`; and

- `outcome_ind` occurs •*t* seconds after the last `bid_ind` occurs (for a given `product_id`).

Designers should specify the service so 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 auction service) and other technology aspects such as operating systems and programming languages.

## Middleware-centered design

In a typical middleware-centered design method, we would've started by enumerating potential alternative solutions based on the identification of application parts and interfaces between these parts. Such an approach focuses on the design of application parts structured with constructs provided by the middleware platform.

This leads to numerous alternative solutions for the auction application, of which we consider

a few. We can characterize these solutions basically as either *asymmetric* or *symmetric*. In asymmetric solutions, an application part acts as a controller, centralizing the auction's coordination. Some other application parts play the role of auction participants, offering and bidding for products. In symmetric solutions, no controller exists, and all application parts have identical coordination roles.

In this example, we assume a component middleware that supports remote invocation. We consider two asymmetric solutions. The first is *callback-based.* The controller is a singleton component that has an interface with operations `register_offer` and `register_bid`. These operations' parameters are product identification, the product's opening value or a bid, and the reference to the participant's interface (seller in the case of `register_offer` or bidder in the case of `register_bid`). Participants offer the following operations: `offer_callback`, bid_callback, and `outcome_callback`. The controller invokes `offer_callback` and `bid_callback` on each participant's interface when offers and bids are registered. Eventually, when no bids are registered for a period of time, the controller invokes the `outcome_callback` operation on each participant's interface. Figure 3a shows this solution, where the arrows depict invocation dependencies.

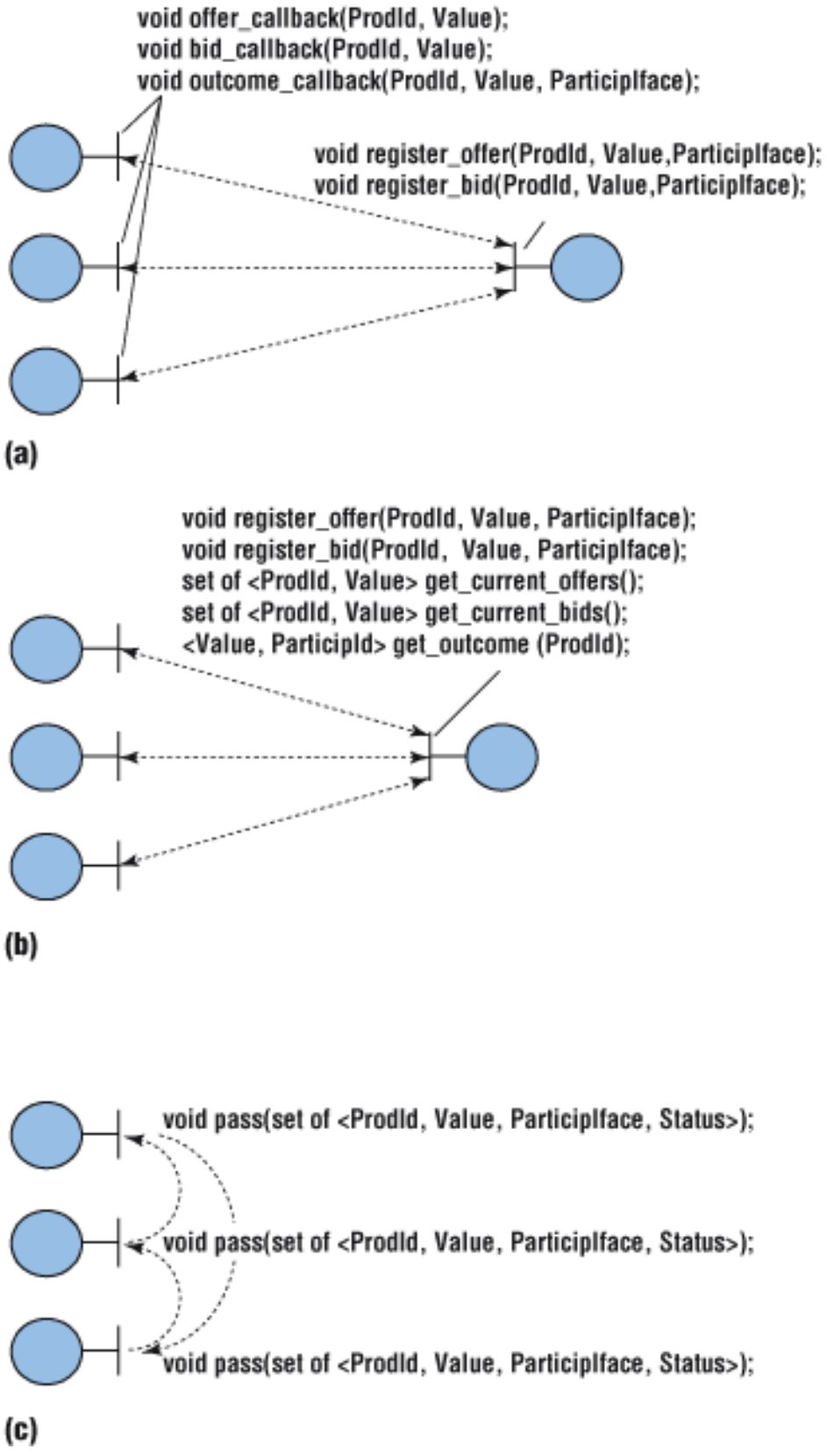


Figure 3. Alternative solutions in the middleware-centered paradigm (arrows depict invocation dependencies): (a) callback-based, (b) polling-based, and (c) token-based.

The second asymmetric solution is *polling-based* (see Figure 3b). The controller is also a singleton component offering operations `register_offer` and `register_bid` as well as `get_current_offers`, `get_current_bids`, and `get_outcome`. The participants poll the controller for offers and bids by invoking the operations `get_current_offers` and `get_current_bids`, which returns sets of current offers and bids. The participants also poll the controller for the outcome of a particular product's auction, with the operation `get_outcome` .

We also consider a symmetric *token-based* solution (see Figure 3c). In this case, a list with the current product offers, their status, and highest bids circulates among the participants. Each participant examines the list, places offers or bids and forwards the list invoking an operation on the following participant's interface. The participant that introduces an offer is responsible for changing the offer's status in the list to `closed` when no bids are made for a period of time. This participant must also remove the closed offer after it circulates the ring once. For simplicity's sake, we assume the set of participants is known a priori, so we can ignore ring management functionality. We additionally assumed that participants don't fake offers and bids and that the time it takes for the list to rotate the ring should be significantly shorter than the bidding time •*t*.

## Protocol-centered design

We would structure a protocol-centered design in terms of protocol entities and a lower-level service. For this example, let's 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 possible alternative protocols include

- an asymmetric protocol similar to the callback-based solution (see Figure 4a),
- an asymmetric protocol similar to the polling-based solution (see Figure 4b), and
- a symmetric protocol similar to the token-based solution (see Figure 4c).

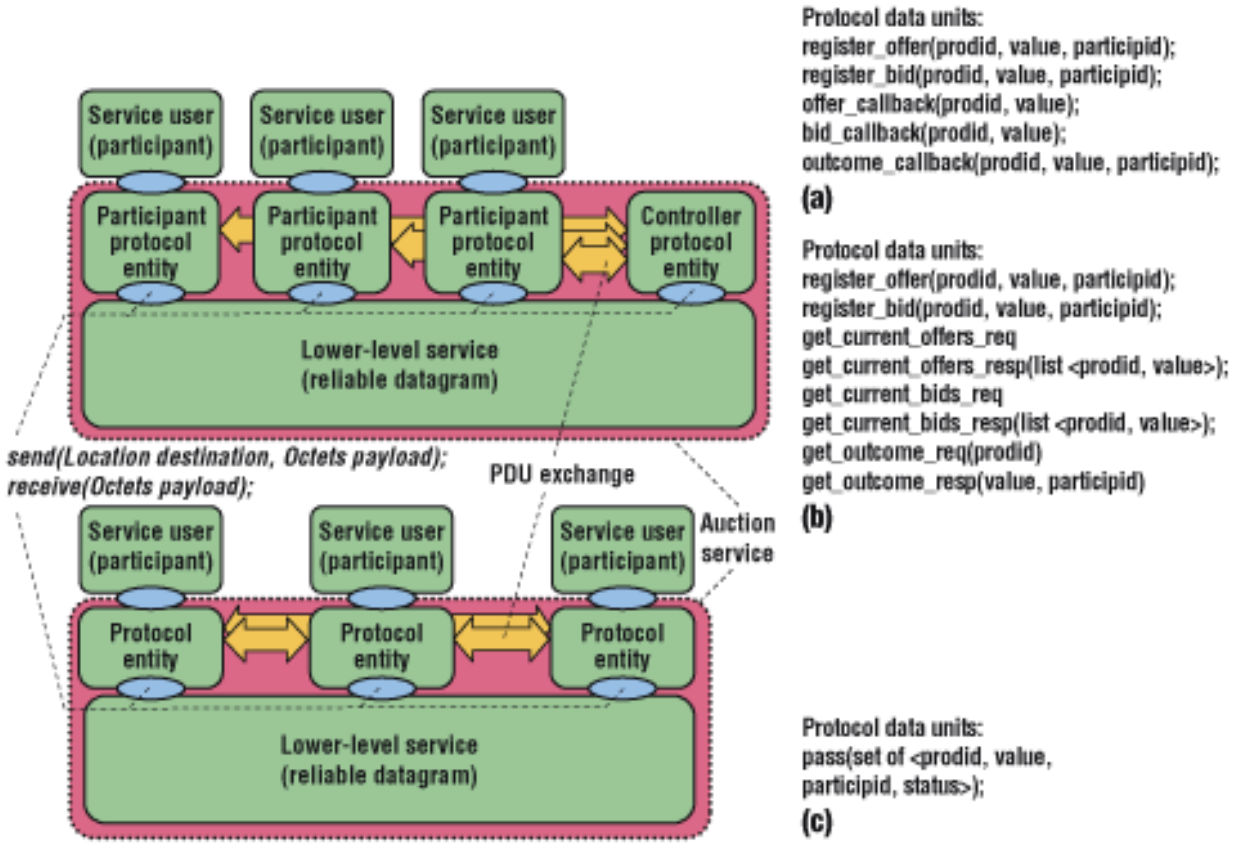


Figure 4. Alternative solutions in the protocol-centered paradigm: an asymmetric protocol (a) similar to the call-back based solution or (b) similar to the polling-based solution and (c) a symmetric protocol similar to the token-based solution.

## Discussion

The solutions we've presented for the middleware- and protocol-centered paradigms could be used as particular implementations of the auction service (see Figure 5). These alternatives introduce abstractions that are bound to particular design solutions, such as the *controller*, an abstraction that the symmetric design doesn't identify. In contrast, the auction service is a stable abstraction and shields subscribers from a service's particular implementation, both with respect to commitments to particular design solutions (callback-, polling-, or token-based) and with respect to commitments to a particular interaction pattern provided by the infrastructure (a middleware platform or a lower-level service provider). This agrees with other work that claims that middleware shouldn't determine nor be mistaken for an application's architecture. [8]

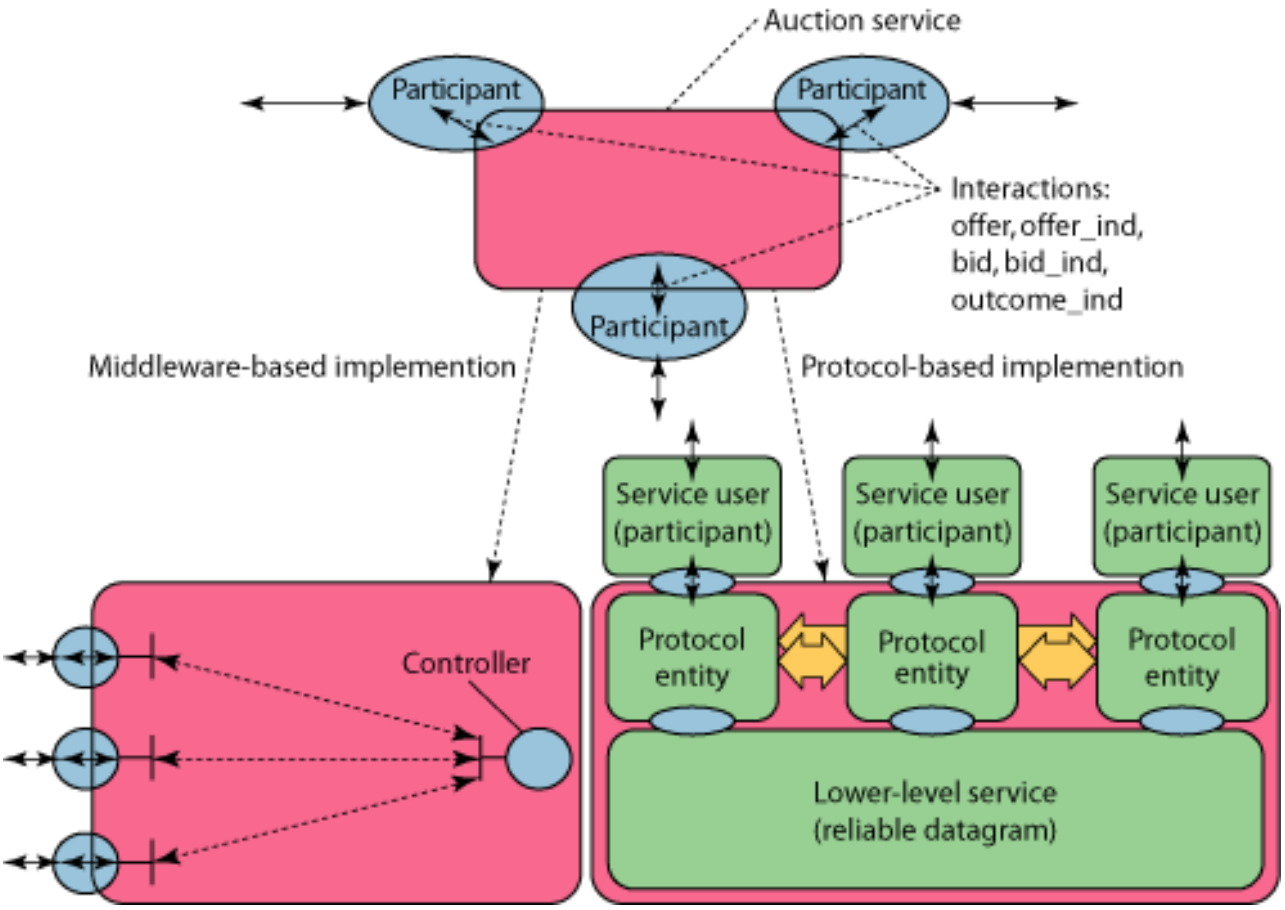


Figure 5. The auction service as a stable abstraction.

Using middleware for application development without considering the required application service explicitly is like designing a protocol without considering the required service explicitly. As pointed out elsewhere,[1] service definition should precede protocol specification. Using the service concept leads to careful consideration of the interaction problem at hand. For systems verification, using service specifications lets designers compare service specifications to implementations of specified services.[3,7] For system structure, using the service concept promotes an appropriate application of the layering principle.

## Conclusion

Starting with a service specification lets designers choose between a protocol- or middleware-centered paradigm when designing application interaction systems. The choice of development paradigm doesn't affect the design of the application parts that use the supporting interaction system. We've shown elsewhere[9] that you can use interaction system design and the service concept to enable middleware-platform-independent design and platform-specific realization in Model-Driven Architecture development.[10]

We've presented here a top-down design trajectory for interaction systems, starting from service definition to service design. However, this doesn't exclude using bottom-up knowledge. Bottom-up experience lets designers reuse middleware infrastructures and lower-level services and find appropriate service designs that implement the required service. Designers should derive stable abstractions for service design from knowledge obtained from the solution space.

Our notion of interaction systems corresponds to the concept of connectors in software architecture. However, most software architecture work has focused either on providing implementations of basic connectors as software infrastructures[5] or on the description of connectors[2,11] as opposed to identifying connectors' roles in the development process and addressing the connectors' design. Designers can also use the criteria we defined for justifying the design of interaction systems to justify the explicit design of connectors.

Eric Cariou and his colleagues[12] have explored the notion of *medium*, which corresponds to our *application interaction system* concept, focusing on the use of the Unified Modeling Language[13] to represent such mediums. In our future research we intend to extend or complement UML with respect to the service concept representation, particularly when specifying complex application interaction systems from different related viewpoints and levels of abstraction.

## Acknowledgments

This work is part of the Freeband A-MUSE project (www.a-muse.freeband.nl), which is sponsored by the Dutch government under contract BSIK 03025. The European Commission within the MODA-TEL IST project (www.modatel.org) has also partly supported this work. We acknowledge Chris Vissers, who provided the foundation upon which we built this work.

# References


1. C.A. Vissers and L. Logrippo , "The Importance of the Service Concept in the Design of Data Communications Protocols,"*Proc. 5th IFIP WG6.1 Int'l Conf. Protocol Specification, Testing and Verification,*North-Holland,1985,pp. 3–17.

2. R.J. Allen and D. Garlan, , "A Formal Basis for Architectural Connection,"*ACM Trans. Software Eng. and Methodology,* vol. 6, no. 3, 1997, pp. 213–219.

3. G.J. Holzmann, , *Design and Validation of Computer Protocols,* Prentice Hall, 1991.

4. M. van Sinderen and L. Ferreira Pires, , "Protocols Versus Objects: Can Models for Telecommunications and Distributed Processing Coexist?"*Proc. 6th IEEE Computer Soc. Workshop Future Trends of Distributed Computing Systems,* IEEE CS Press, 1997,pp. 8–13.

5. N. Medvidovic , P. Oreizy, and R.N. Taylor, , "Reuse of Off-the-Shelf Components in C2-Style Architectures,"*Proc. 1997 Symp. Software Reusability* (SSR 97), ACM Press, 1997.

6. A. Papadopoulos and F. Arbab, , "Coordination Models and Languages,"*Advances in Computers* (The Engineering of Large Systems, vol. 46), Academic Press, New York,1998.

7. C.A. Vissers , et al., "Specification Styles in Distributed Systems Design and Verification,"*Theoretical Computer Science,* vol. 89, Elsevier, 1991, pp. 179–206.

8. J. Wileden and A. Kaplan, , "Middleware as Underwear: Toward a More Mature Approach to Compositional Software Development ,"*Workshop Compositional Software Architectures,* http://www.objs.com/workshops/ws9801/papers/paper061.html, 1998.

9. J.P.A. Almeida , et al., "A Systematic Approach to Platform-Independent Design Based on the Service Concept,"*Proc. 7th IEEEInt'l Conf. Enterprise Distributed Object Computing* (EDOC 2003), IEEE CS Press, 2003, pp. 112–134.

10. *MDA Guide Version 1.0.1,* omg/2003-06-01, Object Management Group, June 2003.

11. N. Medvidovic and R.N. Taylor, , "A Classification and Comparison Framework for Software Architecture Description Languages," http://csdl.computer.org/comp/trans/ts/2000/01/e0070abs.htm, *IEEE Trans. Software Eng.,* vol. 26, no. 1, 2000,pp. 70–93.

12. E. Cariou , A. Beugnard, and J. M. Jézéquel, , "An Architecture and a Process for Implementing Distributed Collaborations,"*Proc. 6th Int'l Conf. Enterprise Distributed Object Computing* (EDOC 2002), IEEE CS Press, 2002, pp. 132–143.

13. *Unified Modeling Language (UML),* v. 1.5, formal/2003-03-01, Object Management Group, Mar. 2003.

**João Paulo A. Almeida** is a PhD candidate in computer science at the University of Twente. His research interests include design methods, architectures and concepts for distributed systems, and model-driven and service-oriented design. He has an MSc in telematics from the University of Twente. He is a member of the ACM and the IEEE. Contact him at the Faculty of Electrical Eng., Mathematics and Computer Science, Univ. of Twente, PO Box 217, 7500 AE Enschede, Netherlands; j.p.andradealmeida@utwente.nl.

**Marten J. van Sinderen** is an associate professor of the Faculty of Electrical Engineering, Mathematics, and Computer Science and manager of the research program on telematics systems and services at the Centre of Telematics and Information Technology (CTIT), both at the University of Twente. His research interests include design methods and architectures for telematics systems. He received his PhD in computer science from the University of Twente. He is a member of the IEEE. Contact him at the Faculty of Electrical Eng., Mathematics and Computer Science, Univ. of Twente, PO Box 217, 7500 AE Enschede, Netherlands; m.j.vansinderen@utwente.nl.

**Dick A.C. Quartel** is an assistant professor in the Faculty of Electrical Engineering, Mathematics and Computer Science at the University of Twente. His research interests include distributed system architecture, architectural modeling, service-oriented design, service-oriented computing technologies, and context-aware services. He received his PhD in computer science from the University of Twente. Contact him at the Faculty of Electrical Engineering, Mathematics, and Computer Science, Univ. of Twente, PO Box 217, 7500 AE Enschede, Netherlands; d.a.c.quartel@utwente.nl.

**Luís Ferreira Pires** is an associate professor in the Faculty of Electrical Engineering, Mathematics and Computer Science at the University of Twente. His research interests include design methods and architectures for telematics systems, especially for context-aware and mobile applications. He received his PhD degree in electrical engineering from the University of Twente. Contact him at the Faculty of Electrical Eng., Mathematics and Computer Science, Univ. of Twente, PO Box 217, 7500 AE Enschede, Netherlands; pires@cs.utwente.nl.

**Cite this article:** João Paulo Almeida, Marten van Sinderen, Dick A.C. Quartel, and Luís Ferreira Pires, "Designing Interaction Systems for Distributed Applications," *IEEE Distributed Systems Online,* vol. 6, no. 3, 2005.