View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace at VU

Towards Object-based Wide Area Distributed Systems

Maarten van Steen, Philip Homburg, Leendert van Doorn
Andrew S. Tanenbaum, Wiebren de Jonge
Vrije Universiteit, Amsterdam

Abstract

In order to facilitate the construction of wide area distributed systems, it is necessary that we adopt a model that simplifies application development. In this position paper we advocate an object-based approach. Our approach allows for flexibility because many of the technical details of distribution, such as communication protocols, consistency rules, etc. can be hidden behind the objects' interfaces. In addition, we allow distributed objects to offer alternative implementations for an interface. A client may choose the most suitable implementation. We discuss the use of distributed objects as the means to this end, and compare our approach to existing ones.

1 Introduction

Wide area distributed applications pose varying demands on the underlying operating systems, often making the development of the application itself a difficult task. For example, development of distributed applications often requires the following:

- Support for expressing communication at a sufficiently high level of abstraction.
- Efficient implementations of different communication models and operating system services, dependent on their usage by different kinds of applications.
- Support for execution and communication in a heterogeneous environment.
- Flexibility with respect to reconfiguring a system during its execution.
- Support for security.

These demands sometimes conflict. For example, it may be difficult to attain efficiency for communication at a sufficiently high level of abstraction without taking (application-dependent) communication structures into account. Likewise, having support for different communication models within a single framework may provide flexibility, but it may simply be too heavy weight for a specific domain.

To alleviate such problems, we feel that a framework should provide support for:

- An integrated approach for developing application-specific components, and developing more general components that handle, for example, state distribution and communication.
- Flexibility with respect to how components are implemented, particularly where implementations depend on the external factors such as operating systems or underlying communication network.

We advocate an object-based approach for two reasons. First, the concept of an object provides a natural way of separating functionality from implementation. This separation is needed because wide area applications require that components can be distributed across different underlying systems. Having different systems, in turn, implies that we should be able to provide different implementations for components that are functionally the same.

The second reason is that objects naturally encapsulate state with permissible operations. The important issue is that this provides modularity with regards to *instances*, and not just with respect to their *description*. In other words, encapsulation leads to a natural way of protecting and hiding information in an individual component. The only way that this information can be accessed is through that component's interface to the outside world. In view of the necessity for different implementations of functionally equivalent components, shielding implementations behind interfaces on a per-component basis, is essential.

The combination of separating functionality from implementation, and protecting and hiding information into a single component, allows for the integration and flexibility mentioned above. In particular, we have adopted a model in which a distributed object may offer several alternative implementations for an interface. A client selects the most suitable implementation based, among other things, on its present environment. In wide area distributed computing, it is this type of flexibility we think is important. We return to this issue below.

What functionality should the environment offer? It is here that we feel that many object-based models restrict developers in several ways. First, many offer functionality that is not well-suited for the development of wide area distributed applications. Second, the functionality is offered at a relatively high level of abstraction, which may make it difficult to implement efficiently. Third, many do not support developers to adapt these implementations to their own needs. In our approach, we do not restrict developers in these ways, but instead provide the flexibility that is needed. We return to this in Section 4.

2 The global model

In the model we propose, an object is an entity with:

- A collection of values, referred to as the **state** of the object.
- A collection of **methods** by which the state of the object can be inspected and modified.
- A collection of **interfaces**, each interface pointing to a subset of the methods. An invocation of a method can occur *only* via an interface.

Each object has a globally unique and location-independent object identifier. Different, and mutually independent naming services can be used to attach user-oriented names to objects. Objects are shared by either using a global naming service, or by exporting their identifier to other name services.

We do not support inheritance as we feel that alternatives exist that are more flexible for structuring applications. In our model, we instead allow an object to have multiple interfaces. In this way, a developer can extend or adapt an object by adding new method implementations and an interface comprising those methods, without affecting clients that already use the object. We take the approach that constructing an object should focus on designing its interfaces. Interfaces determine the functionality of an object, which in turn can be implemented in different ways as explained above. In our model, interface design and implementation is independent of any programming language. Objects, and users of objects can be written in any language.

Structuring is further supported by making a distinction between non-decomposable objects, called **primitive objects**, and **composite objects**. A composite object aggregates one or more (composite or primitive) objects into a single object. To this aim, an interface of a composite object may point to methods from different objects in the composition. From the outside, however, a composite object is indistinguishable from a primitive object.

Objects are passive: their methods are invoked by threads which are the active entities in our model. Threads and objects are orthogonal. This means that threads exist independently of objects, and that objects have no predefined properties regarding concurrency. Instead, any of such properties should form part of an interface specification.

In the following section we pay attention to how objects are organized in our model.

3 Object organization

An object may have its interfaces and state distributed across multiple address spaces. Such a distribution is realized through local objects. A **local object** has its state, methods, and interfaces in a single address space. A **distributed object** is a collection of local objects that may reside in different address spaces. The local objects of a distributed object communicate with each other in order to maintain and present a consistent view on the overall state. By properly encapsulating this communication, a distributed object appears on the outside as an ordinary object with all communication hidden from its clients.

Basic organization. Communication is implemented through communication objects. In its simplest form, a **communication object** is built as a non decomposable object encapsulating an exist-

ing message-passing facility such as local RPC, an Ethernet device driver, or TCP/IP. More advanced facilities, such as general multicasting and shared memory mechanisms, are encapsulated in objects that make use of these simpler objects.

State partitioning and replication is handled through separate objects that make use of (simple) communication objects. For example, replication can be supported by adding an object to each local object that multicasts state changes to the local objects at other locations, possibly using hardware multicast, if available. Similarly, a separate object may be responsible to forward operations on the distributed object's state when that state is maintained by a remote local object. A state consistency protocol may be implemented in a separate object that interacts with objects handling state partitioning and replication.

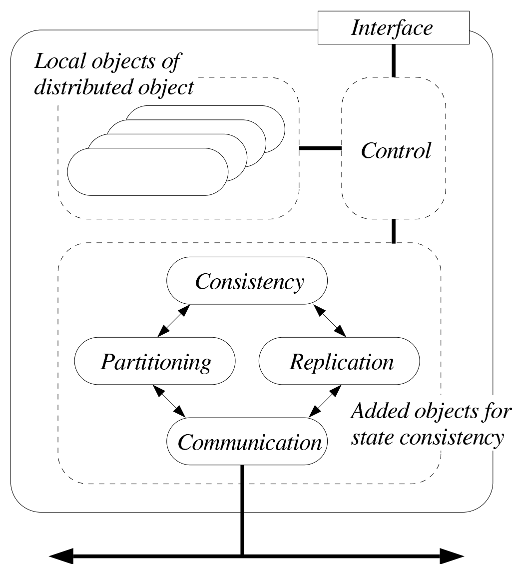

Figure 1: The general organization of a distributed object in a client's address space.

This approach leads to the general organization shown in Figure 1. Three different kinds of components are distinguished. The local component consists of objects that implement the basic functionality made available through methods at the interface. The communication component implements a protocol stack needed for information exchange with other parts of the distributed object. The control component, finally, takes care of argument marshaling and controls invocation of method implementations.

Our approach differs from most others in that a distributed object may offer different implementations for its interfaces. For example, an interface may be implemented by having a replica of the object's state placed in the client's address space. This would then be combined with method implementations and an implementation of a specific protocol stack to keep the overall state consistent. Alternatively, a client may possibly also have the choice for selecting an RPC-based implementation of that interface. In that case, the local component of Figure 1 would be obsolete, and only the implementation of a basic communication object would be necessary. It is up to the client to select the "best" possible implementation, as is discussed below.

We do not assume that objects handling communication, partitioning, replication, and consistency, are necessarily provided beforehand. In other words, our model allows an application developer to construct such objects if so required. This also means that if it is necessary or convenient to place certain objects into kernel space, or likewise, if objects such as device drivers are to be placed in user space, we do not prevent a user from doing so. As explained in [6] this does require special certification support.

Binding. In order for a client to invoke a method of a distributed object, it is necessary that local objects are installed and initialized in the client's address space. These local objects implement the interface(s) of the distributed object as requested by the client. This binding process, which is described in more detail in [2], consists of four steps.

- *Name resolution.* The client provides a name that identifies the object. This name is then resolved to the object's unique identifier.
- *Peer group resolution.* The distributed object may offer different ways to implement the interface requested by the client. Each implementation requires communication with a peer group: a collection of local objects that are part of the distributed object and which reside in other address spaces. In this step, exactly which peer groups supporting the interface's implementation, together with its associated communication protocol, is resolved.
- *Peer group selection.* From the set of peer groups, the one that is most suitable for im-

plementing the interface at the client's side, is selected.

- *Implementation selection.* Finally, the objects that take care of the actual communication, and which implement the interface(s) of the distributed object at the client's side, are selected, installed, and initialized. By including a separate selection step for object implementations, possible heterogeneity of the underlying system is completed shielded for the client.

The binding process is entirely done during runtime. There is no need for configuration of components at the client's side beforehand. The development of distributed naming service that scales to wide area networks is a subject of ongoing research.

Structuring. Composition of distributed objects is presently limited to a single address space. To explain, assume that two local objects of different distributed objects reside in the same address space. In that case, we can aggregate these two local objects into a composition by providing an interface consisting of entries that point to methods from either local object. Consequently, to a client residing in that address space, the composed distributed object appears as a normal object. Composition of distributed objects in general is a also subject of ongoing research.

4 Discussion and related work

Our model has a strong focus on encapsulating and hiding implementations, and provides flexibility with respect to how interfaces are implemented. This contrasts the approach followed by others.

For example, objects in CORBA are based on the existence of a runtime system, called an object request broker [5]. This request broker is augmented with object adapters that provide the basic means for invoking methods of (remote) objects. The implementation of the broker and its adapters is not part of any object. And because they offer only limited communication facilities, it becomes difficult to construct, for example, groups of replicated objects. Making the runtime system open to adaptations by developers would have avoided this problem.

Fragmented objects [3] follow an approach quite similar to that of ours, except that the runtime selection of a peer group and its associated communication protocol is not supported. Instead, the actual selection criteria have to be provided by an implementor beforehand. In our case, we let a client follow its own selection strategy, constrained, of course, by what the distributed object offers.

The approach followed in Spring [4] is also more or less in line with ours. Flexibility with respect to implementations is obtained through so-called subcontracts [1], which can be perceived as commonly agreed protocol stacks between two communicating local objects. However, subcontracts are not structured as objects in the way we envisage. Instead, they appear as indivisible units.

A notable difference with other approaches is that the architecture we have outlined in this paper, is aimed towards support for wide area applications.

References

[1] G. Hamilton, M. Powell, and J. Mitchell. "Subcontract: A Flexible Base for Distributed Programming". In *Proceedings 14th Symposium on Operating Systems Principles*, Asheville, North Carolina, December 1993. ACM.

[2] P. Homburg, L. van Doorn, M. van Steen, A. Tanenbaum, and W. de Jonge. "An Object Model for Flexible Distributed Systems". In *Proceedings 1st Annual ASCI Conference*, pp. 69–78, Heijen, The Netherlands, May 1995.

[3] M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M. Shapiro. "Structuring Distributed Applications as Fragmented Objects". Technical Report 1404, INRIA, January 1991.

[4] J. Mitchell et al. "An Overview of the Spring System". In *Proceedings Compcon Spring 1994*. IEEE, February 1994.

[5] Object Management Group. "The Common Object Request Broker: Architecture and Specification, version 1.2". Technical Report 93.12.43, OMG, December 1993.

[6] L. van Doorn, P. Homburg, and A.S. Tanenbaum. "Paramecium: An Extensible Object-based Kernel". In *Proceedings Hot Topics on Operating Systems V*, Orca's Island, Washington, May 1995. IEEE.