An Object Model for Flexible Distributed Systems

Philip Homburg *
Leendert van Doorn
Maarten van Steen
Andrew S. Tanenbaum
Wiebren de Jonge

Vrije Universiteit
Amsterdam

March 29, 1995

1 Introduction

Current distributed applications like E-mail, electronic news, distributed calendars, and network information browsers often have a complex structure. This is partly due to lack of suitable support from the underlying operating system, which is often too low level. At the same time, multi media applications, parallel programs, wide-area applications, and database systems have very different operating system demands. Although it is possible to put support for the above-mentioned systems in a single operating system kernel, this is generally undesirable from a software engineering point of view, as the resulting software may become unmanageable.

In this paper we describe a new model for constructing operating systems and applications in an integrated fashion. Compared to current approaches we provide high-level primitives for supporting distributed and parallel applications. We also provide the flexibility to configure both applications and kernels to only include the functionality that is actually used.

The model we describe is based on objects. Objects are used to structure both applications programs and operating system kernels. They also provide the application interface to the operating system kernel, and access to hardware devices for both kernels and applications. By providing structuring mechanisms for large (distributed) objects, we believe that applications will be are easier to build. At the same time we provide flexibility by allowing extensions of operating system kernels and applications with new objects at run time[8], and by providing a way to bind to objects dynamically.

An important aspect of a distributed system is the scalability of the system [6]. A scalable system should not depend on centralized resources or on algorithms that need global information. At the same time, a flexible system can use different algorithms depending on the situation. For example, the use of broadcasting and multicasting on a local Ethernet can be quite effective but should be avoided on a world wide scale.

In this paper we discuss an object model that provides two kinds of objects: local objects and distributed objects. In Section 2 we describe the nondistributed (local) objects, followed by distributed objects in Section 3. We compare our work to that of others in Section 4.

2 Local objects

In our model, an object logically consists of three parts:

- The object's current "value" or **state**.

*This research is in part supported by a grant from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (N.W.O.)

- The object's **methods**. These methods are the only way to inspect or modify an object's state.
- A collection of **interfaces**. An interface is a collection of methods. Methods are invoked via interfaces.

Our objects are passive, language independent, and conceptually self contained. Furthermore, dynamic configuration is supported by loading objects at run time from an object repository (a tool-box of components) and by a flexible object naming system. In our model, threads are the active elements that invoke methods.

Objects can be divided into two categories: local objects and distributed objects. **Local objects** are objects that are restricted to one address space. This in contrast to distributed objects that can span multiple address spaces. Distributed objects are discussed in Section 3.

In this paper we only discuss one kind of local object, the so called primitive object. The other kind of local object is the composite object. **Composite objects** are objects that are built out of other objects. We also make a distinction between system objects and normal objects. Normal objects are centered around state (i.e. they provide access to, and operations on, some specific state). In contrast, system objects provide access to operating system entities like threads, and to (logical) devices.

As we have said, an interface is a collection of methods that provides access to the state of the object to which the interface belongs. At run time, an interface is an array of $(methodpointer, statepointer)$ pairs. Multiple copies of the state pointer are stored in an interface table to support composite objects. The method pointer points to the start of the executable code that implements that particular method, and the state pointer points to the state of the object on which the operation should be invoked. This scheme allows multiple objects to share the same method implementation, and also gives some freedom as to where a particular method implementation is stored. From another perspective, we view interfaces as a design tool. By designing interfaces that support a wide variety of implementations (in different objects) a system gains flexibility. The way inheritance or delegation at a language level is translated to interfaces is beyond to scope of this paper.

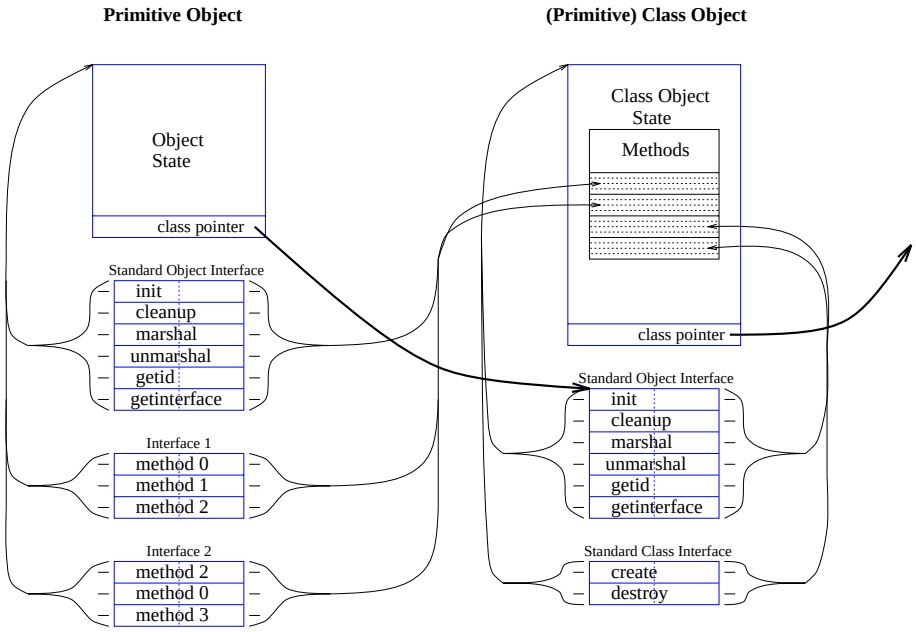

Figure 1: A primitive object and its class object

A primitive object is the building block for more complex objects such as distributed objects. Therefore, to illustrate the concepts introduced so far, we discuss an example primitive object first.

Primitive objects consist of three parts: state, methods and interface tables. These parts, together with a class object are shown in Figure 1. The state of an object contains among other things, the instance variables of that object. The example shows five interfaces: a "Standard Object Interface," "Interface 1," and "Interface 2" belonging to the object, and a "Standard Object Interface," and "Standard Class Interface" belonging

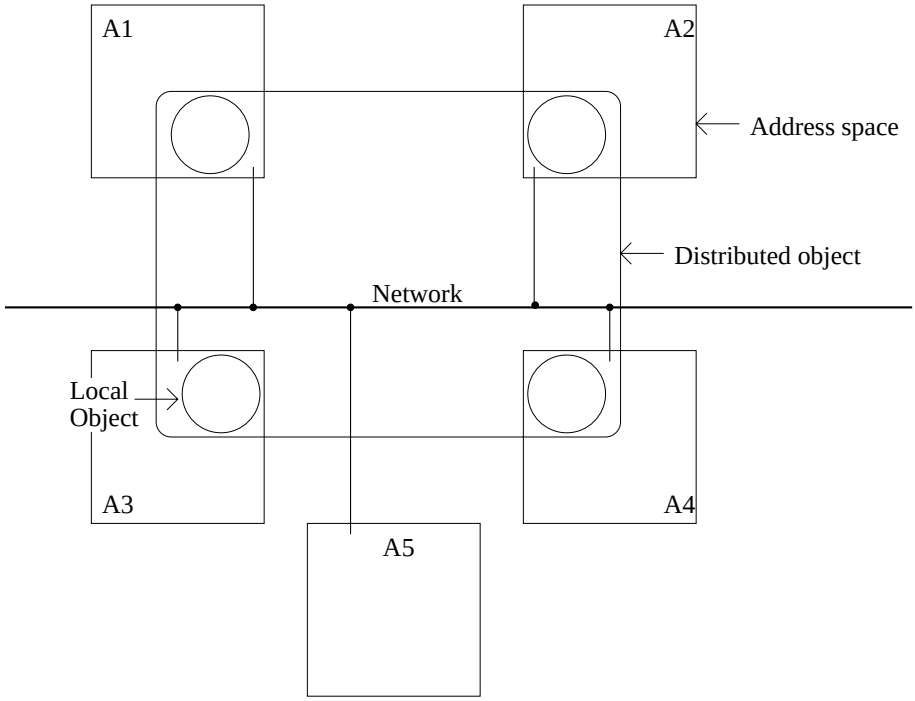

Figure 2: A distributed object

to the class object. In this example, the methods (actually, the machine code that results from compiling the method implementations) are stored in the class object.

All objects have to provide a so-called "Standard Object Interface." Objects of certain types (e.g. file objects, thread objects, integer objects) also have extra interfaces that correspond to the particular type. For example, class objects must provide a "Standard Class Interface." The primary function of a class object is to support creation and destruction of objects of the "class" represented by the class object. But a class object also provides a convenient place for storing the methods of instances of that class. The methods of object can be placed either in the object itself or in the state of its class object. In most cases, the method implementations of an object will be stored in its class object to support sharing of the executable code between instances of the same class. If an object is one of a kind, the method implementation may be stored in the object itself. For example, the methods of the class object in this example are stored in that same class object.

3 Distributed Objects

The objects described so far (local objects) are limited to one address space. This implies that both the state and the interface instances are in a single address space. To be able to create objects that span multiple address spaces we introduce distributed objects. A **distributed object** is an object that can have interface instances in multiple address spaces, or can have its state spread out over multiple address spaces, or both.

Distributed objects can be implemented in different ways depending on the partitioning and replication of the state [1] and on the way in which the overall state is kept consistent. The state is said to be **partitioned** if it is split up in a number of disjunct parts that are stored in different address spaces. The state can be **replicated** by storing copies in different address spaces. These two techniques can be combined by replicating partitions.

To have distributed objects, one needs communication to keep an object consistent if the state of the object is replicated. It is also needed if a method is invoked on an object but not all the state of the object that is needed by the method implementation is available in the address space in which the method was invoked. For example, a simple distributed object might keep its state in one address space and have multiple interface

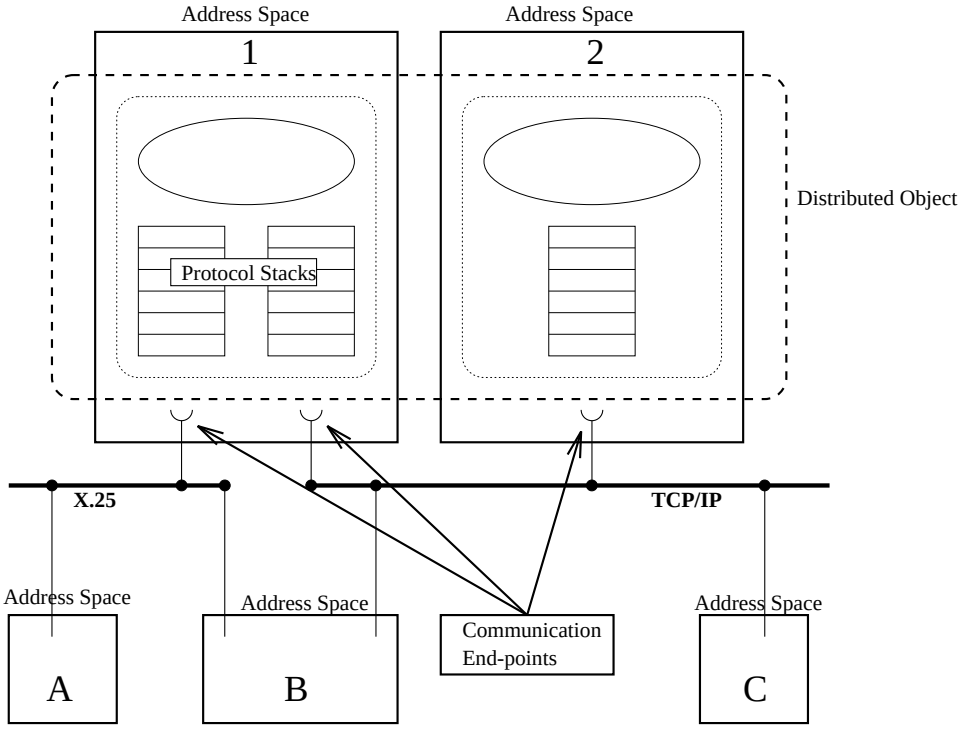

Figure 3: A distributed object with multiple protocol stacks

instances in other address spaces. A method can then be invoked in an address space where the state is not locally available, so it needs to be forwarded (e.g. using RPC) to the address space which keeps the state of the object. Another example is an object with fully replicated state. In this case, each address space with an interface instance also contains a full copy of the object's state. When a method is invoked that changes the state of the object, all other address spaces that keep a copy of the state must be informed about the change. When this happens is object dependent.

We structure distributed objects as a collection of communicating local objects. In Figure 2, we see a distributed object which spans 4 address spaces: A1 through A4. Each of these address spaces contains a local object that *represents* the distributed object, by providing the interfaces of the distributed object. Together, these local objects implement the distributed object. Method invocation on a distributed object is only possible in a given address space if that address space contains a local object that is part of the distributed object. The process of installing and initializing such an object in an address space is called **binding**.

Two aspects of the implementation of the local objects are: (1) the structure of the distributed object, and the role the local object plays in the distributed object, and (2) the communication protocols which are supported by the local object. The first aspect comes from the fact that distributed objects vary in partitioning, replication, etc. This is reflected in the implementations of the local object. The second aspect is due to the varying communication demands of distributed objects. For example, small distributed objects with centralized state can be implemented best using a low-latency RPC protocol. Distributed objects with many replicas work best with multicast and/or group communication protocols, and objects that move large amounts of state back and forth benefit from bulk data transfer protocols.

Note that the best communication protocol depends on the internal structure of a distributed object, and not primarily on its type. A simple file object can be implemented in various ways: with many replicas and a voting mechanism, or partitioned, or just stored in one place. This means that multiple communication protocols should be supported in a transparent way. The *user* of the object should not be concerned with these details: the user only invokes methods on the object.

Figure 3 shows a distributed object with local objects in two address spaces. The local object in address space 1 supports two protocol stacks; one based on TCP/IP and the other one based on X.25. The local object

in address space 2 supports only TCP/IP. In many cases the upper communication layers of a protocol stack (the layers that deal with data representation, consistency of replicas, etc.), are independent from the lower layers (that deal with host addressing, routing, etc.). We call instances of these lower layers **communication end-points**.

The distributed object in our example can be reached at three communication end-points, one X.25 end-point in address space 1, and two TCP/IP end-points in address spaces 1 and 2. In the figure we see the address spaces, A and B, connected to the X.25 network. Address space B is also connected to the TCP/IP network together with address space C. Different communication end-points can be used depending on the address space from which the binding takes place. Address space A can only use the X.25 end-point, address space B can use all three end-points, and address space C can use the two TCP/IP end-points.

Note that if we bind to the distributed object from address space A, we need a local object that can communicate using X.25. If we bind from address space C we need a local object that speaks TCP/IP. Depending on the communication protocols we may need different implementations of the local objects, but the interface which is presented to the user of a distributed object is always the same.

In the remainder of this section we will describe an algorithm for the naming [10] of and the binding to distributed objects. In general, a distributed object is created in one address space, and registered under a certain name with a name service. After that, other address spaces can bind to that distributed object. The function that implements binding accepts the name of a distributed object, and returns a pointer to a standard object interface that can be used to access the distributed object. During this binding process new executable code can be loaded in the form of class objects. The ability to load new objects depending on the particular distributed object that is accessed allows straightforward object implementations without leading to inflexible systems.

Basically, the binding system should:

- Provide a way to refer to (to name) existing distributed objects.
- Determine the "location" (in terms of communication end-points) of the object.
- Select and load a suitable implementation (class object) for the local object, and instantiate and initialize this local object.

3.1 Object Naming

Distributed objects can be named in various ways. Object naming based on object identifiers is one extreme: when an object is created it is assigned an identifier that uniquely identifies that object. At the other extreme, objects are addressed by contents or functionality. For example, the object name "printer" can be used to refer to the local printer. Many systems, including ours, offer an intermediate solution: a user-chosen name is used to refer to a single object. One name refers to one object at the same time but can be bound to different objects at different times.

In most distributed systems, the naming system maintains a mapping from names to network addresses. A disadvantage of this approach is that support for objects that migrate, or objects that are replicated is hard. Updating the network addresses is especially difficult if multiple names refer to the same object. We propose to separate naming objects from locating objects. A lookup of a name in the name service returns a location independent object handle. This object handle is passed to the location service which maps the object handle to multiple communication end-points. The properties of the object handle are discussed below in Section 3.3. This extra level of indirection allows multiple names to refer to the same object while at the same time the actual location of the object is maintained in one place only (logically that is, the location service might be replicated).

3.2 Object Location

We use a single distributed location service to keep track of where objects are. The location service as a whole stores the location information about all distributed objects. Since the number of distributed objects can be extremely large, the complete state of the location service needs to be partitioned. These partitions are stored in location objects. Each location object keeps the location information of some number of distributed objects. Location objects can also refer to other location objects.

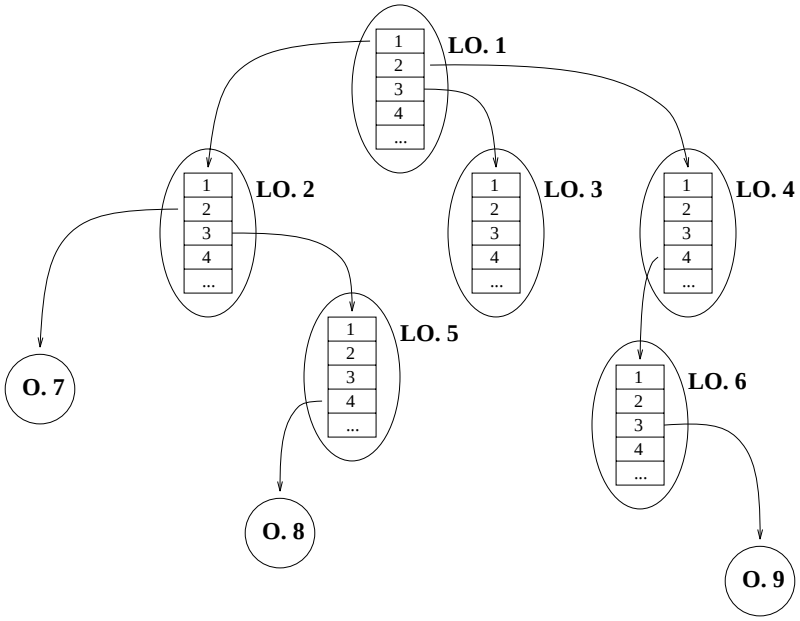

Figure 4: the Distributed Location Service

Figure 4 shows how such a location service can be organized. The figure shows nine objects, six location objects (LO.1, ... LO.6), and three ordinary distributed objects (O.7, ..., O.9). LO.1 is the root location object. Each location object maps a small index to the location information of another object. This other object can be a location object or any other distributed object. The global location index of an object is a concatenation of the indexes of the location objects that form the path from the root location object. For example, the location index of object O.7 would be 1.2, and O.9 would be 2.4.3.

When an object is created, it registers its location information (the addresses of its communication end-points, and the corresponding protocol stacks) somewhere in the location service, in a location object. This location object returns the location index. This location index is then used to create an object handle to allow the object to be registered with the name service. The object informs the location object where it is registered when its list of communication end-points is changed. This happens, for example, when the object moves, or when the state of the object is stored in additional address spaces.

A location object is just a regular distributed object. This means that the state of a location object can be replicated and that different location objects can use different replication strategies, different consistency protocols, etc. Objects can be registered with different location objects depending on the locality, replication, etc., of the object. In general, it makes sense if an object is registered at a location object at a short distance (from a network perspective). Similarly, an object with a high replication degree should be registered at a location object that is also widely replicated.

3.3 Algorithm for Naming and Binding Distributed Objects

The naming/binding algorithm consists of four steps: Name Lookup, Location Lookup, Destination Selection and Implementation Selection. We will describe each step briefly before going over the details.

Step 1: Name Lookup

In the first step, a name is resolved to an object handle. This object handle consists of two parts: an object identifier and a reference to the location of the object. This object identifier is (world wide) unique and location independent, and is used as an end-to-end check for the location lookup. The reference to the object location functions as an index in the location service. This effectively decouples the name of an object from the actual location of that object.

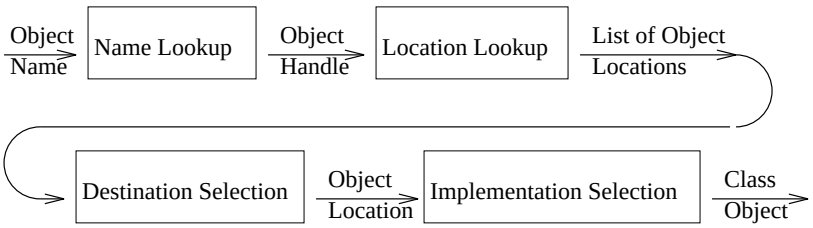

Figure 5: the Naming/Binding process

Step 2: Location Lookup

In the second step, the location reference and the object identifier are resolved to a set of $(address, protocolID)$ pairs. Each element of this set describes the network address, port number, etc., where the object can be reached, and the protocol stack that has to be used to communicate with that particular destination.

Step 3: Destination Selection

In the third step, one $(address, protocolID)$ pair is chosen. This selection should be made, based on the protocol implementations available to the address space from which binding takes place, security constraints, and locality.

Step 4: Implementation Selection

In the fourth step, a suitable implementation for the selected protocol is chosen. We assume that implementations are available on a nearby site which acts as an *implementation repository*. Implementations are stored as class objects; new ones can be added at dynamically. Once a class object is loaded, it can be used to access multiple distributed objects of the same class. To access a single object from all over the world, we use world wide unique protocol identifiers that are resolved locally to a class object that implements that protocol.

After the class object which has been selected in step 4 has been loaded, it is used to create a new local object. This local object is then initialized with the distributed object's ID and the address of the chosen communication end-point. The distributed object can now be accessed through the local object. Depending on the actual implementation of the class object, the local object can act as a proxy forwarding requests to another address space, or it can get a copy of the current state of the distributed object and act as a new replica. Many intermediate configurations are also possible.

3.3.1 Name Lookup

The first step of the name lookup is to parse the name into a starting point and a number of components. For example, a UNIX style name like */foo/bar* results in two components *foo* and *bar* and the root as starting point for the name lookup.

The organization of the name service is such that components are looked up in lookup tables, which are implemented by (logically) independent directory objects. That means that different directories can have different replication strategies, different consistency guarantees, etc.
A name lookup on a directory object is functionally:

$$dirObject.lookup([components]) \rightarrow objectHandle,\ [components]$$

The lookup gets as arguments a list of components and returns a reference to another object, and a new list of components.

The object handle returned by the lookup method can either refer to the requested object (the object denoted by the name) or to a directory object. If the list of components returned by the lookup method is not empty then the object handle belongs to a directory object, and that directory object should be used for the next lookup. On the other hand, if the component list is empty then the name lookup is completed and the returned object handle is the object handle of the named object.

3.3.2 Location Lookup

The final result of the name lookup step is an object handle. This object handle consists of two parts: a location independent object identifier and a location reference that can be used as an index in the (world wide) distributed location service.

First, that location index is split into a list of components, which are indexes in individual location objects and an object handle for the root location object is obtained.

Starting at the root we repeatedly look up some components until the desired location information is found. A lookup on a location object looks like:

$$locObject.lookup([components],\ objectID) \rightarrow \\ [(protocolStackID,\ address)]\ [components]$$

The lookup method gets as parameters the current list of components, and the object identifier. The result of the lookup is a list of tuples that contain a protocol stack identifier and an address, and also a new list of components.

The list of tuples specifies the protocols and addresses of communication end-points that are used by the object to communicate. Just like in the name lookup step, if the new list of components is empty we have the desired information. If the list of components is not empty, a new lookup is needed. The new lookup is a method invocation on the location object whose addresses are in the list. These addresses are converted to an object handle through the destination selection and implementation lookup steps described below.

The object ID is also passed as an argument to the lookup method to support the implementation of the location object. For example, this allow shortcuts: a location object half way on the path can also store the location information of an object at the end of the path.

3.3.3 Destination Selection

The result of the location lookup step is a list of $(protocol, address)$ pairs. The purpose of this step is to select the best $(protocol, address)$ pair from the list. Destination selection looks like:

$$selObject.select([(protocolStackID,\ address)]) \rightarrow \\ (protocolStackID,\ address)$$

$selObject$ is a reference to an object that performs the selection. The details of how to do this selection are beyond the scope of this paper. The select method gets a list as parameter and returns one element of that list. Note that it should be possible to backtrack if failure occurs later on.

Destination selection solves two problems. First, it allows a single object to support multiple protocols. Destination selection filters out unsupported protocols and can prefer one protocol over another. For example, MPEG video compression is very efficient if a hardware MPEG implementation can be used, but can be unusable if MPEG decoding has to be done in software. This suggests that different protocols should continue to coexist. Second, multiple addresses support replication. Replication can be used to increase availability and to improve locality.

3.3.4 Implementation Lookup

So far, the steps deal with abstract data items like names, location information, protocol stack identifiers and addresses. The purpose of the implementation lookup step is to load a suitable class object into the caller's address space. This class object is then used to instantiate a new local object. If initializing this object succeeds, we have a local object that represents the distributed object.

The selection of the class object is based on the protocol stack ID. This identifier should have the following properties:

- It should uniquely identify the relevant protocol stack that should be used, for example SunRPC over UDP over IP.

- It should uniquely identify the data encoding and consistency protocols used.
- It may specify extra security protocols or demultiplexing layers.
- And finally, it should specify which interfaces the object should offer to the outside world.

Implementation lookup looks like:

$$classRep.lookup(protocolStackID) \rightarrow classObject$$

$classRep$ is a reference to an object that acts as a class repository. This object can implement a search path: private class repositories are searched before system-wide ones.

4 Conclusions and Related Work

In this paper we have described an object model that can be used to structure operating system kernels, runtime systems and distributed applications. We can divide related work in two categories: other systems that provide ways to structure operation system kernels and runtime systems, and systems that provide support for distributed applications.

In our model, we use local objects as a structuring mechanism and to gain flexibility. Two other systems that are based on objects are Choices[3] and Clouds[9]. Choices provides a structuring mechanism called "Frameworks." These frameworks are mainly used during the design of the system and are represented by C++ classes at run time. Where Choices uses objects to structure the operating system kernel, Clouds (version 2) supports objects on top of a small micro kernel called "Ra." Our approach differs from both Choices and Clouds in our attempt to provide a single object model that can be used for (structuring) objects in applications (runtime systems), operating system kernels, and communication. For a comparison of our work to current projects that deal with flexible operating system kernels see the paper by L. van Doorn, et al. [8].

Other systems that encapsulate communication in objects include Corba[7], Spring[4], S.O.R.[5] and Network Objects[2]. The main difference between these systems and our model is the way access to a distributed object is provided. Both S.O.R. and Network Objects support "remote objects": a method invocation in one address space is transparently transferred by the local runtime system to the address space where the object resides. Both systems lack support for replication, other than layered on top of the remote objects. In Network Objects, an object always stays in the same address space. S.O.R. supports object migration.

Corba and Spring are more complicated but basically have the same model. Corba supports "request brokers" to get some flexibility. Spring supports "subcontracts" that can be used to implement caches, support for replication, etc. Subcontracts provide a client stub with five operations: marshal, unmarshal, marshal_copy, invoke_preamble, and invoke. Client stubs implemented using these operations can be used with different subcontracts. Example subcontracts deal with simple remote objects (singleton subcontract), replication (cluster subcontract), caching and persistency.

In the four systems mentioned above, the client stub is provided by the runtime system and is not part of the distributed object. In contrast, in our model the local object that provides the interface to the distributed object is logically part of the distributed object. This local object is (again logically) self-contained and provides its own communication mechanisms. This means that completely new ways of structuring distributed objects can be deployed without changes to the model.

References

[1] G.R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[2] A. Birell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. In *14th Symposium on Operating Systems Principles*, pages 217–230, Asheville, North Carolina, December 1993. ACM.

[3] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. Choices, Frameworks, and Refinement. In *Proc. Workshop on Obj. Orientation in Op. Sys.*, pages 9–15, Palo Alto, CA, October 1991. IEEE Computer Society Press.

[4] Graham Hamilton, Michael L. Powell, and James G. Mictchell. Subcontract: A Flexible Base for Distributed Programming. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79, Asheville, NC (USA), December 1993.

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

[6] B.C. Neuman. Scale in Distributed Systems. In T.L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 463–489. IEEE Computer Society Press, Los Alamitos, California, 1994.

[7] Object Management Group. The Common Object Request Broker: Architecture and Specification. Technical Report 91.12.1, December 1991.

[8] L. van Doorn, P. Homburg, and Andrew S. Tanenbaum. Paramecium: An extensible object-based kernel. In *IEEE workshop on Hot Topics in Operating Systems*, 1995.

[9] C. J. Wilkenloh, U. Ramachandran, S. Menon, R. J. LeBlanc, M. Y. A. Khalidi, P. W. Hutto, P. Dasgupta, R. C. Chen, J. M. Bernabeu, W. F. Appelbe, and M. Ahamad. The Clouds experience: Building an object-based distributed operating system. In *Distributed and Multiprocessor Systems Workshop Proceedings*, pages 333–347, Fort Lauderdale, FL, October 5-6 1989. USENIX.

[10] A.K. Yeo, A.L. Ananda, and E.K. Koh. A Taxonomy of issues in Name Systems Design and Implementation. *ACM Operating Systems Review*, 27(3):4–18, July 1993.