A Framework for Describing Distributed Object Models

Ihor Kuz

September 13, 1996

Abstract

Computer networks are becoming increasingly more popular and more widely used. In order to take full advantage of these networks, new distributed programs have to be written. Many of the tools available for distributed programming, however, are unsatisfactory because too many of the problems faced during distributed programming are left up to the programmer. A solution to this is middleware. Middleware is a computing environment for distributed applications which provides a model of information exchange and a common programming interface to application programmers. One popular form of middleware is that of distributed objects where communication is done through shared objects. A problem with the popularity of this type of middleware solution is that there are so many different models of distributed objects and no common way of describing or comparing them. This report presents a common framework and terminology for describing distributed object models, as well as descriptions of a number of models.

Contents

Chapter 1

Introduction

There are two main reasons for the current popularity and importance of networks and distributed programming. The first is the low cost of powerful computers and improvements in network technology. These make it easier and more feasible to connect computers into networks. Of course, just the fact that something is possible doesn't necessarily make it desirable, it must also be useful to people. The second reason then has to do with the benefits that computer networks have for their users. Networks allow hardware and software resources such as disks, printers, programs and databases to be shared by multiple users. They can also be used to increase the performance of certain types of applications, as well as to increase the reliability and fault tolerance of applications and data through replication. Communication between users is also an important possibility offered by networks. There are even some types of applications such as cooperative applications or games which are inherently distributed and can only exist with a network of some sorts.

In order to benefit from interconnected computers, applications have to be specially designed and written to take the network environment into account. Unfortunately writing distributed applications is not easy. The biggest problem is that much of the burden of managing communication between the different parts of the application is often left up to the programmer. Not only the communication but also reliability, security and concurrency problems usually have to be taken care of by the programmer. This means that every time that a new application is written all of these issues must be taken into account and often re-implemented. A problem often caused by this is that the applications are written in such a way that changing the distribution mechanisms, e.g. the network protocol, is a very difficult task. Another problem is that many existing non-distributed applications cannot easily be integrated into distributed applications, nor can they take advantage of the distributed environment.

The solution to these problems lies in middleware. Middleware's main goal is to provide a consistent computing environment for distributed applications. It consists of a software layer above the operating system and networking layer and below the application. This layer provides a model of information exchange and a common programming interface to the application. In this way details of the mechanisms used for distribution (e.g. interaction with the network software and operating system) are hidden and can be replaced without having to change the programs using them. This computing environment also provides the services, such as reliability, security and concurrency control, needed by many distributed applications. Middleware may also allow legacy applications to be accessed by or be part of distributed applications. Apart from keeping the communication, distribution and other services transparent, the environment provided by most middleware can play an important role in structuring the applications.

One class of middleware that has received much attention and acceptance is that of distributed objects. This type of middleware combines object-oriented programming and distributed programming by focusing on objects and the interaction between them. In object-oriented programming the basic idea is that all the state of a program is encapsulated by objects. Objects are entities which contain data and define operations to access and modify that data. An important feature of objects is data encapsulation, which means that an object's data is not visible from outside the object and can only be accessed through the operations (called methods) defined in its interface. Another important feature of the object oriented paradigm is inheritance. This allows existing objects (or classes, which are the definitions of objects) to be extended, creating new objects (or classes). In this way reuse is facilitated because new objects do not have to be built from scratch but can be based on existing ones, and users of previously written objects can easily extend functionality without having to change the original object.

The basic idea behind distributed objects is that all communication is done through objects. These objects are usually mobile and communicate mainly through invoking methods on other, possibly remote, objects. Whether an object is remote or not, and how the method call is transferred to the remote object should be transparent to the user of that object.

One of the reasons why object-oriented programming and distributed programming go together well is due to the structure of most distributed applications. Many distributed programs are structured as separate communicating entities, for example clients and a server, or a resource and its users. This maps naturally onto a model of communicating objects. Object boundaries also provide a good place to impose security restrictions and object accesses provide a chance to perform auditing. Migration of data or processes can be modeled by migrating objects and because an object is basically defined through its interface, the actual implementations of objects can differ depending on their surroundings (e.g. the underlying hardware or operating system). It is possible to achieve reliability and performance gain by replicating objects. The reuse and inheritance mentioned above also allow previously developed objects to be reused and extended in new systems with minimal effort, as well as allowing existing systems to be easily extended or modified.

Not all of these are trivial matters of course. For example, implementing efficient replicated services and process migration are areas where research is still being done in and the combination of inheritance and distributed programming can often cause problems. However, once the system is implemented, the programmers do not have to worry about these details and can concentrate on their specific applications.

The merging of objects and distributed programming has been widely accepted and as a result many different models have been proposed and implemented. The problem is that there is no common terminology much less a common framework for describing these models. This has led to much confusion because it is not always easy to see the differences and similarities between the various models. The goal of this report is then to define a common terminology and present a framework for describing models for distributed objects.

The rest of the report will be structured as follows. In the next chapter the common framework and terminology will be presented and explained. After that the framework will be used to describe a number of models. The conclusion will discuss the framework and will present some general remarks about comparisons of the models.

Chapter 2

The Framework

A common framework for models of distributed objects should provide a basic structure for a description of each model. This description should allow the structure and working of a model to be quickly discerned. It should also allow one to easily see the availability or lack of any features, and should function as a base for a comparison of various models for distributed objects.

The framework described here breaks a model up into a number of important issues in distributed objects. Each issue is also broken down into a number of aspects, which include definitions of terms and descriptions of features found in models of distributed objects.

The approach taken in this chapter will be to first introduce some terminology used in describing the framework. Then the issues and their aspects will be explained. This will provide a description of the framework, as well as a 'how to' guide, where necessary giving tips and information about how to use the various aspects when drawing up a description of a model. Finally an overview of all the issues and aspects will be presented.

2.1 Overview of the Framework

2.1.1 External vs Internal View of a Model

When describing a model we can either take an external or an internal view The outside (or external) view is how a programmer using the model would view a feature, including the syntax and semantics. The inside (or internal) view describes what a feature looks like 'under the surface' of the model. This includes internal semantics and a description of the actual implementation of the system (if deemed necessary for further understanding of the model).

In the description of a model most entries will include information about the external and internal views. For some aspects of a model this separation of views is not necessary, and will be explicitly stated when appropriate.

2.1.2 Terminology

Model (when referring to a distributed objects model) refers to the descriptive part of a distributed objects model, whereas **system** refers to an implementation of the model - its run time system. **Process** and **thread** are used interchangeably and refer to an active entity. An object's **client** always refers to a process or object which accesses that object (i.e. calls the objects method). The terms **user** and **programmer** always refer to people using the objects in the model to write a program or application and to people programming objects.

An **address space** is the memory directly accessible by an active entity. **Machine** refers to a physical computer (assuming single processors) or CPU (with multiprocessors). Different address spaces may be on the same or on different machines. A **domain** is a collection of address spaces managed by a single domain manager. A domain can span multiple machines or multiple domains can be present on one machine.

2.1.3 Main Issues

The main classes of issues covered in the framework include: objects, object management, object location, interaction with objects, concurrency, object oriented programming issues, typing, persistence, failure and fault tolerance, programming interface, and system issues.

2.2 Objects

Generally an object consists of a state, methods for accessing the state, and one or more interfaces which specify how clients can access the object. Each model may however deviate from this structure.

2.2.1 Objects and Object Structure

Granularity. The first aspect is the *granularity* of the object model. Granularity refers to the size of the objects managed by the system. In large grain models the objects usually encapsulate whole applications and smaller entities (like data structures) are not visible to the system. In medium grain models the objects represent medium sized data structures (like lists or records) as well as larger entities (as in the large grain model). The fine grained models treat everything as objects, this includes even the smallest data types (like integers). For this aspect no external/internal division is needed.

Kinds of Objects. Some models may distinguish different *kinds of objects*, for example system and user objects, or local and remote objects. Often a user sees different kinds of objects (external view) than the system does (internal view). The *object structure* is also an important aspect which may be seen differently by the user and the system. Object structure is a definition of what an object actually IS. This includes a description of the parts that constitute an object (for example an object may have a state and methods).

Composition. Object *composition* refers to creating objects out of other objects. One common way of doing this is through nesting. A nested object is one that is included in another object's state. Important here is whether objects can be nested at all, and if so how this is done (e.g. by including the whole object in the state, or by including only a reference in the object's state), and if there are any restrictions as to what kinds of objects may be nested (e.g. only local objects can be nested). The external view refers to how a user would include an object and the internal view deals with how the nesting is actually done at the system level (these two do not have to be the same).

2.2.2 Interface

An object's interface refers to how clients access the object. This aspect deals with the possibilities (or lack thereof) for multiple and evolving interfaces. Some models allow objects to have multiple interfaces, meaning that different clients of an object may see different interfaces, and therefore have different ways of accessing the object. The interfaces may be completely different or may be restricted subsets of other interfaces. Another possibility offered by some models is evolving interfaces, which means that an object's interface may change during the lifetime of that object. In both cases the external view refers to how a programmer and user of an object can manipulate or use the multiple or evolving interfaces. The internal view refers to how multiple and evolving interfaces are actually handled by the system.

2.2.3 Activity Model

The activity model deals with the relationship between processes and objects. Three kinds of activity models are possible: passive, active and hybrid. In the passive model processes and objects are separate entities. The active entities communicate through shared objects[1]. In the active model processes and objects are bound to each other, that is every object has a process that is bound to it and every process is bound to an object. Another aspect of the active object model is that all active entities communicate directly. Each object has a main thread bound to it which receives messages from other threads and arranges

[1] Other communication mechanisms are possible, however this is the only one that we will discuss

that the appropriate action is taken (this usually entails a worker thread being activated and executing the appropriate method). The hybrid model encompasses combinations of the above two, for example a model which allows both active and passive objects to coexist. This aspect requires no external/internal division.

2.3 Object Management

With object management we refer to how the system and users keep track of objects. This includes managing the object's life cycle as well as how objects can be found and referred to by users.

2.3.1 Object Life Cycle Management

An object's life cycle consists of being defined, being created, executing methods as requested by its clients and then being destroyed. Here life cycle management refers to taking care of the *definition*, *creation* and *destruction* of an object.

Definition. When an object is defined (or declared) usually the data types in the state, the methods and the interface(s) are defined. How a user defines an object belongs to the external view, whereas what actually happens when an object is defined (e.g. a class object is created) may be both part of the external and internal view.

Creation. Once an object is defined it must be created (sometimes the definition and creation are merged into one action). This usually entails memory being allocated, and perhaps a process being activated as well as the user receiving a reference to the object.

Destruction. Once an object is no longer needed it must be destroyed, in some models this must be done explicitly by the user, while in others objects are garbage collected when they are no longer needed. Once destroyed an object is gone for good and can no longer be accessed by any clients. In both cases the external view specifies how the creation or destruction is done by the user and what the user perceives, whereas the internal view describes what is done by the system.

2.3.2 Class and Class Objects

In many models objects are in fact instances of a class. A class is the description/definition of an object and it is the class which is defined, rather than the object during the definition phase of the life cycle.

Often the class is itself an object which is created upon definition. If a model includes class objects then a description must include information about whether class objects are normal objects (i.e. can be accessed by other objects or processes, also have a class, etc.), whether they impose any restrictions on the objects (e.g. when an object moves its class object must move with it), information about sharing of class objects (by objects which are instances of the same class) and any other peculiarities which class objects might introduce.

2.3.3 Object Naming

In order for potential clients to find an object some sort of name service must be included in a model. A name service allows objects to have names and allows objects to be found using these names. A potential client uses the name service to resolve names, that is given a name it can get a reference to the object with that name. In order to be known by the name service an object must register itself under a specific name.

There are many possible structures for a name service and a description should include the structure of an object name, how objects register themselves (or get registered) and how name resolution takes place both from an external as well as an internal view.

2.3.4 Object Reference

In order for a client to do anything with an object it must first be able to reference it. This could mean a number of things including having access to the object through a memory pointer and having the actual object in its address space and being able to access it directly. Usually there is a big difference between the way that a user sees a reference and the way that a reference is actually implemented (or seen by the system). Also one must take into account the possible difference between referencing different kinds of objects (e.g. local and remote).

2.4 Object Location

The whole idea of distributed objects is that objects on different machines (or at least in different address spaces) can communicate with each other. Therefore object location is an important issue. However, object location is a difficult concept to define in general and it is used differently in many models.

Considering an object as described in Section 2.1.1 (an object has a state, methods and interfaces) an object's location will be a combination of its state and interface location. The state location refers to the complete set of address spaces where all the parts of the state can be found.

The interface location is the set of address spaces (on possibly different machines) where the interface can be accessed. For message-passing interfaces this would be where the messages can be sent to, while for direct invocation this would be where the methods can be invoked.

2.4.1 Location Transparency

In order to make programming easier many models try to hide the fact that some objects may be remote and others local. If that difference is not hidden then the actual location of remote objects usually is. Location transparency refers to two concepts: whether clients know that the objects that they reference are remote, and if so whether they know the location of the objects that they reference. No internal/external division is needed for this aspect.

2.4.2 Location and Finding Objects

On the system level the location of an object often needs to be made explicit (e.g. to access a remote object). Sometimes (depending on the location transparency) the location also has to be made explicit to the user. The location aspect refers to how an object's (or any other) location is specified. Finding objects refers to how the location of an object is determined given a reference to that object.

2.4.3 Mobility

In a distributed object model objects can often be moved or copied to different address spaces on possibly different machines. *Moving* an object changes its location while *copying* creates a new object based on the original object. The external view describes how a user can explicitly move or copy an object and what they perceive happening, the internal view describes what actually happens when an object is moved or copied.

2.4.4 Distribution

Distribution of an object refers to how an object's location can be spread out over a number of address spaces. There are two main ways of distributing an object: by fragmention or by replicating.

When an object is fragmented the object is split over more than one location. This means that an object's state is partitioned over multiple address spaces.

When an object is replicated its whole state is maintained at multiple locations. Objects are often replicated for efficiency or availability reasons. For example, an object may be replicated so that it does not form a bottleneck. In order for this sort of replication to be possible an object's interface must also be accessible at multiple locations. In our terms we say that the interface is replicated. These two means of distribution can also be combined to produce fragmented replicas and replicated fragments.

Note that the possibility for these must be offered by the system. That is the programmer should not have to implement the fragmentation and replication communication protocols because these should already be offered by the system. Alternatively, the system may offer the possibility for external implementations of these protocols (e.g. third party libraries) to easily be added to objects to make them distributed.

2.5 Interaction with Objects

In distributed object models method invocation on objects is used by active entities to communicate with each other. Therefore, interaction with objects is an important issue.

2.5.1 Method Invocation

From the client's perspective there are two ways that method invocations can be performed: static and dynamic. For static invocation the method invocation is constructed at compile time, while for dynamic invocation it is constructed at run-time. For dynamic invocation the method interface is often not known at compile time and must be discovered before the invocation can be constructed. The presence or absence of *static and dynamic invocation* is the first aspect of the interaction issue. This does not require external/internal separation.

Next comes the *invocation* itself. A method invocation can be seen as a message to an object asking it to invoke a method, or a direct operation invocation on an object or its reference. The external and internal views for this point do not have to correspond at all. For the external view this usually depends on the language used for programming. For example, in POOL [3] messages are sent, while in C++ methods are invoked directly. For the internal view this depends on the operational semantics and implementation aspects of method invocation. This aspect should deal with the techniques, problems and restrictions of the actual invocation mechanism from the external and internal viewpoint.

Dynamic binding is a feature of many object-oriented programming languages; it basically means that the actual method which is to be executed is determined at runtime depending on the context (e.g. the actual object that a reference points to). However, with distributed objects the method invoked is almost always determined dynamically, that is an explicit look up is done at runtime. Nevertheless, any interesting details of a model that have to do with dynamic binding can be described here.

2.5.2 Parameter Passing

When a method is invoked values are often passed as arguments or return values. The external view of parameter passing deals with how these are passed (e.g. by-value, by-reference, by-copy-restore), what kinds of parameters are possible (e.g. in, out, inout) and restrictions as to what values can be passed as parameters.

The internal view deals with the implementation of parameter passing. Usually parameters have to be marshaled and unmarshaled so details should be given about how this is done. This is especially the case when other objects are passed as parameters.

2.5.3 Security

Security is important in any system where possibly untrusted entities can have access to resources and program elements owned by others. In an object model the object boundary offers the ideal place to implement security checks and restrictions because all interactions with objects take place through their interfaces. Many security issues can thus be addressed by looking at and restricting access to an object's interface.

Enforcing encapsulation

The encapsulating of data by objects is a first layer of security which lets a programmer determine how an object's data may be read or modified. This layer of protection is of course only as strong as the encapsulation provided by the system. This aspect then describes how strong the encapsulation is and how it is enforced by the system.

Authorization and Auditing

Encapsulation does not prevent untrusted entities from invoking an object's methods, so another level of security can be offered in the form of *authorization*. This refers to limiting access to an object (i.e. invoking an object's methods) only to authorized principals/clients. Authorization also involves authenticating principals/clients in order to find their true identities. The external view includes aspects of authorization and authentication which can be controlled by the user. The internal view relates to the implementation of authorization and authentication at the system level.

Auditing can be used along with authentication and authorization to keep track of accesses to objects. Here any auditing done by the system or auditing facilities offered by the system should be described.

2.5.4 Atomicity

Atomicity here refers to an action being all or nothing, that is either all of the steps of an action are taken or none are. This is important for objects because they have to insure that their state is always in a consistent state. Here any services offered by the system which allow atomic actions should be described.

2.6 Concurrency

In a distributed application multiple processes will want to simultaneously to access the same data or invoke the same operation. The techniques developed to help solve concurrency problems often require much work or loss of flexibility from a programmer. Object methods and the encapsulation provided by the object model provide a good opportunity to make concurrency easier for the programmer.

2.6.1 Concurrent access to objects

This aspect refers to the possibility of more than one client invoking an object's method(s) at the same time. If concurrent access to an object is not allowed then there is no need to worry about serializing method invocations. No external/internal division is necessary for this aspect.

2.6.2 Isolated Actions (Serializing Concurrent Actions)

When actions are serialized or isolated the result is as if all the actions took place one after another even though they might have all taken place at once. Two types of serializability are possible in distributed objects models - serializability of actions within an object, that is actions which affect only that object's state, and serializability of actions between objects, that is actions which may interact with other objects and therefore affect the state of multiple objects.

2.6.3 Synchronization

Synchronization in distributed programming refers to allowing concurrent processes to synchronize their actions. There are two aspects to synchronization: condition synchronization - waiting with an action until a certain condition has been met, and mutual exclusion synchronization - preventing processes from simultaneously accessing shared data. Here we deal with synchronization mechanisms offered by the system to the programmers (e.g. semaphores, monitors, etc.).

Synchronization mechanisms can also be split into two categories in distributed objects. Those that work within objects - that is it affects only the one object's state, and those that work between objects - for actions that affect the states of multiple objects.

As an example of the difference between (synchronization or serialization) mechanisms that work within objects and ones that work between objects we can look at monitors and distributed locks. A monitor controls the execution of a method within one object, its purpose is to protect the state of just that object. Distributed locks can affect multiple objects and may be used by entities that invoke operations on these objects.

2.7 Object-Oriented Programming Issues

Other than the concept of objects and encapsulation object-oriented programming is usually associated with a number of other features. Often models which do not at least include inheritance are called object-based rather than object-oriented [52].

2.7.1 Inheritance

Inheritance is a mechanism by which existing objects or classes can be extended in order to create new ones. There are two basic types of inheritance: interface inheritance and implementation inheritance. In interface inheritance only the interface of the parent object (or class) is taken over, the programmer must still implement the actual methods. For implementation inheritance the actual implementation of the methods and the state variables are inherited as well. Usually the inherited code can be overridden allowing for customization of the new object or class.

The external view of inheritance includes a description of what kind of inheritance (if any) a model has (i.e. interface or implementation) as well as any restrictions that the programmer notices when using inheritance (e.g. only local objects can be inherited). It also notes whether it is objects or classes that are inherited. The internal view includes any restrictions for the system when dealing with inheritance (e.g. the parent object/class must always be moved with a child object when it is moved) as well as how inheritance is actually implemented.

2.7.2 Multiple and Dynamic Inheritance

Multiple Inheritance. Often multiple objects or classes can be inherited. In this case the new object or class has multiple parents. One important issue which must be addressed when dealing with multiple inheritance is what happens when an object inherits more than one method with the same interface. Also important (for the internal view) is whether multiple inheritance is handled using the same mechanisms as single inheritance.

Dynamic Inheritance. Normally inheritance is static, that is it is done at compile time, however sometimes dynamic inheritance is possible. This means that the object inherits new methods or state variables at runtime from other objects or from its environment.

2.7.3 Polymorphism

Polymorphism is a feature often (but not exclusively) found in object-oriented languages. It usually refers to a number of concepts and has to do with variables and values (including objects) which may have more than one type and functions whose parameters may be of different types.

There are four types of polymorphism [15]. The first two are are grouped under the term universal polymorphism and include parametric and inclusion polymorphism. Parametric polymorphism refers to functions or methods which work uniformly on a range of types. Inclusion polymorphism means that an object may have many different, not necessarily related types. The next two types are grouped under the term ad hoc polymorphism and include overloading and coercion. Overloading relates to functions which have the same name but are further unrelated (i.e. have separate implementations). Coercion refers to parameters of a function or method being unnaturally (i.e. not following the type hierarchy) coerced to appropriate types.

2.8 Typing

A type represents a set of values and a set of operations defined on those values. Types help to organize and classify data according to characteristics and purpose. The typing in a system can vary from very weak typing (basically no typing) to strong typing where all expressions are type consistent. Also the type system must be enforced by checking the types in an expression, this can either be done at compile time (static typing) or during runtime (dynamic typing).

2.8.1 Type and Subtyping

In object-oriented systems the type of an object is often equal to an object's class, though this is not always the case. Also some systems offer primitives or mechanisms which allow the types of objects to be explicitly determined and compared by the user. The definition of type in a model and the possibilities for explicit type determination and checking should be described here (no external/internal division). Often types can be subtyped, creating type hierarchies. The description of subtyping should explain the type hierarchy in the model and how subtypes can be created. There is also no external/internal division.

2.8.2 Type Checking

In order to enforce the typing model some checks must be done to ensure that objects are used appropriately according to their type. As mentioned above, type checking can be done either at compile time by the compiler or at runtime by the runtime system (or both). Often with distributed objects some kind of dynamic typing will be necessary because the type of a remote object may not be known at compile time. The kind of type checking done and the basis of type equality should be explained here.

2.9 Persistence

A persistent object is one that persists until it is explicitly deleted, even if there are no clients referring to it. This means that persistent objects must persist even if, for example, the machine that they are on fails or is turned off, or between program executions. Usually persistent objects reside on secondary storage as well as having a copy resident in memory. Actions on persistent objects modify the version of the object in memory and the version on secondary storage is usually only updated when an action commits. Persistent objects are often used for storing information which must not be deleted unless explicitly requested, for example files, or databases.

A persistent object can be in two states; melted or frozen adopting the terminology used in the Spring model. In the melted state it is present in memory and on secondary storage while when frozen it is present only on secondary storage.

2.9.1 Persistent Objects

Like a regular (non persistent) object a persistent object also has a life cycle. Unlike regular objects a persistent object's life cycle also includes melting and freezing. Note that it may also be possible for regular objects to become persistent and for persistent objects to become regular objects (losing their persistence).

Creation and Deletion. *Creation* deals with the creation of persistent objects. This includes creation from scratch (comparable to the creation of a regular object) or the transformation of a regular object into a persistent object. *Deletion* deals with the deletion of a persistent object. This means deletion of the persistent part of the object (that is the part on secondary storage) - so destruction of just the memory image does not fall under deletion. Like creation this aspect includes the changing of a persistent object to a regular object (which implies the deletion of the persistent state).

Melting and Freezing. *Melting* describes how (and/or when) a frozen persistent object is melted and *freezing* describes how (and/or when) a melted persistent object freezes.

2.9.2 Durable Actions

Durable actions refer to actions whose results are permanent once the action completes. This is usually related to persistent objects as they offer the possibility of making modifications to the state permanent (by making the state persistent). This aspect should describe the mechanisms offered by the system which allow actions to become durable.

2.10 Failure and Fault Tolerance

In distributed applications on distributed architectures the chance that something goes wrong is bigger than that something goes wrong on a traditional (non-distributed) application running on a single machine. This is because there are now more computers that can fail, as well as network connections over which communication can fail. It is important that systems for distributed programming deal with (preferably transparently) possible failures.

2.10.1 Failure Detection and Availability

Failure Detection. There are two types of failures that can occur in a distributed object system; existence failures and transient failures. Existence failures refer to failures that happen before a method invocation has started (e.g. an object cannot be located). Transient failures refer to failures that occur once a method invocation has started (e.g. the machine where the invocation is running on crashes). Failure detection refers to what kind of failures the system can detect and what it does once these failures are detected. The external view relates to what the user notices (and has to do) when the system detects a failure, and the internal view relates to how the system actually detects failures and handles them.

Availability. Availability is a promise made by the system regarding the availability of objects in the face of hardware, software or network failure. The external view should describe what the model promises with respect to availability and any restrictions as to what objects remain available. The internal view should explain how the system fulfills these promises.

2.10.2 Exception Handling

Exception handling deals with how errors are signaled to other objects. These could be system errors or user defined errors.

2.11 Programming Interface

In order to write a program using distributed objects a programming interface is necessary. This allows a programmer to access and work with the objects and services offered by the system. The types of interfaces can vary from a completely new language for using the system to a library which is linked in with a program written in an existing programming language. Other possibilities include modifying an existing programming language to include new constructs on which the model relies. Alternatively, interface definition languages can be used which allow the interfaces of objects to be defined, but which require the actual implementation and rest of the program to be written in an existing language.

2.12 System Issues

This aspects deals with specific features of the actual system. This includes resource management, extra services offered by the system that haven't been described above and any special internal structure of the system. Also we deal with interoperability and scalability.

Interoperability. Interoperability refers to how systems work together. It has two aspects: interoperability between different models, and interoperability between different implementations of the same model.

Scalability. Scalability is concerned with to what extent a system/model can expand. That is whether the model or system has any inherent limitations which prevent it from growing (with respect to number of machines, number of objects, number of users of objects, etc.). Scalable systems should not depend on any centralized resources or on algorithms that need global information.

2.13 Overview of the Framework

The following is an summary of our framework.

1. Objects

Chapter 3

Descriptions of Selected Models

3.1 Introduction

This chapter will present the descriptions of a number of models for distributed objects. The models will be presented per major issue; for each issue first a table will be presented to give a quick overview of the features present and then each model's features will be presented in more detail. The models described are Spring, CORBA, Emerald, Obliq, Fragmented Objects, Globe and Legion.

The models were chosen in order to thoroughly 'test' the framework, thus they include many different approaches to distributed objects. Also where possible similar models were chosen so that the comparative qualities of the framework could be tested. Spring and CORBA are similar models which use a proxy/server structure. Emerald is a language based model and Obliq is a scripting language based model built on top of another distributed object model. Fragmented Objects and Globe are both models based on object fragmentation and replication. Legion is an 'in-between' model that is not language dependent, is not based on the proxy/server structure and is not based on fragmented objects.

3.1.1 Spring

Spring [33, 39, 19] is a distributed object-oriented operating system and programming environment. All Spring resources and system functions are modeled as objects. Spring also provides subcontracts [20] which allow objects to use different kinds of runtime techniques for operations like method invocation, parameter marshalling, etc.

The description of Spring will focus primarily on the object model and subcontracts.

3.1.2 CORBA

CORBA (Common Object Request Broker Architecture) [36, 37] is a standard for distributed objects being developed by a consortium (called the Object Management Group - OMG) of software vendors and end users. CORBA specifies an Object Request Broker (ORB) which provides the mechanisms by which objects transparently make requests and receive responses. This specification, however, only defines the interface and to a certain extent the mechanics of an ORB, but leaves implementation details open. Along with the ORB and object model CORBA also specifies a number of services which expand and/or complement the ORB's features.

The description of the CORBA model will describe the object model, the runtime system (ORB) and the services as specified by the OMG.

3.1.3 Emerald

Emerald [42, 7] is a programming language and programming system providing both process and object mobility [25] on a network of workstations. Objects are the units of programming and distribution [8] , and the entities between which communication takes place. Operations can be invoked on non-local (in the network) objects, and objects can move from node to node.

In the description of Emerald we will concentrate on Emerald's runtime system rather than on language aspects.

3.1.4 Obliq

Obliq [10, 11] is a lexically-scoped untyped interpreted language that supports distributed object-oriented computation. Obliq objects have state and are local to a site. Computations in Obliq can roam over the network, while maintaining network connections.

Obliq is implemented using Modula 3 Network Objects [6] however in the description of Obliq we will primarily focus on Obliq's features and mention Network Objects only where necessary.

3.1.5 Fragmented Objects/SOS/FOG

SOS [45] is an operating system built upon the ideas of Fragmented Objects [30, 43]. Fragmented Objects is a model for distributed shared objects. The SOS system is an implementation of the Fragmented Objects model and adds some extra functionality in the form of services. The SOR group has also developed a specific language, FOG [16] , used for structuring distributed applications as fragmented objects.

This description will handle Fragmented Objects, SOS and FOG.

3.1.6 Globe

Globe [22, 51, 21, 23] provides a model where processes communicate and interact through objects which are physically distributed and shared between processors. The implementation of distribution, consistency and replication of state is completely encapsulated in these distributed objects. This makes the objects less dependent on runtime services and allows object specific solutions to problems.

Globe is currently in the design phase and the model's current state will be described. This means that many aspects of it may not be completely or only partially specified.

3.1.7 Legion

Legion [18, 29] provides an architecture for building distributed systems that give the illusion of a single virtual machine. It is designed so that it can be used above existing computer architectures and operating systems and uses standard protocols for networking. Legion is also designed with high performace programming in mind and allows users to make their own scheduling (with respect to placement of objects) decisions thereby allowing them to access remote computing power when needed.

3.2 Objects

	Spring	CORBA	Emerald
granularity	f/m	m	f
kinds of objects	server, proxy	server, proxy	direct, local, global
composition	y	not defined	y
interface	m, e	m, e	m
activity model	p	p	h

	Obliq	FO	Globe	Legion
granularity	m	m	m	m
kinds of objects	global	isolated, distributed	isolated, distributed	local, global
composition	y	y	y	n
interface	n	m	m, e	n
activity model	p	p	p	a

3.2.1 Spring

Granularity

The Spring model has medium grain granularity, however, it is possible to have fine grain Spring objects as well.

Kinds of Objects

For users of the Spring model all objects seem the same. From the internal view there are two kinds of objects: global and local objects; Spring calls these server based and serverless objects. Global objects are those that may be in different address spaces than their clients. Local objects on the other hand always have their state in the client's address space and are usually used for light-weight objects like simple data types.

Structure

For users an object is defined by a strongly typed interface. Internally a global object is usually split up into a server and multiple proxies. The server object contains the actual implementation and resides in a different address space than a client. Proxies are client side representations of the object. They reside in the client's address space and provide a way to access the server. Each object contains subcontract routines. Subcontracts are replaceable modules which are responsible for object management as well as object interaction. By making subcontracts replaceable programmers are given control over object management and interaction policies.

Interface

Spring supports multiple interfaces through multiple inheritance. Because an object which has inherited from a number of other classes can be cast to any of those object's types, its interface can be different depending on how it is being accessed. Spring does not support evolving interfaces.

Composition

In Spring composition is achieved through nesting of objects. This is done either by including a reference to an object in the outer object's state or by including the entire nested object. The subcontract used determines how a nested object reacts to actions of the composite object, e.g. whether the nested object moves along when its outer object is moved.

Activity Model

The activity model in Spring is passive.

3.2.2 CORBA

Granularity

The CORBA model has medium grain granularity.

Kinds of Objects

CORBA users can distinguish between local and global objects. The local objects are plain objects, that is programming language specific objects which are not visible to the CORBA system. The global objects are the only objects that the system knows about. They are objects which may be referenced by clients which reside in different address spaces than the object itself.

Structure

For users, the global objects consist of only an interface. Internally a global object is structured as a server and a number of proxies. The server includes the actual implementation of the objects as well as stubs used by the system for accessing the object. The proxies consist of (language dependent) stub routines which call on the system to access the server.

Interface

Multiple interfaces can be provided by the CORBA objects through use of multiple inheritance. By multiply inheriting the interfaces of several objects a CORBA object can provide all of those objects' interfaces. CORBA also provides evolving interfaces. The runtime system maintains a database of interfaces (of objects managed by it) called the interface repository. This database can be dynamically modified by adding or removing an interface's methods. Clients using dynamic invocation will be able to notice these changes and take advantage of the new interface.

Composition

Whether CORBA objects can be nested by reference is not defined. A Relationship service will allow relationships between independent objects to be created dynamically. A number of different types of relationships will be possible including ownership, reference, containment, etc.

Activity Model

The activity model in CORBA is passive.

3.2.3 Emerald

Granularity

The Emerald model has fine grain granularity, that is all entities in a program are Emerald objects.

Kinds of Objects

From the external view all objects in Emerald are the same. From the internal view there are three different kinds of objects corresponding to their implementation style, these are global, local and direct. The style of an object is determined by the compiler depending on how that object will be used. Global objects are those which can be moved and which can be accessed remotely, that is they may reside in different address spaces than their clients. Local objects are local to another object, that is they can only be accessed by methods of that object and are always located in the same address space as that object. Direct objects have the same characteristics as local objects but their data is allocated directly in the state of the enclosing object.

Object Structure

Users see objects as having a name, a state, methods and an optional process. Internally an object consists of a data area, a concrete type (see Class Objects) and optionally a process. The data area includes a pointer to the concrete type and the state data.

Interface

In Emerald variables are typed. This means that any objects accessed by a variable will be seen to have the same type (and thus interface) as the variable. Thus an object will present different interfaces depending on the type of the variable used to access it. Emerald does not support evolving interfaces.

Composition

Users can create composite objects by including an object reference in an object's state. Internally depending on how the nested object is implemented either the entire object or a reference to it is included in the state of the composite object.

Emerald also allows objects to be attached to each other using the attach primitive. An attached object always moves with the object that it is attached to.

Activity Model

Emerald has a hybrid activity model. All processes are bound to an object, however, not all objects have to include a process. Processes communicate through shared objects as in the passive model rather than directly as in the active model. When a process executes a method of another object the process conceptually executes inside that object. Thus if an object moves, part of the process moves as well.

3.2.4 Obliq

Granularity

The Obliq model has medium grain granularity.

Kinds of Objects

All Obliq objects are global objects, that is they can be accessed by clients in different address spaces.

Object Structure

An Obliq object consists of a collection of named fields. There are three types of fields: method, alias and value. A method field holds a method, an alias field holds an alias which redirects method invocations to other methods and a value field is a field containing anything else. An object's interface consists of all its fields.

3.2.5 Interface

Obliq does not support multiple or evolving interfaces.

Composition

An object can be nested in another object by assigning it to one of the fields (thereby including the whole object) or by assigning a reference to the object in one of the fields.

Activity Model

The activity model in Obliq is passive.

3.2.6 Fragmented Objects

Granularity

The Fragmented Objects model has medium grain granularity.

Kinds of Objects

In SOS there are three kinds of objects: plain objects, isolated objects and distributed objects. Plain objects are ones that are not seen by SOS (e.g. C++ objects), isolated objects (called elementary objects by SOS) are objects which at any one time reside and are accessible in a single address space, and distributed objects (called fragmented objects by SOS) reside (and are accessible) in multiple address spaces. Distributed objects consist of a group of communicating isolated objects. These will often be referred to as fragments. Internally there are isolated objects and primitive connective objects. A primitive connective object is a system object that is distributed over multiple address spaces and provides the means of communication between the fragments of a distributed object.

Structure

Isolated objects consist of state and a public and private interface. A distributed object appears to clients as a single shared object. For programmers a basic distributed object is composed of: a set of isolated objects a client interface which is presented to each client through the public interface of a local fragment, a group interface which is used for communication between the fragments, and connective objects. An object's method implementations are provided by a class object.

In SOS a distributed object is composed of three kinds of isolated objects: a server object which serves the request, fragments which represent the service locally and a provider which provides fragments on client requests. The provider object receives requests from the distributed object's potential clients and arranges that new or existing fragments are created in their address spaces.

Interface

Distributed objects allow multiple interfaces because the programmer of such an object may decide how the interface is presented to a client. This is done by choosing the isolated objects through which a client will access a distributed obect. The compiler and runtime system verify that a client gets the interface that it expected.

Composition

Composite objects can be created in SOS by including references to other objects in an isolated object's state. Both isolated and connective objects can be nested in this way. It is also possible to include memory pointers in an object's state, however there is no support for pointer management with respect to objects moving or being moved. When isolated objects are nested using references they will migrate along with their outer object and be written to secondary storage when that object checkpoints.

Activity Model

The activity model in SOS is passive.

3.2.7 Globe

Granularity

The Globe model has medium grain granularity.

Kinds of Objects

There are two kinds of objects (from both the external and internal views) in Globe: isolated (called local) and distributed. Isolated objects are placed entirely in one address space and can be either primitive or composite [1]. Primitive objects do not include any other objects, while composite objects can include other local objects. Distributed objects reside in multiple address spaces.

Object Structure

Externally an object consists of state, a collection of methods and a collection of interfaces (an interface denotes a subset of the methods of an object). Each object includes a number of standard methods for initialization, cleanup and identification. Each object also has a standard object interface which denotes the standard methods. From an internal view a primitive isolated object consists of a state instance, which is an instance of a data structure which can hold the state, method instances, which include a method implementation per method (as described in the class) and interface instances, which are tables consisting of a method address and state pointer pair for each method in the interface.

Method implementations may actually reside in a class object rather than in the object itself. For a description of composite objects see Composition.

Distributed objects are a collection of communicating isolated objects which will often be refered to as fragments. A fragment is a composite object with a standard organization. The object consists of a semantics object, a communication object, a replication object and a control object. The semantics object is implemented by the developer of an object, it contains the actual functionality of the distributed object. The communication object is responsible for communicating with other objects and the replication object is responsible for keeping the state of the distributed object consistent. Both communication and replication objects can be chosen from a library or implemented by the programmer. The control object is generated automatically using interface information from the semantics object. It is responsible for handling the interaction between the semantics and the replication object.

Interface

An object in Globe can have multiple interfaces. Each object can contain multiple interface tables and an object can choose which ones it will export to a client. The interfaces exported to a client can be changed at runtime so evolving interfaces are also available.

Composition

Isolated objects can be nested within other isolated objects resulting in composite objects. Composite objects can also be nested. Internally the state of a composite object is a combination of all the states of the nested objects plus any other state belonging to the object itself. The method instances are the union of all the included objects' methods. The interfaces are defined by the composite object itself.

Activity Model

The activity model in Globe is passive.

3.2.8 Legion

Granularity

The Legion model has medium grain granularity.

Kinds of Objects

In Legion users are aware of local and global objects. The local objects are plain objects, that is language dependent objects that are not visible to the system. The global objects may reside in different address spaces than their clients. Global objects must ultimately inherit from a system provided base object class and are address space disjoint. Internally there are two types of global objects: stateful and stateless. Stateful objects are the most common and maintain their state from instantiation to destruction, while stateless objects do not maintain any state between invocations and their methods can be considered to be pure functions.

Structure

Users see Legion objects as consisting of state and an interface through which the state is manipulated. The state consists of contained local objects. An object's interface is described by the complete set of method signatures (typed formal parameter list and name) defined for it. Internally an object exists in an address space, has a class, a name and a set of capabilities. An object also contains a binding cache and the network address of a binding object. The cache is used to quickly map object identifiers to network addresses and a binding object is used to do this mapping if the information cannot be found in the cache.

Interface

Legion objects do not have multiple interfaces. The dynamic inheritance (in the form of InheritsFrom()) allow methods to be dynamically added to an objects interface, thus providing evolving interfaces.

Composition

Legion does not offer support for nesting of other Legion objects.

Activity Model

The activity model in Legion is active.

3.3 Object Management

	Spring	CORBA	Emerald
class	n	y	n
class objects	n	n	n
reference	native	languge mapping	reference

	Obliq	FO	Globe	Legion
class	n	y	y	y
class objects	n	y	y	y
reference	variable	object	local object	language mapping

3.3.1 Spring

Object Definition

The object interface is defined in an interface definition language (IDL) which is then compiled to produce a language specific interface. The actual implementation of the object is written around this generated interface and compiled into an application. The subcontract code is dynamically linked in when the application is started.

Object Creation

Spring server objects are usually started like normal applications. The proxies are not explicitly created by the clients but are acquired as results of operations (e.g. name binding). In the internal view proxies are created by the server objects. The server's subcontract creates a communication endpoint and constructs an object (the proxy) which uses this to communicate with the server. Once the proxy object is created it is moved to the client's address space by the server's and the proxy's subcontract code.

Object Destruction

To delete proxies the language specific method of deleting an object is used. Internally this causes the client stub (see Invocation) to to call the consume function in the subcontract belonging to the proxy. The result of calling this function is that the proxy is destroyed and any communication resources are cleaned up. When a server is no longer accessible by any proxies then it is destroyed. When a server object dies all communication endpoints that it uses become invalid.

Class and Class Objects

There are no class objects in Spring.

Object Naming

In Spring any object can be bound to any name in any name space [41]. A name space consists of a number of connected directories (called contexts) which form a directed graph. The directories themselves are objects and can be manipulated directly, e.g. passed as arguments. Directories may be passed to processes by their parents and by default each process is passed at least one directory by its parent. Binding an object associates it with a name in a directory. An object may be bound to several different names in possibly different directories or to no names at all. The name service defines a structural representation for names including identifier, version number, kind, etc. Presentation and parsing is done by the user software. Resolving a name returns a reference to the object bound to that name.

Internally a name space is created by binding directories within other directories. A single object may implement multiple directories, however, clients are unaware of which object contains which parts of the name space. When an object is bound to a name then the object is actually moved to the appropriate directory's address space and stored there. Thus the objects that are usually bound to names are those that can provide proxies when they are contacted during name resolution.

Name Resolution involves invoking a resolution method on a directory object. If the directory object does not include the object the request is forwarded on to another directory as specified in the name. When a directory is found which directly references the object then the object is contacted and it can return a copy of itself, create new objects, return a proxy to itself, etc.

Requests forwarded between directories implemented by different objects may be subject to security checks if the objects do not trust each other.

Object Reference

Clients treat Spring objects as native objects (or language constructs for non object-oriented languages). Internally the native object/construct referenced by the client is actually a generated stub that transfers calls to appropriate methods in the proxy. When Spring object references are passed as arguments to procedures or methods then the whole proxy (or a copy depending on the parameter passing semantics) is passed, rather than a reference.

3.3.2 CORBA

Object Definition

An object's interface is defined in an IDL and its implementation in a normal programming language. One way that this can be done is as follows: an IDL precompiler produces language skeletons for the actual implementation of the object, the object implementation is added to the skeletons and the code is compiled. This whole process generates interfaces for the interface repository, proxy code, server stubs and server code.

Object Creation

Normally a server object will be created by explicitly 'installing' it, e.g. running a compiled version of it. On the client side a reference to this object could be acquired by instantiating a language dependent proxy object.

Objects can also be created at runtime through the life cycle service. The life cycle service provides operations for creating, copying, moving and destroying objects. It can also handle associations between groups of related objects (containment and reference relationships) and allow varying semantics for its operations - e.g. deep and shallow operations.

Object Destruction

Server objects can be destroyed using the life cycle service. Proxies can be destroyed using native language techniques.

Class and Class Objects

There are no class objects in CORBA.

Object Naming

Object naming is provided by the naming service which binds names to object references. An object's name is unique within a name space (called a name context) which is a graph of interconnected directories. An object's name consists of a sequence of names and forms a hierarchical naming tree. Each name component has two attributes; an identifier - the name string, and kind - a descriptive attribute.

The name service is implemented by two kinds of objects: a directory object and an iterator object. The directory objects provide methods for binding objects and resolving names as well as keep track of bindings for a directory. The iterator objects are used to iterate through directory trees.

Object Reference

Clients have references to objects and invoke operations through these references. Because of language mapping clients can reference CORBA objects in the same way that they would reference native objects or data. The language mapping from any particular programming language must be the same for all implementations of CORBA.

From the internal view the client's reference will often be a reference to a proxy which controls the contact with the server object. Interoperable Object References must be used to pass object references between different implementations.

Interoperable Object References contain a collection of profiles which describe how to contact the object using a particular implementation's mechanism.

3.3.3 Emerald

Object Definition

Objects are defined using an object constructor primitive. The object constructor includes state and method declarations.

Object Creation

In Emerald object creation and object definition are not separate events. When an object is defined then an instance of the object is created and a reference to the object is returned. During object creation an optional initialization method is called which initializes the object's state. After this the process (if any) belonging to the object is started.

Internally when an object is created a data area is allocated for it and, if it does not already exist on that machine, a concrete type (see Class Object) is created.

Object Destruction

Objects are garbage collected when there are no more references to them so the user does not have to worry about destroying objects. There are two types of garbage collectors, a local (for local garbage) and a distributed one (in which the nodes cooperate to collect distributed garbage). Both use a modified mark and sweep algorithm [1]

Class and Class Objects

There is no class concept in Emerald, therefore Emerald does not support explicit class objects. However at the system level concrete types resemble class

[1] A mark and sweep algorithm works as follows: all objects are originally white, then all known reachable ones are marked gray, the references in all gray object are then also marked gray and the objects are marked black. This is repeated until there are no more gray objects, then all the white objects can be deallocated. In Emerald every node holds an object descriptor for each remote object that has been referenced since the last garbage collection. The local garbage collector works as follows: each object descriptor has a flag RefGivenOut which is set if a reference to the object is ever given to another node, or if the object has been moved to its current node. Resident objects with the flag set are considered reachable and non-resident objects are ignored.

The distributed garbage collector works as follows: a global collecting process started on each node, all global objects are marked white, then all explicitly reachable ones are marked gray and scanned (if attempting to scan a remote object, a mark-gray message is sent to the node, and an is-black message is returned when the object has been checked), all objects that move are scanned and marked black. The collection is finished once all nodes have done their scanning and there are no more gray objects.

objects. A concrete type is a kernel structure (not an object) which contains the code and templates for all objects created using the same object constructor. A template describes the data layout of an object's state as well as the layout of activation records for the object's methods. A concrete type must be present on the same node as an object, however they are not automatically moved with an object but only if requested by the destination kernel. This happens if the appropriate concrete type is not already present on that node.

Object Naming

Emerald does not provide a name service.

Object Reference

Users access objects through variables which hold object references. A reference can be acquired either through language primitives such as object constructors or as the result or parameter of an operation. Internally a variable contains a vector and an object reference. The vector consists of pointers to the method implementations in the concrete type of the object being referenced. The structure of the object reference is different for each implementation type. For local and direct objects this is basically a memory pointer to the object's data area, while for global objects this is a pointer to a local structure which tells where the actual object can be found.

3.3.4 Obliq

Object Definition

An object is defined by specifying all of its fields and their initial values.

Object Creation

Objects are created when they are defined or when they are cloned (see Interaction) from other objects.

Internally creation of an Obliq object causes a Network Object to be created.

Object Destruction

Objects are garbage collected when they are no longer referenced[2].

[2] Internally the Network Objects runtime system takes care of the garbage collection. It works as follows: for each network object the runtime records in the dirty set the clients which contain a proxy for the object. As long as the set is not empty the runtime keeps a reference to the object so that it will not get garbage collected. Once the set is empty the runtime removes the reference and the object can be cleaned up. Clients add or remove themselves to or from the dirty set when surrogates are created or destroyed.

Class and Class Objects

There is no class concept nor are there class objects in Obliq.

Object Naming

Objects can be bound to names via the name server. Name binding and resolution occurs using a built-in net module which has functions for binding objects to names and resolving names to object references. The name service is implemented by a name server which is an external process uniquely identified by an IP address. This server maintains a table associating text strings with object references.

Object Reference

Clients reference objects through variables. Internally these variables contain a pointer to the object's data area if it is in the same address space as the variable, or a network reference[3] for remote objects.

3.3.5 Fragmented Objects

Object Definition

In FOG an isolated object definition looks like:

```
class elobj1 :  parentof1 {
        private:
                // private state and methods
        public:
                // public methods
        group:
                // group methods
        export:
                // provider methods
};
```

and a distributed object definition looks like:

```
group foName { elobj1, elobj2 };
```

[3]A network reference is a Network Object reference which is a pointer to a local proxy object.

Isolated objects are defined as a normal class, the 'group:' keyword specifies which methods are for the group interface and the 'export:' keyword specifies the provider method, that is which method provides new fragments when clients request one. All isolated objects must (possibly indirectly) inherit from a root ancestor object which defines a minimal object interface. Distributed objects are defined as a group of isolated objects. Object definition leads to the creation of a class object.

Object Creation

Object creation is managed by an object management service (called the Acquaintance Service in SOS) and is done in two steps. The first step involves actually creating a memory image of the object; the second step requires giving the object a reference to its class object. Between the two steps initialization methods (belonging to the object and its super objects) are called to initialize the object's state.

For distributed objects an isolated object must first be created and then registered in a new group. Isolated objects can also be added to existing groups.

Object Destruction

Isolated objects are destroyed by calling a destroy method. Before it is actually destroyed any destructors defined by it or its superclasses are invoked. Distributed objects disappear when their last fragment is destroyed.

Class and Class Objects

The class object is an instance of a predefined class of isolated objects and holds the compiled code for a class. A class object is a normal object. An object's class obect must always be present in that object's address space. Class objects may be shared by objects of the same class in the same address space. Note that distributed objects do not have a class or class objects, they are defined by their fragments.

Object Naming

Objects may have names that are unique to their name space. A name space is a distributed object whose fragments allow clients to maintain personal views of the name space. That is the clients may map other name spaces in their own name space thus creating a personal naming hierarchy. Clients bind or resolve names at these fragments. When an object is bound then the name space fragment adds a reference to the object along with its name into an internal table. When a name is resolved then the client is given a reference to a distributed object. The client is responsible for requesting a fragment from the object.

Object Reference

Objects are accessed through variables which hold object references. There are two types of references, global and local. An object manager[4] provides ways to convert between these two. Internally each object has a unique identifier, and an object reference is either a pointer to the objects data area or it consists of the objects identifier and network location information.

The SOS system also keeps track of all isolated objects in each address space. A table present in each address space contains information about the objects' identifiers, which distributed objects they are part of, where their class objects and data areas are, etc.

3.3.6 Globe

Object Definition

An object definition consists of a class definition and a class implementation. The class definition is a collection of interface definitions. An interface definition describes the syntax and semantics of a collection of methods and a class implementation is a description (and implementation) of the methods and state of an object. Definition of a class object leads to the creation of a class object.

Object Creation

To create an object a class object create method must be called causing an instance of that class to be created. During the creation of an object an initialization method is called to initialize the state. For distributed objects first an isolated object is created and its name registered, the object then grows as more clients connect to it and more fragments are created. Internally object creation causes memory for the objects state, methods (if they are not contained in the class object) and interface tables to be allocated.

Object Destruction

When an isolated object is to be destroyed the destroy method of its class object must be called with the object to be destroyed as a parameter. This causes a destructor to be called and then the object's memory to be freed.

[4] An object manager is a service provided by the system for managing objects (e.g. taking care of creation, location management, etc.)

Class and Class Objects

Class objects are objects that represent class implementations. They are isolated objects and have a class. Class objects include two additional standard methods: a method to create a new object of the class and a method to destroy an object of the class. A class object must exist in each address space where an object of its class is found.

Object Naming

When an object is created it has to bind itself to a name using the name service. Globe provides one global name space, thus a name refers to at most one object at any given time. The name space is constructed as a graph of directory objects. A name consists of a starting point and components which refer to directory objects and eventually the final object. In Globe name binding results in the object being registered with the location service (see Finding Objects) and the name and internal object identifier (called an object handle) being stored in a directory object. Name resolution includes a name lookup, a location lookup and finally the creation of a fragment of a distributed object. The name lookup produces the object identifier which is used in the location lookup step.

Object Reference

Distributed objects are either referenced by name or through a pointer to the interface of a fragment or isolated object.

3.3.7 Legion

Object Definition

A class interface is defined in an IDL and the actual code is defined in any supported language using generated headers. A class may have multiple implementations possibly written in different languages. The definition of a class leads to the creation of a class object.

Object Creation

Objects can be created by calling a class object's create or derive methods. Create creates a new object, while derive creates a new class object. Actually object creation is initiated by native language features (e.g. C++ new, or C++ inheritance) and the compiler generates the code to call the create or derive methods. Both an object manager and host object (which represents the machine on which the object will be created) are involved in creation of objects.

Object Destruction

Object destruction is also initiated by native language features, and ultimately occurs by calling the delete method of the object manager. Note that calls to the object manager are requests and may be rejected.

Class and Class Objects

Class objects are objects which define an object type. All class objects ultimately inherit from a system provided base class object. This base class object defines and implements all of a class object's mandatory interface. A class object contains among other things (possibly multiple) source and implementations of its object's methods and information needed for locating instances of the class. The class object is also responsible for assigning identifiers to new objects upon object/sub-class creation.

Object Naming

Legion provides a single name space for all legion objects. Symbolic names are mapped to object references by name server objects. Internally the naming service is a two layer scheme where symbolic names are mapped to object identifiers by a name server object, and the identifiers are mapped to address bindings by binding objects and class objects. Legion allows multiple name servers and users can define their own name servers. One well-known name server class which provides basic name services and is used as a bootstrap server, is provided by the Legion system.

Object Reference

Legion objects are referenced transparently through language mappings. Internally objects are referenced by object identifiers which are globally unique bit strings. Object identifiers are used to find objects' network addresses

3.4 Object Location

	Spring	CORBA	Emerald
location transparency	y, y	y, y	optional
finding objects	n	n	y
moving	y	y	y
copying	y	y	n
distribution	f, r, fr, rf	f, r	r

	Obliq	FO	Globe	Legion
location transparency	n, y	n, y	y, y	y, y
finding objects	n	y	n	n
moving	y	y	not specified	n/y
copying	y	y	not specified	n/y
distribution	n	f, r, fr, rf	f, r, fr, rf	r(+-)

3.4.1 Spring

Location Transparency

Clients cannot tell whether an object is remote or not. Nor are they aware of the location of an object.

Location

Clients cannot directly refer to an object's location. The internal representation of an object's location depends on the subcontract used.

Finding Objects

Clients are not able to determine the location of an object. Subcontracts are responsible for finding the appropriate server object upon method invocation.

Moving

Spring allows only proxy objects to be moved, server objects always stay at the same location. Objects are moved when passed as a parameter to other object's methods. Internally when objects are moved they are marshaled by the subcontract routines and their local state is deleted. At the destination the subcontract's unmarshal routine rebuilds a new object using the marshaled version of the other object. The subcontract needed for unmarshaling can be determined from the the marshaled state of the object.

Copying

Objects can be explicitly copied or passed as by-copy parameter. Internally when a proxy is explicitly copied, the object stub calls a subcontract copy routine which will create a new Spring object based on the original. The copy semantics (shallow, deep, etc.) actually used depend on the subcontract. When an object is passed as a by-copy parameter then the subcontract performs an optimized combination of a copy followed by a marshal on the copy.

Distribution

Because subcontracts are used to implement different access policies, they can also be used to implement fragmented and replicated objects. Fragmented replicas and replicated fragments could also be implemented in this way. However, the Spring object model is based on a server/proxies structure which can lead to awkward implementations of fragmented and replicated objects. This is because all of the replication and distribution control is in the proxies rather than in the server objects. This can cause problems when for example one proxy decides to create a replica, but others are not aware of this - some method invocations will then be performed on both servers while others only on one (depending on which proxy was used).

3.4.2 CORBA

Location Transparency

Clients cannot tell whether an object is remote or not. They are also not aware of the clients location.

Location

Clients have no way of directly dealing with an object's location. The runtime system has an implementation dependent representation of object references and location and thus the location representation is also left up to it.

Finding Objects

The runtime system is for finding an object and the way this is done is implementation specific. However the implementation repository (where implementation information is stored per domain) stores information about the classes implemented at a domain and is often used to find objects. Also there is a standard communication protocol between the different domains (called the GIOP) which offers mechanisms to communicate with and find object implementations across domain boundaries.

Moving and Copying

The life cycle service allows objects to be moved and copied to different servers. Objects which will need to use the life cycle services need to inherit (both interface and implementation) from a life cycle object. Copying across machines uses a factory object[5] at the target location which knows how to create an instance of that object. The factory objects can be found using special factory

[5] A factory object creates instances of other objects

finder objects. The move operation restricts object movement to within the scope of an appropriate factory finder object.

Distribution

The Collection service will allow groups of objects to be created. Clients will then be able to operate on the objects as a group rather than on the individual objects themselves.

3.4.3 Emerald

Location Transparency

Both an object's remoteness and an its location are unknown to a client unless this information is explicitly asked for.

Location

Location is specified by a reference to a node object, which is an abstraction of a host, or in some cases a reference to any object on that host.

Finding Objects

The location of an object can be explicitly found by a user through a locate primitive. Calling this primitive for any object will return a reference to the node where the object is located. Internally each node contains a table with location information for all objects either resident on or accessed from that node. For an object on the same node as its client this location information refers directly to the object. For remote objects this information consists of a forwarding address. In order to find remote objects the forwarding addresses are followed until the object is found. If the chain of forwarding addresses is broken for some reason (e.g. a node is unreachable) then a broadcast protocol is followed. During this broadcast protocol messages are sent to all nodes in the system, and the one containing the actual object should respond.

Moving

Global objects can be moved using the move or refix primitives. It is also possible to fix an object at a node so that it cannot be moved unless explicitly unfixed. The move primitive is a hint to the system to move an object to another location, because this is a hint the system is not obliged to actually move the object. If an object is moved then all processes executing in it at the time will continue to execute at the new location after the move. The refix primitive is not a hint and will always be performed. Its effect is to unfix a fixed object, move it to the destination location and then fix it there all in one atomic action.

Internally moving an object requires moving its state, and any processes executing in the object. When moving the state a message consisting of the data area followed by translation information is created and sent to the destination. The translation information is used at the destination to find and change all direct memory references. At the destination memory is allocated for the data area(s), the data is copied and the memory pointers are changed according to the translation information.

When an object moves, the parts of the processes currently executing within it must move as well. In order to do so the relevant contexts of the processes executing in the object must be stored in, and sent along with, the message. At the destination the process contexts are set up and the processes reactivated.

Copying

Objects cannot be copied.

Distribution

Objects can be replicated by using Emerald's checkpoint at ¡nodelist¿ and confirm checkpoint primitives. Checkpoint at ¡nodelist¿ specifies the nodes at which the object should be replicated and confirm checkpoint replicates the object at those nodes. Note that this is not an active replication, the replica's methods cannot be executed and they are updated only when confirm checkpoint is called. Internally confirm checkpoint causes copies of the object to be sent to all the nodes in ¡nodelist¿. The object is marshaled in the same way as when moving objects.

3.4.4 Obliq

Location Transparency

Users do not need to know whether an object or variable is remote or not, nor do they know the object's locations. In Obliq data and computations are location transparent, that is their meaning does not depend on allocation sites or execution sites.

Location

Users do not deal with location explicitly. Internally an object's location is not directly dealt with either, instead network references are used to reference remote objects.

Finding Objects

An object's or variable's network location cannot be explicitly determined.

Moving

Objects are local to a site and are never automatically moved over the network. However network references can be transferred and atomic object migration can be coded by using the cloning and aliasing primitives combined with remote execution engines (see Invocation).

Copying

Objects can be copied using the copy primitive which produces local copies of Obliq values. If performed remotely a remote copy will be made. Cloning can also be used to copy an object.

Distribution

Obliq does not support fragmentation or replication of objects.

3.4.5 Fragmented Objects

Location Transparency

The user of a distributed object cannot tell whether it is a distributed object or not, therefore the fact that an object is 'remote'[6] as well as its location are transparent.

Location

Users do not deal with object locations directly. Internally location can only be specified using a reference which contains the exact location of an object. Such a reference can be acquired as the result of the object manager's find operation (see Finding Objects).

Finding Objects

Users are able to explicitly find an object's location. They do this by calling the object manager's find method with a reference to the object as an argument. If the argument was a reference to a distributed object then a reference to the closest fragment is returned. Otherwise if the reference is to an isolated

[6]here 'remote object' refers to distributed object

object then a reference containing exact location information for that object is returned. Looking at this process internally the object manager checks whether it or any of its fragments knows the object's exact location and if so returns a reference containing that information.

Moving

Clients cannot explicitly move objects except by causing a fragment to be moved into their own address space. Internally objects are moved by calling an export routine on a connective object. This method migrates an object (with copy or move semantics depending on the call) along a connection provided by a connective object. During migration a copy of the object is made, the copy is sent to the destination and the original object is deleted. If the object depends on any other objects (called prerequisites in SOS), such as its class object, then these objects are moved or copied as well.

Copying

Copying is basically the same as moving except that the original object is not deleted after being copied to the destination.

Distribution

Fragmented Objects supports fragmentation as the fragments can also hold part of the state depending on how the provider object is implemented. Replication is possible, by adding an appropriate connective object to the distributed object. Fragmented replicas and replicated fragments would also be possible given appropriate connective objects.

3.4.6 Globe

Location Transparency

Once a reference to an object is acquired a user cannot tell whether it is a distributed or an isolated object. All objects are accessed locally so that users cannot determine the location of an object either.

Location

Users do not deal with the location of objects directly. Internally an object's location can be specified using its contact address. This is the address where initial communication with the object can take place (called a contact point in Globe)

Finding Objects

Users are also unable to determine the location of an object directly. The location service [50, 49] can therefore only be seen from the internal view. It is responsible for finding the contact address of an object given the object's identifier. The location service is composed of a hierarchy of directory objects. For every object there is a path from the root of the tree to a leaf node which contains the contact address for that object. The location service decomposes the whole network into regions and each leaf node resides in one of these regions. To find a contact address a search is started at a leaf node, and continues up, until a directory where the object is known is found. From this point the forwarding pointers are followed to the leaf where the contact address can be found.

Moving and Copying

Whether objects can be moved or copied has not yet been specified.

Distribution

In Globe distributed objects are always fragmented. Replication is possible by using appropriate replication objects in a distributed object's isolated objects. Fragmented replicas and replicated fragments are also possible using the appropriate (self written or provided by libraries) replication objects.

3.4.7 Legion

Location Transparency

Machine boundaries are invisible to users therefore users are not aware of an object's location or remoteness.

Location

Users do not deal with object locations. Internally an object location is described by a structure containing the objects identifier and a list of object addresses where it can be reached.

Finding Objects

The object location process is completely hidden from users. It involves mapping an object identifier to addresses where this object can be found. The process is carried out by the client, special binding objects, the object managers and class objects. Internally the binding objects query each other trying to find the object's address. If the binding objects fail then the object's class object must be found. An object's class object must always know the locations of all its instances.

Finding a class object involves the base class object which contains information about how to find all class objects. This information is collected when new class objects are contact the base class object during creation in order to get a unique identifier.

Moving

Migration decisions are made by special user supplied objects, users cannot explicitly move objects. When an object is to be moved its object manager's move method is called. The results of a move are equivalent to executing a Copy and then a Delete

Copying

As with moving, copying decisions are not explicitly made by the user. The object manager is responsible for copying an object. When an object is copied it is first frozen creating a persistent representation of the object. This persistent representation is then sent to the object manager of the destination domain where it is melted to form a new object.

Distribution

Replicated objects are those whose address structures contain multiple physical addresses. This replication is however on a system level and cannot be controlled by users or programmers. Legion does not support fragmenting of objects.

3.5 Interaction with Objects

	Spring	CORBA	Emerald
static/dynamic	s	s, d	s
invocation	direct	language dependant	direct
dynamic binding	n	y	y
parameters	move, copy	copy, reference	ref, move, visit
security			
encapsulation	y	y	y
authorization	y	y	n
auditing	n	y	n
atomicity	n	y	n

	Obliq	FO	Globe	Legion
static/dynamic	s	s	s	s, d(+-)
invocation	direct	direct	direct	language dependant
dynamic binding	y	y	y	y
parameters	reference, copy	reference, copy	reference	value in, out, inout
security encapsulation authorization auditing	 y n n	 y/n y/n n	 y n n	 y y n
atomicity	n	n	n	y(+-)

3.5.1 Spring

Static and Dynamic Invocation

Spring supports static invocation.

Invocation

Users directly invoke methods on object references according to the programming language used. The caller blocks until the operation returns. From the internal view invoking a method on a local object is different than invoking one on a global object. For a local object the client calls a stub which directly calls the appropriated method of the object. For a global object clients invoke methods on a proxy. The proxy then invokes appropriate subcontract operations to marshal the parameters and remotely invoke a method on the server. At the server subcontract code will unmarshal the parameters and call the appropriate server method. Results are returned in the same way, first through the subcontract and then up through the proxy and to the client.

Dynamic Binding

Depending on the mechanisms used by subcontracts for passing method invocations on to server objects, static and dynamic binding are both possible. For example, if kernel ports are used by the subcontract to provide 'direct' access to the servers methods, then the binding would be static. This is because a method invocation on an object reference would always invoke the same server method. However if the subcontracts were to use message passing then the server could change its methods and the binding would be dynamic.

Parameter Passing

Users use the native parameter passing techniques of their programming language to pass arguments to the generated stubs. These stubs then pass the arguments on to the proxy objects. The types of the parameters are restricted to the types defined in the IDL. By default parameters are passed by-move, however they can also be passed by-copy. Internally the proxies call subcontract routines to marshal and unmarshal the parameters. Unmarshaling involves recreating an object given the marshaled version, this is done by a subcontract at the receiving end. A marshaled object contains the data of a proxy and a subcontract identifier which identifies the subcontract type needed to reconstruct the object.

Security

Enforcing Encapsulation. The only way that a Spring object can be accessed is through stubs generated from IDL descriptions of the object's interface. Also access to objects in other domains is only possible through subcontracts which prevent server data from being accessed directly.

Authorization. There are a number of authentication/authorization mechanisms available in Spring. The name service provides security by only allowing authorized principals access to objects. An object can also define its own access control list, which is checked before any client gets a reference to a proxy of that object. Such a reference is like a capability and offers anyone who has it access to the object.

Because untrusted parties may participate in the naming process the name service allows trust to be established where appropriate by means of authentication. The name service also protects the name service information from unauthorized operations (e.g. it ensures that clients performing naming operations are permitted to do so).

Authentication is achieved through authenticated directories which remember the principal for whom operations are being performed. This principal is most often established when the initial directory for an address space is acquired. Once the principal is determined the client can use that directory without authentication until a directory or object which does not trust the context is encountered. When this happens the client must authenticate itself to the directory or object and proceed from there.

An object can define an ACL of users and access rights which is checked at runtime (e.g. when a client asks for a proxy) to determine whether a client is allowed to access the object. When a client proves that it is allowed to access an object, the object's server creates a proxy and gives it to the client.

Auditing. Spring does not provide any auditing mechanisms.

Atomicity

Spring does not provide support for atomic actions.

3.5.2 CORBA

Static and Dynamic Invocation

CORBA supports both static and dynamic invocation.

Invocation

Methods are invoked by users on object references. For static invocation this is done in a language specific way. The method is invoked on an object reference and through language mapping this leads to an invocation on an object type specific client stub. This stub first marshals the parameters and then passes an invocation request to the runtime system. For dynamic invocation library calls have to be made in order to build an invocation, and execute it. This entails acquiring interface information from the interface repository and using it along with the arguments to be passed as parameters to create an argument list. Using the argument list and a reference to the object a request is created and executed. This causes the arguments to be marshaled and the runtime system to be contacted as with static invocation. When the remote operation is finished the results are received, unmarshaled and passed back up to the client.

The runtime system is responsible for and manages control transfer and data transfer to the actual object implementation and back. The way that the control and data transfer is actually achieved is implementation specific.

At the server side when a message comes in, parameters are unmarshaled and the appropriate method is invoked.

Dynamic Binding

The method actually executed depends on the object being referenced, also dynamic invocation makes static binding impossible.

Parameter Passing

Users use the native parameter passing techniques of their programming language to pass arguments to the client stubs. The types of the parameters are however restricted to the types defined in the IDL. The arguments can be passed by value, by result or by value/result. Internally the parameters are marshaled and unmarshaled to and from a standard representation (Common Data Representation) and are copied to the server. CORBA objects are always passed by reference.

Security

Enforcing encapsulation. Encapsulation is enforced because only the CORBA runtime system actually has direct access to a server object. All other entities must therefore access server objects through the runtime system which only allows access as defined by the object's interface.

Authorization and Auditing. The runtime system guarantees that for every method invocation it will identify the principal on whose behalf the request is performed. The server object can obtain this information from the system and do with it what it likes, however the runtime does not enforce any specific security policies.

Further security policy will be provided by the security service. Servers will use ACLs to authorize clients and acquiring an authenticated identifier can be done using third party authenticators. Auditing mechanisms will also use the authenticated identifiers.

Atomicity

Atomic actions will be provided by the object transactions service (OTS). The OTS supports flat (required) and nested (optional) transactions. Both CORBA and non CORBA applications will be able to participate in the transactions which can span domains. To make an object transactional (that is to allow it to take part in transactions) it must inherit from an abstract OTS class.

3.5.3 Emerald

Static and Dynamic Invocation

Emerald supports static invocation.

Invocation

Methods are invoked directly on a variable containing an object reference. When a method is invoked on another object the thread of control enters that object for the duration of the call. Internally when methods are invoked on objects on the same node then the method is called directly. However when methods are invoked on remote objects then the context of the process is collected and moved to the remote object's location where execution continues. In order to locate an operation (i.e. find the location of the possibly remote object and the address of the actual code) the vector of method pointers contained in variables is used. If the object is local then the pointer is used to find the implementation, otherwise the request is sent to the object's forwarding address (see Finding Objects).

Eventually the request finds its way to a node where the object implementation is located and the method invocation takes place there.

Dynamic Binding

Because the type system (see Typing) is based on interfaces, the methods actually executed depend on the object referenced and not on the syntactic type of the variable referencing it.

Parameter Passing

In Emerald parameters are mostly passed by reference. There are two other possibilities which are used to improve performance: pass by move, and pass by visit. Whether call by move (or visit) is used can be explicitly specified by the user or can be automatically determined by the compiler based on compile time information. Both are hints to the system to move the referenced object to the node where the method will be executed. With pass by visit semantics, the object is also moved back after execution of the method.

Parameters may also be declared to be value or result parameters.

Security

Enforcing Encapsulation. Encapsulation is enforced at compile time. The runtime system itself does not prevent data (for local and direct objects) from being accessed directly.

Authorization and Auditing. Authorization and Auditing are not supported by the system.

Atomicity

Emerald does not provide any mechanisms for atomic actions.

3.5.4 Obliq

Static and Dynamic Invocation

Obliq supports static invocation.

Invocation

Obliq supports four basic operations on objects: selection/invocation, updating/overriding, cloning and aliasing. Selection/invocation returns a value when applied to a value field and invokes a method when applied to a method field. Updating/overriding updates a value field or overrides the method invoked through a method field. Cloning produces a new object from existing objects. It can be used with one or more arguments. With one argument a new object with the same fields and values as the argument object is created. With more than one argument a single object is created that contains the values, methods, and aliases of all the argument objects. Aliasing replaces a field's contents with an alias. It is also possible to alias all the components of an object at once.

Remote invocation of procedures is done at remote compute servers (or execution engines). These are remote procedures which accept procedures or methods as arguments and execute them. The language defines a number of primitives for exporting and importing these engines.

Internally when a method is invoked on a remote object, the arguments are transmitted over the network to the remote site, the results are computed remotely and the final value is returned to the site of invocation[7].

Dynamic Binding

The code of a method can be changed at runtime, therefore the actual code that will be run must be determined at runtime.

Parameter Passing

Any values can be transmitted as parameters including procedures and methods. When these are transmitted, lexically scoped free identifiers retain their bindings to the originating sites. Objects are passed by reference, whereas other values are passed by copy. Non-object values are passed by copy, whereas objects are passed by reference.

Security

Enforcing Encapsulation. All state is visible and by default also accessible in Obliq objects. However an object can be protected meaning that it rejects all external update, cloning, and aliasing. This provides some encapsulation, however the object's state remains visible.

Authorization and Auditing. Obliq does not provide any authorization or auditing mechanisms.

[7] A call on a remote network object goes through the object's proxy which marshals the arguments and does a remote procedure call on the server object.

Atomicity

Obliq does not provide any mechanisms for atomic actions.

3.5.5 Fragmented Objects

Static and Dynamic Invocation

SOS supports static invocation.

Invocation

Users invoke methods directly on isolated objects or fragments. Internally the fragments use their common connective object to invoke methods in the group interface. When fragments invoke methods in their group interface then a message must be built and passed to the connective object. The connective object then sends the message to the destination(s), where the parameters are unmarshaled and the appropriate method is called. FOG automatically generates stubs which do the marshaling and unmarshaling of the arguments.

Dynamic Binding

SOS relies on C++'s or FOG's dynamic binding mechanisms to let users customize methods which are upcalled. Programmers may also use this dynamic binding for distributed object fragments and isolated objects.

Parameter Passing

In FOG arguments can be passed by copy or by reference. FOG also allows one to specify parameters as value, result and value/result. When passing arrays and pointers the size of the data to be passed can be given using a special construct. If a distributed object is passed by reference then FOG tries to migrate a fragment of the object into the callee's address space. The migration code for each object is written by the programmer and must decide how best to do the migration, e.g. whether it should migrate itself, a copy, or create a fragment. For parameters which are not distributed objects, value/result parameter passing is used for passing arguments by reference.

Security

Enforcing Encapsulation. Encapsulation of the state of an isolated object or a fragment is enforced at compile time by the language that the objects are implemented in. For group communication (between fragments) encapsulation is enforced because fragments can only access each other through connective objects using methods belonging to the group interface.

Authorization. The only authorization mechanism provided by SOS is that connective objects only allow their owner(s) to use them. This implements authorization for the group interface only. Any authorization for the public interface must be done by the fragments themselves.

Auditing. SOS does not provide any auditing mechanisms.

Atomicity

SOS does not provide any mechanisms for atomic actions.

3.5.6 Globe

Static and Dynamic Invocation

Globe supports static invocation.

Invocation

Methods are invoked by users directly on an isolated object. Internally they are invoked through a binary interface which consists of an array of method and state pointer pairs. This binary interface is accessed through language mappings. When a method is invoked on a fragment it is actually invoked on the control object which exports the same interface as the semantics object does. The control object synchronizes access to the distributed object by serializing access (through locking) to the semantics object. It then decides how to handle the invocation. For example the semantics object might be allowed to modify local state or a message might be sent to another fragment invoking one of its methods. When a message arrives from another fragment a popup thread is created which performs the call through a callback interface to the replication object.

Dynamic Binding

When a method is invoked on an object the interface table that the reference points to is used to determine which method should be executed. Because this table can be updated dynamically, binding cannot be done statically.

Parameter Passing

The parameter passing semantics seen by users is that of the language that they are programming in. When an object is passed as a parameter then it is passed by reference. Internally all the arguments have to be marshaled before a message is sent to any other objects. Likewise the arguments have to be unmarshaled at the receiving end. The marshaling and unmarshaling is handled transparently in the fragments of the distributed object.

Security

Enforcing Encapsulation. Encapsulation is enforced because methods can only be invoked through an interface and thus the actual data of an object cannot be accessed directly.

Authorization and Auditing. Authorization and Auditing have not yet been included in the model.

Atomicity

Globe does not support atomic actions.

3.5.7 Legion

Static and Dynamic Invocation

Legion supports static invocation. If the implementation language or a special library allowed method calls to be dynamically constructed, then dynamic invocation could also be possible. This is due to all class objects containing a method for getting that classes interface.

Invocation

Methods are invoked using native language constructs and the compiler produces code which makes Legion calls. Internally methods are invoked on object identifiers. The address corresponding to the object identifier is found and then a message is sent to the destination asking it to perform the method.

Legion uses standard protocols (e.g. XTP) and the communication facilities of the host operating systems to support communication between Legion objects.

Method calls are asynchronous, the caller can choose whether to block waiting for results, or proceed until they are needed. Also the callee may accept the methods in any order that it chooses.

Dynamic Binding

For remote objects the method to be executed must be determined at run time when the invocation is performed.

Parameter Passing

Formal parameters can be passed by value, result or value/result. Non-object arguments are always passed by-copy. Data coercion to and from an intermediate format is done by type specific coercion functions which are generated by the IDL compiler. The runtime system decides whether it is necessary to use the functions or not. Legion objects are passed by sending the object's identifier.

Security

Enforcing Encapsulation. The runtime system does not allow objects to be accessed in any way other than by calling their methods.

Authorization. In Legion authorization invocation rights are granted per method. The basic concepts in Legion's security model [53] are as follows. Every object must provide certain security related member functions. These may be implemented by the object itself or by inheriting them from other objects. User objects can play two security roles, that of responsible agent (RA) or security agent (SA). The responsible agent identifies the principle who was (indirectly) responsible for initiating the call and the security agent is responsible for enforcing security policies. Every invocation of a method is performed in an environment consisting of a triple of objects: SA, RA, CA (calling agent - the agent directly responsible for the call). There are a set of actions that Legion takes at method invocation. The actions are as follows. An object's MayI() method is called before an invocation of any other of its methods. MayI returns a license for the object and all its methods. The license issued depends on the identity of the caller. It is up to MayI how to authenticate the caller, Legion provides the CA, RA and SA and these may (but do not have to) be used. A license is a capability which is checked before every invocation of the object for which it was issued. It may be revoked if certain conditions no longer hold. If an SA is defined in an environment then all calls must be passed through the SA using its pass method. The SA then decides if a call may even be attempted.

Auditing. There are no auditing mechanisms provided by Legion.

Atomicity

Legion provides atomic actions for stateless objects, but not yet for stateful objects. Note that providing atomic actions for stateless objects is not completely trivial as changes to output parameters must also be taken into account.

3.6 Concurrency

	Spring	CORBA	Emerald
concurrent access	y	y	y
isolated actions	n	y	y
synchronization	n	y	y

	Obliq	FO	Globe	Legion
concurrent access	y	y	y	n
isolated actions	n	n	y	-
synchronization	n	n	y (not specified how)	y

3.6.1 Spring

Concurrent Access to Objects

Both local and global objects can be accessed simultaneously by multiple clients in the same address space. Global objects can also be accessed by multiple clients active in different address spaces. Internally, for global objects a server process is started for each request and thus multiple requests can be handled simultaneously. The subcontracts might, however, restrict concurrent access to an object's methods if they use blocking communication.

Isolated Actions

Spring does not offer any mechanisms for isolated actions.

Synchronization

Spring does not offer any mechanisms for synchronization.

3.6.2 CORBA

Concurrent Access to Objects

Whether concurrent access to objects is possible depends on the implementation of the runtime system and of the object itself. The user of an object is, however, generally not aware of the implementation of an object and therefore not aware of whether concurrent access is actually possible or not.

Isolated Actions and Synchronization

Isolated actions and Synchronization are provided by the concurrency service. The concurrency control service provides interfaces to acquire and release locks. The role of the concurrency service is to prevent multiple clients from simultaneously owning locks to the same resource if their activities conflict. The concurrency service is tightly coupled to the transaction service and therefore the locks may be acquired for the transaction service and may be used in nested transactions. Of course if the object implementation does not allow concurrent access to the object then all actions are automatically isolated.

3.6.3 Emerald

Concurrent Access to Objects

Multiple clients can execute any of an object's methods simultaneously.

Isolated Actions

Isolated actions are provided by means of monitors. A monitor can be defined within an object and includes a number of the object's method definitions. The methods in the monitor can only be executed by one process at a time. The methods within a monitor may call methods of other objects, however those methods may (if they are not in monitors) be simultaneously invoked by multiple clients.

Synchronization

Synchronization is provided through condition objects. The condition objects can be signaled and waited on. Condition objects can only be used within a monitor.

3.6.4 Obliq

Concurrent Access to Objects

Objects can be accessed concurrently by multiple remote and local clients.

Isolated Actions

Obliq provides isolated actions through serialized objects. In a serialized object at most one client can operate on a value or run one of its methods at any one time. Self inflicting operations are not serialized in this way. Internally serialized objects have an implicit object mutex associated with them. External operations always acquire the mutex of an object, and release it on completion. Self-inflicted operations do not need to and so never acquire the mutex of their object. When serialized objects are cloned a fresh mutex is created for the clone.

Synchronization

Obliq provides a conditional synchronization statement which makes use of conditions and guards. The synchronization statement can only be used inside of a serialized object's method. The statement evaluates a condition and a boolean guard expression. If the guard evaluates to true then the statement ends and control can continue in the serialized object. Otherwise the object is 'unlocked', that is other clients can access it, and the client executing the synchronization statement waits on the condition. When the condition is signaled the object is 'locked' and the guard is reevaluated. This continues until the guard is true and the statement ends.

3.6.5 Fragmented Objects

Concurrent Access to Objects

Isolated objects can simultaneously be accessed by multiple clients resident in the same address space. Distributed objects can be simultaneously accessed by multiple clients resident in the same address spaces as any of the objects fragments.

Isolated Actions

Isolated actions are not supported by the system.

Synchronization

No synchronization mechanisms are offered by the system.

3.6.6 Globe

Concurrent Access to Objects

An isolated object can be accessed concurrently by multiple objects as long as they reside in the same address space as the object. However the accesses to fragments of a distributed object will be serialized by the fragment's control object. Distributed objects may be accessed concurrently by multiple objects as long as the acceses are through separate fragments (so that the calls are not serialized).

Isolated Actions

Control objects serialize all access to a semantic object's methods, however there is no mechanism to serialize actions invoked on different fragments which are part of a distributed object.

Synchronization

Synchronization will be possible, how has not yet been specified.

3.6.7 Legion

Concurrent Access to Objects

Legion does not provide concurrent access to objects.

Isolated Actions

There is no need for isolated actions because objects cannot be accessed concurrently.

Synchronization

Due to the fact that objects cannot be concurrently invoked they can be used by processes to synchronize.

3.7 Object-Oriented Programming Issues

	Spring	CORBA	Emerald
inheritance	int	int	n
multiple inheritance	y	y	n
dynamic inheritance	n	n	n
polymorphism	i	i	p, i, o

	Obliq	FO	Globe	Legion
inheritance	y	y	not specified	y
multiple inheritance	n	n	n	y
dynamic inheritance	n	n	n	y
polymorphism	p	i, o, c	n	n/a

3.7.1 Spring

Inheritance

Spring offers interface inheritance through IDL.

Multiple Inheritance

Spring supports multiple inheritance through its IDL.

Dynamic Inheritance

Spring does not offer dynamic inheritance.

Polymorphism.

Inclusion polymorphism is available through the type system. Due to the fact that an object's type depends on its interface, an object can take on any type which is based on a subset of its interface.

3.7.2 CORBA

Inheritance

CORBA offers interface inheritance through IDL. Implementation inheritance may be provided by higher level tools.

Multiple Inheritance

IDL also offers multiple inheritance.

Dynamic Inheritance

CORBA does not offer dynamic inheritance.

Polymorphism

Inclusion polymorphism is available through the type system. Due to the fact that an object's type depends on its interface, an object can take on any type which is based on a subset of its interface.

3.7.3 Emerald

Inheritance

Emerald does not provide any form of inheritance.

Multiple Inheritance

Emerald does not provide multiple inheritance either.

Dynamic Inheritance

Nor does it provide dynamic inheritance.

Polymorphism

Emerald provides parametric, inclusion and overloading polymorphism. Parametric polymorphism is present because functions can work on objects of different types as long as the expected and actual types conform (see Typing). The Requirements for inclusion polymorphism are satisfied because an object can have multiple types due to the type system in Emerald. Overloading is also possible because an operation is not uniquely defined by its name.

3.7.4 Obliq

Inheritance

Cloning is used to simulate implementation inheritance. The new object is an independent object and is in no way dependent on the parent.

Multiple Inheritance

Cloning can also be used for multiple inheritance.

Dynamic Inheritance

Obliq does not support dynamic inheritance.

Polymorphism

Obliq provides parametric polymorphism because the language is untyped, so any values can be used as parameters.

3.7.5 Fragmented Objects

Inheritance

FOG supports C++'s implementation inheritance.

Multiple Inheritance

FOG also support C++'s multiple inheritance.

Dynamic Inheritance

FOG does not support dynamic inheritance.

Polymorphism

FOG supports inclusion, overloading and coercion polymorphism. Inclusion is supported due to the type system. Overloading and coercion are supported due to FOG being based on C++.

3.7.6 Globe

Inheritance

Whether inheritance will be available or not has not yet been specified.

Multiple Inheritance

Not yet specified.

Dynamic Inheritance

Globe does not support dynamic inheritance.

Polymorphism

Globe uses binary interfaces, so the polymorphism issues are not dealt with by Globe but the languages and compilers used to implement Globe objects and clients.

3.7.7 Legion

Inheritance

Legion offers dynamic implementation inheritance using the derive method of a class object. Derive creates a new subclass class object which inherits everything from the class object. Users use native language techniques for inheritance which are then mapped to a call to derive by language mapping.

Multiple Inheritance

Multiple inheritance is achieved using a class object's inheritfrom method. This causes member functions from the argument to be added to the class object's interface. Future instances of the class will then have the extended interface.

Dynamic Inheritance

Dynamic inheritance is also possible using a class object's inheritfrom method.

Polymorphism

Legion offers overloading as the only form of polymorphism.

3.8 Typing

	Spring	CORBA	Emerald
type checking	s, d	s, d	s, d

	Obliq	FO	Globe	Legion
type checking	d	s, d	d (not specified)	d

3.8.1 Spring

Type

An object's interface defines its type.

Subtyping

The type hierarchy follows the inheritance hierarchy, so an object is subtyped when it inherits another's interface.

Type Checking

Type equality is based on equality of interfaces. Two objects have the same type if their interfaces are the same. All arguments to methods are statically checked by the programming language used. Dynamic type checking is done when the type of an object is to be narrowed (changed to a subtype). Before this can be done the object is checked for the new type and the change may fail or succeed.

3.8.2 CORBA

Type

An object's type is based on its interface.

Subtyping

The type hierarchy follows the inheritance hierarchy. An object's type can become a subtype of another object by inheriting from it.

Type Checking

Type comparison is based on interface comparison. An object satisfies an interface if it can be specified as the target object in each potential request described by the interface. Both static and dynamic type checking are done.

3.8.3 Emerald

Type

The types of objects are based on method signatures. Abstract Types are objects that explicitly define types. They are explicitly defined by specifying an interface. A client can determine the type of an object using the typeof primitive, this determines the maximal type (the largest type that it can belong to) of an object.

Subtyping

The type hierarchy is based on type conformity. S conforms to abstract type T if an object of type S can always be used where one of type T is expected. This means that S provides at least the operations of T: corresponding operations have the same number of parameters and results, the types of the operations conform (S's to T's) and the types of the parameters conform (T's to S's!). Any object that supplies the interface of T is a subtype of T. Subtype relations are implicit - there is no means for explicitly declaring the conformity relation.

Type Checking

Type checking is based on type conformity. Each identifier has a declared type which is evaluated at compile time, this is called its syntactic type. Any object to which such an identifier is bound must satisfy (conform to) the syntactic type of that variable. Static type checking is done whenever possible, otherwise dynamic type checking is done.

3.8.4 Obliq

Obliq is an untyped language. The runtime system is however strongly typed. This means that when objects are used in ways that are not possible (e.g. a nonexistent field is accessed) then precise error information is returned.

Type

Runtime type is based on Modula 3 typing.

Subtyping

For Network objects the subtyping hierarchy follows the inheritance hierarchy.

Type Checking

Only dynamic type checking is done.

3.8.5 Fragmented Objects

Type

FOG and SOS typing is based on C++ typing which is based on classes.

Subtyping

Subtyping follows the inheritance hierarchy, thus objects can be subtyped through inheritance.

Type Checking

Static type checking is based on C++ typing and is done by the FOG compiler. FOG also generates code to check the interfaces at run time. This means that dynamic type checking is based on equality of interfaces rather than classes.

3.8.6 Globe

What kind of typing Globe will have is not known yet.

3.8.7 Legion

Type

Type for Legion objects is based on classes

Subtyping

Subtyping is based on the kind-of relationship which follows hierarchies built through derive.

Type Checking

Objects are equal if they have the same class. Type checking is both static and dynamic.

3.9 Persistence

	Spring	CORBA	Emerald
persistent objects	y	y	y
durable actions	n	y	y

	Obliq	FO	Globe	Legion
persistent objects	n	y	y (not specified)	y
durable actions	n	y	not specified	n

3.9.1 Spring

Persistent Objects

Persistence in Spring [40] is provided through naming. If the part of the name space where an object is found is persistent then the object will also be persistent. The server object controls how to make its own state and that of its proxies persistent.

Creation

When an object is bound to a name in a name space which is persistent then the object will become persistent.

Deletion

When an object is unbound from a name space that is persistent then it will no longer be persistent and its persistent image (e.g. a copy of the object on secondary storage) will be deleted.

Melting

An object bound in a persistent name space is melted as soon as its name is resolved by a client. Internally a freeze token can be melted only at the same freeze service where it was acquired at. Melting causes an object which shares some state with the original object to be generated.

Freezing

An object is typically frozen when no clients have access to it, however an object can also offer methods which let it be explicitly frozen at any time by a client.

An object can be frozen directly through its freezing interface but this is usually done by a (persistent) directory object when the object is bound to a name. When an object is frozen a freeze token is generated. A freeze token persistently represents an object's state and the object can be reconstituted from the token at a later time. Freezing can either be done by the object itself or by a freeze object which is chosen by the object.

Durable Actions

Spring does not provide any mechanisms for durable actions.

3.9.2 CORBA

Persistent Objects

The runtime system provides minimal persistence support. Every object is persistent in that a client can use an object reference at any time - whether an object is activated or not (see Creation). However the system provides only a small amount of extra storage (outside of the objects implementation) which is usually not enough to store the state of the object. This data can be used to store information about, for example, the location of the persistent state.

The Persistent Object Service allows the state of an object to be saved in a persistent store and restored when it is needed. The service defines the interfaces to the objects which compose it but not the implementations of these objects (which could be database systems or just simple file systems). The persistence of an object may be completely transparent to a client or the client may be involved in the persistence management. The client chooses how much management it wants, however a server object can also completely hide its persistence. The Persistent object is ultimately responsible for its persistence - it can manage the persistence itself or delegate it to the persistence services.

Creation

An object can become persistent by inheriting from a special Persistent Object class, it must also provide a mechanism for freezing its state. Clients typically interact with the interface provided by this persistent object class.

Deletion

The persistent object's delete method allows a persistent object to be deleted.

Melting

Clients can invoke a persistent object's restore method.

Freezing

Clients can invoke a persistent object's store method. Persistent object method invocations are routed by the Persistent Object Manager to the appropriate Persistence Data Service which is an interface to the particular datastore implementation.

Durable Actions

An action performed in a persistent object can be made durable by explicitly freezing the object.

3.9.3 Emerald

Persistent Objects

Emerald provides persistent objects through the checkpoint primitive [24].

Creation

An object becomes persistent the first time that it is checkpointed. When this happens the object is written to stable storage and becomes a persistent object. Distributed persistence is also possible by executing checkpoint at ¡nodelist¿. Nodelist lists nodes to which the object is copied after being checkpointed.

Internally when an object is checkpointed it is marshaled (as when it is moved, except that process information is not needed) and written to disk.

Deletion

Persistent objects cannot be deleted.

Melting

Melting is triggered automatically. When an object melts a designated recovery process starts up. In the internal view melting is triggered automatically when a copy of an object cannot be found during a remote invocation, or when frozen copies of an unavailable object are found during garbage collection.

Freezing

A persistent object becomes frozen when its memory image is garbage collected.

Durable Actions

Any change to an object's state can become durable when the object is checkpointed.

3.9.4 Obliq

Persistent Objects

There is no mechanism for persistence in Obliq.

3.9.5 Fragmented Objects

Persistent Objects

In SOS persistent objects are those which inherit from a special base persistent object.

Creation

Classes which wish to be persistent must inherit from a special persistent object. Internally a persistent object has access to a storage object, which is a fragment of the storage service object. The persistent object's methods have privileged access to the storage object (for communication with the storage service), and the storage object is automatically imported when a persistent object is created.

Deletion

A persistent object once stored on disk can never be deleted.

Melting

It has not been specified how/if a frozen object can be melted.

Freezing

A persistent object can be frozen by destroying its memory image.

Durable Actions

When a persistent object checkpoints, its state is written to secondary storage, also any other objects (persistent or not) referenced by the object are written to disk.

3.9.6 Globe

Persistent Objects

Persistence will be available in Globe but the details have not yet been specified.

3.9.7 Legion

Persistent Objects

All Legion objects are persistent and may be melted or frozen, this is not visible to users. A frozen object is represented by an Object Persistent Representation which is a description of the frozen object and can be used to melt it e.g. an executable file and state. Moving objects between melted and frozen state is the responsibility of the object managers but every object will have methods which can be called to save and restore its state.

Creation

All objects are automatically persistent.

Deletion

When an object is destroyed its persistent image is also deleted.

Melting

Melting happens automatically when a method of a frozen object is called. Internally the object manager finds a suitable location for the object, allocates process resources, begins execution of an appropriate implementation and installs and initializes the state.

Freezing

The object manager decides when an object is to be frozen. All objects have a method for freezing themselves. When the object manager decides to freeze an object then it calls this method and releases any process resources associated with the object.

Durable Actions

When an object is frozen then the state is saved.

3.10 Failure and Fault Tolerance

	Spring	CORBA	Emerald
failure detection	t	e	e
availability	n	n	y
exception handling	y	y	return values

	Obliq	FO	Globe	Legion
failure detection	e	e	n	t, e
availability	n	n	y(+-)	n
exception handling	y	events	n	y

3.10.1 Spring

Failure Detection

When a called address space crashes the kernel forces all active incoming calls to return with an error code. When the caller's address space crashes the call is arranged to return to the kernel (of the machine where it is executing) so that the crashed address space can be removed. The alert mechanism allows a called address space to detect that its caller has gone away and neatly abort whatever it was doing. Alerts are propagated down the call chain and to any threads that it has called (e.g. invocations in different address spaces)

Availability

Spring does not guarantee availability of objects

Exception Handling

A method may raise exceptions.

3.10.2 CORBA

Failure Detection

The Corb runtime system produces exceptions on invocation failure.

Availability

CORBA does not guarantee availability of objects.

Exception Handling

Exceptions can be produced by the runtime system or by an object and these are then propagated to the client. The exceptions are passed to the client through an appropriate mapping into a native mechanism for the language in which the client is implemented.

3.10.3 Emerald

Failure Detection

Existing faults (specifically node crashes) can be handled if the desired object cannot be accessed and that object is a distributed persistent object. This is done by melting one of the frozen copies of the object. Internally what happens is that when an attempt to locate an object finds only frozen copies of the object then an election is started to find one to melt. During the location procedure (which fails if no active object is found) the querying node is informed of the most recent frozen replica (determined by an election protocol) on the network and this is chosen as the new active object.

Availability

The distributed persistence (i.e. replication) allows objects to remain available in the face of failures, however this is only possible for objects which checkpoint, and is not transparent.

Exception handling

Failures are signaled through return values. There is also a special return and fail statement which allows graceful failure.

3.10.4 Obliq

Failure Detection

Communication failures produce exceptions that can be caught.

Availability

Obliq does not guarantee availability of objects.

Exception Handling

Exceptions can be raised and caught. Exceptions are equal if their names are equal, so an exception can be caught at a site different from the one in which it was raised.

3.10.5 Fragmented Objects

Failure Detection

Node crashes can be signaled using dependency families and events. A dependency family is a set of objects which agree to a common signaling protocol; they join the family and receive any events generated within the family, or any system events. The objects are responsible for the handling of these events.

Availability

Availability of objects is not guaranteed.

Exception Handling

Errors can be signaled by generating a user defined event within a dependency family. FOG also offers C++ like exceptions.

3.10.6 Globe

Failure Detection

No special measures are taken by the Globe system to detect and handle failures.

Availability

Availability is not a guarantee of the system. However due to the distribution of objects they will most likely remain available in the face of problems (crashes, etc). The degree of availability will largely depend on the how the objects are distributed (e.g. degree of replication, centralization, etc.).

Exception Handling

Globe does not provide any exception handling.

3.10.7 Legion

Failure Detection

For fail stop faults of hardware components the underlying message system guarantees delivery of messages. If a host fails then legion reconfigures itself to remove the host, it also reconfigures when the host recovers again. In the case of a network partition legion treats the unreachable hosts as dead.

The communication layer detects attempts to communicate with invalid object addresses.

Availability

Legion does not guarantee availability of objects.

Exception Handling

Legion supports exception handling.

3.11 Programming Interface

	Spring	CORBA	Emerald
programming interface	IDL	IDL	languge dependant

	Obliq	FO	Globe	Legion
programming interface	Lang	FOG, C++	IDL, existing language	IDL, mentat

3.11.1 Spring

The Spring model is language independent. An object's interface is defined in an IDL. The actual implementation of the object is defined in another language (e.g. C, C++). An IDL compiler produces: a language specific form of the interface (e.g. C++ header files), client-side stub code which is later dynamically linked into a client's program and server-side stub code which is linked into the server to translate incoming remote object invocations, i.e. they call the correct methods.

3.11.2 CORBA

The CORBA model is language independent. Interfaces are defined in an IDL while the implementations of an object can be defined in any other programming language as long as there is a binding between IDL and the language. This binding should be the same for all CORBA implementations. Language mappings allow objects and the runtime system to be accessed through regular language features. An IDL precompiler produces language skeletons for the actual implementation of the object. The object implementation is added to the skeletons and the code is compiled generating interfaces for the interface repository, client stubs, server stubs and server code.

3.11.3 Emerald

The model is dependent on the Emerald programming language.

3.11.4 Obliq

The model is dependent on the Obliq language.

3.11.5 Fragmented Objects

The model is meant to be language independent, however some aspects of SOS are dependent on features of C++ or FOG. These aspects include: upcalls which depend on C++'s virtual functions, the system interface which depends on C++ constructors and destructors and SOS's dependence on C++'s 'this' pointer for authentication (in connective objects) when making method calls.

3.11.6 Globe

The Globe model is language independent. The class and interface definitions are described in either an existing language or through an IDL. If an IDL is used then the implementation can be defined in an existing language as long as an IDL binding to that language exists.

3.11.7 Legion

The Legion model is language independent. Object interfaces are defined in an IDL or Mentat [17] and the implementation in another language. This code then has to be compiled with special Legion compilers to produce Legion objects or programs.

3.12 System Issues

	Spring	CORBA	Emerald
interoperability	n/y	y, y	n,n
scalability	+	depends on implementation	-

	Obliq	FO	Globe	Legion
interoperability	y, y	n	n, y	n, y
scalability	+-	+-	++	+

3.12.1 Spring

Interoperability

Spring does not specify interoperability with different models. Interoperability between different implementations should be possible because each implementation would be responsible for its own objects and when objects are passed over the network they are always marshaled into intermediated forms. Also objects do not have to be searched for upon method execution and so communication between machines with different implementations would be minimal.

Scalability

The communication mechanism is expandable which allows different types of communication depending on how objects are to be used, this could be point to point communication, broadcasting, etc. In this way objects that are expected to have many clients can be programmed using subcontracts that allow the object to be replicated or cached to avoid problems.

The naming service allows private name spaces for each address space so that name searches do not always have to be done starting at some 'root' directory. Also shared name spaces can be attached to an address spaces name space so that shared objects can easily be found. Because directories are normal objects name spaces can also be dynamically expanded. Popular name spaces could have their nodes replicated to reduce congestion.

Other

System Structure The Spring operating system is organized as a microkernel. The nucleus and virtual memory manager run in kernel mode, all other system services are implemented as user level objects. The services provide object-oriented interfaces, and clients communicate with the services by invoking their methods. The nucleus manages processes and IPC. It supports three basic abstractions: Domains - provide an address space for applications to run in, Threads - execute within a domain and Doors - support object-oriented calls between domains. A door is basically an entry point into a domain. All objects in Spring reside in a domain.

An important concept in Spring is the subcontract. Subcontracts are replaceable modules which control the basic mechanisms of object life, invocation and argument passing. Subcontracts cannot be accessed directly by clients.

3.12.2 CORBA

Interoperability

CORBA implementations are interoperable between different implementations of the CORBA model and may also be interoperable between different models. Interoperability between different models is possible using the dynamic skeleton interface to translate incoming and outgoing requests to and from CORBA requests. Interoperability between implementations is required of CORBA runtime systems (called ORBs). To achieve this CORBA defines the GIOP (General Inter-Orb Protocol) and IIOP (Internet Inter-Orb Protocol) which allow different ORBs to communicate. This way they can find interfaces and implementations stored at other ORBs as well as forward invocation requests to other ORBs. Interoperable Object References allow different ORBs to pass object references to each other.

Scalability

Much of the scalability aspect in CORBA depends on the implementation of the runtime system and services.

Other

Services CORBA has specifications for a number of services. For all the services CORBA defines the interfaces, not the implementations. Also many of the services have not yet been fully specified. Here follows a list of the main services (they have all been described elsewhere): Life cycle, Naming, Transactions, Concurrency, Persistence, Collections, Relationships, Security.

System Structure Every CORBA object has an object adapter (and an object adapter can be responsible for multiple objects) that provides it with a total environment. It takes care of registration of the objects class with the implementation registry, instantiating objects at run time (for balancing the supply of objects with the incoming client demands), generating and managing references for its objects and handling the incoming requests/invocations and routing them to the proper methods. CORBA specifies a Basic Object Adapter that can be used for most ORB objects with conventional implementations.

The Interface Repository stores IDL information in a form available at runtime and is used for getting information about object interfaces when constructing dynamic invocations. Extra information such as debugging information can also be stored there.

The Implementation Repository contains information which lets ORBs locate and activate object implementations. It can also store additional (e.g. debugging) information.

3.12.3 Emerald

Interoperability

There is no specification of interoperability between Emerald and different models or of interoperability between different implementations of the Emerald model.

Scalability

The Emerald model is limited to use on LANs because of its location mechanisms - forwarding addresses and the broadcast protocol. Also the original implementation is limited to any homogeneous nodes, but a new implementation exists which does allow heterogeneous nodes [46].

3.12.4 Obliq

Interoperability

Because Obliq objects are implemented using Network Objects it is possible for them to communicate with other Network Objects. Therefore Obliq should be interoperable with other implementations of the Obliq model as well as with some other (namely Network Object based) models.

Scalability

Most of the scalability of Obliq depends on how scalable Network Objects are. Network Objects cannot be replicated which can cause congestion problems for objects which are heavily used. Also the name service is centralized - being implemented at one network address, even though naming is not used heavily, it can nevertheless cause problems in a wide area application of Obliq.

3.12.5 Fragmented Objects

Interoperability

SOS has no mechanisms for interoperability with different models. Different implementations of SOS could probably interoperate due to well described interfaces and structures (such as references).

Scalability

A number of properties of SOS distributed objects theoretically allow the system to scale well. These include the fact that traditional connective objects can be substituted for more complex ones offering the possibility of implementing replicated objects, or of using more efficient communication protocols. Also the fact that the key services are implemented as distributed objects allows then to be distributed preventing network problems when many objects try to access them at once. Ironically it is dependence on this distribution that actually prevents the SOS system from scaling well. The problem is that the object manager needs to have a fragment in every address space. As the number of connected machines grows the communication between all the fragments will also increase and eventually cause problems.

Other

System Structure SOS consists of a kernel and a number of services. The kernel provides separate address spaces, lightweight threads and communication within the same address space. The services are all implemented as distributed objects. The main services are: the object manager (called the Acquaintance Service) which is a distributed object manager and a fragment of which is present in each address space. The Communication service which provides protocol implementations and takes care of remote communication. The Storage service which handles persistence and the Naming service which provides symbolic and internal name binding.

3.12.6 Globe

Interoperability

The Globe model does not support interoperability between different models but does support interoperability between different Globe implementations.

Scalability

Globe achieves scalability by allowing crucial components of objects to be replaced (by more efficient ones) without affecting the objects themselves. The organization of a number of key services (such as naming and location) into scalable structures (such as trees) also prevents problems when there are many objects in the system. By replicating objects which are heavily used problems with many users of objects can also be avoided.

3.12.7 Legion

Interoperability

Legion does not provide any mechanisms for interoperability with different models. It does offer interoperability between different implementations of the model because it uses existing network protocols for communications, and the structure of important entities such as object references is clearly defined.

Scalability

The Legion model provides a reasonable degree of scalability. This is achieved by taking into account the heterogeneity of networks and through Legions use of decentralized object management. The fact that all objects are persistent also takes into account situations where the number of objects is too large for them to reasonably fit into main memory of the computers.

Legion manages and exploits heterogeneity of hardware and configuration. Different architectures (or configurations of the same architecture) are better suited to some problems than others and Legion allows this to be exploited by allowing the scheduling decisions to be user definable. Legion also takes into account that different systems have many differences which have to be masked.

Object managers are responsible for local object management which decentralizes object management, preventing problems with too many objects needing to be managed.

A large drawback is that the Legion model does rely on some centralized logical objects like LegionClass and class objects. This can be solved by replicating these objects and organizing them into hierarchical structures, however a design for this is not incorporated into the model.

Other

System Structure Resources are logically partitioned into possibly non-disjoint domains (called Jurisdictions). Control over these domains is given to object managers (called Magistrate objects).

There are a number of basic core objects which other Legion objects inherit. They can always be reimplemented by the users once inherited. The objects are: LegionObject - defines all the mandatory object functions. All objects are instances of classes were that eventually derived from LegionObject. LegionClass - defines all mandatory class member functions. This object is derived from LegionObject and all classes are eventually derived from LegionClass. LegionHost - models hosts. LegionMagistrate - models Magistrates. It includes functions to manage activation, deactivation and migration of objects in a Jurisdiction. LegionBindingAgent - bind object identifiers to addresses.

Chapter 4

Conclusion

4.1 Summary

The proposed framework for describing distributed object models divides such a model into a number of issues. Each issue is then split up into a number of features which may be present in a model. A common terminology allows these features to be used to describe different distributed object models in a similar way. A model description based on this framework describes, using the common terminology, how each of these features are represented in that model. A number of models have been described in order to show how this framework can be used as well as an initial attempt to provide general descriptions of various existing distributed object models.

4.2 Discussion

The proposed framework provides a starting point for comparisons of different distributed object models. Because all models are described in common terms it is much easier to see in how far aspects of some models resemble those of other models. One problem, however, is finding a good level of detail, that is the descriptions cannot be too detailed, nor too general. If they are too detailed then we no longer have an overview of the features and it is no longer possible to describe models in general terms. Also, the more detailed the descriptions become the more they tend to describe the implementation of a model rather than the model itself. If the descriptions are too general then all of the models will begin to look alike, after all the most general description "the model has objects" would not distinguish any of the models from each other. It is a question thus of finding the 'middle ground' of detail for the descriptions.

Having found a middle ground of detail for the framework, using it as a basis for a comparison of distributed object models still has its problems. In some situations the framework is too specific, that is it does not bring out the big picture of a model. The focus of the framework is biased towards descriptions of features rather than of a model as a whole. For example with Spring and CORBA, we notice that some features are quite similar, especially with regards to interface (as both use the same IDL) and the proxy server structure. However the descriptions do not make clear how similar the two models actually are, in fact the Spring system can actually be viewed as an extension to CORBA [39]. On the other hand, many of the descriptions are too general. In order to make a conclusion about a model's scalability, for example, it is often necessary to know much more detailed information about many of its features than our descriptions can provide. For example, name services play an important part in many of the models which means that they will be used a lot. Such a service can form a bottleneck if it is not cleverly designed and implemented in order to avoid that. However, due to the fact that the descriptions do not go deeper into the mechanics of the name services it is not possible to make a judgment about the possible problems that such a service would cause.

In order to make a general comparison of distributed object models it will be necessary to construct new comparison criteria. These may include some of the features described in the framework (e.g. scalability), but on the whole the divisions suggested by the framework are too fine for a general comparison.

4.3 Future Work

A good comparison of distributed object models will have to be done on a number of levels. First the general characteristics of the different models will have to be compared. These will be the characteristics that most distinguish different models. Next the general descriptions, as presented in this report, can be compared. This will offer a comparison of the various features available in the models. Finally more detailed and model specific descriptions can be compared. These would basically follow the framework presented earlier, but will describe the features in more detail. This way they can provide backing for statements made in the previous two levels as well as allow more detailed comparisons.

The models will first have to be described at all three levels, these descriptions can then be used as the sources for all the comparisons. The higher level descriptions can and should draw on information from the lower levels to support any claims made. The lower levels can of course also refer to characteristics or properties of the models mentioned in the upper levels. In this way the three levels of descriptions can be treated as a whole.

Characteristics described in the first level will relate to the global structure of a model, that is the characteristics will be the ones that distinguish models from each other rather than the most important characteristics. For example, concurrency and management of concurrent actions is very important in distributed programming but it is not an issue that distinguishes distributed object models from each other. The distinguishing issues will include: *the structure and* *relationship between objects*, which will deal with kinds of objects, object structure, composition and object reference as described in chapter 2. Next will come *mobility*, which tells whether objects can be moved or copied between address spaces. *Distribution* will concern the possibilities for fragmentation and replication of objects. *Method Invocation* will described what kind of method invocation is possible as well as how methods are invoked in different situations (e.g. remote vs. local invocations). *Persistence* will refer to the presence or absence of persistent objects and *programming interface* will describe how programmers use the object model. Finally *scalability* will discuss the scalability issues of the model. This level of description can be used not only to compare different models but also as a mechanism for categorizing different models depending on their main characteristics.

The second level will be based on the description framework developed in this report. Important to note is that some of the issues dealt with in the framework received much less attention than others in the object model descriptions. For example, the description have very little to say about the issues of *inheritance* and *failure and fault tolerance*. The reason for this is that they require more specific (e.g. implementation details) information than the other issues. These issues could easily be moved down into the third level where the necessary amount of detail can be given. Also the terminology used in the framework (as well as the terminology used in the first level) will have to be more precise. For example, some of the terms used in this report, like local, plain and isolated object, have very similar meaning and are sometimes used interchangeably. One way to achieve this precision is to introduce the terms and their precise meanings in a separate chapter or section.

The third level will also be based on the framework used in the second level but will describe the models and their features in more detail. This will include implementation details if necessary. Also the terminology will not have to be as general as in the previous two levels, however, different terms should not be used to refer to the same concept.

Bibliography

[1] Local Objects. Internal Research Note #1, February 1995.

[2] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] P. America and F. van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *ECOOP-OOPSLA 90*, Ottawa, Canada, October 1990.

[4] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[5] P. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):87–98, February 1996.

[6] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. Technical Report SRC-115, Digital Systems Research Center, Palo Alto, Calif., February 1994.

[7] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86. ACM, October 1986. Published in SIGPLAN Notices, vol. 21, no. 11, November 1986.

[8] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[9] G. Brun-Cottan and M. Makpangou. Adaptable Replicated Objects in Distributed Environments. Technical Report 2593, INRIA, May 1995.

[10] L. Cardelli. Obliq: A Language with Distributed Scope. Technical Report SRC-122, Digital Systems Research Center, Palo Alto, Calif., June 1994.

[11] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, January 1995.

[12] L. Cardelli and P. Wegner. On Understanding Types, Data Abstractions, and Polymorphism. *ACM Computing Surveys*, 17(4), December 1985.

[13] D. Caromel. Toward a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[14] R.S. Chin and S.T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[15] E. Gamma, R.Helm, R. Johnson, and J.Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software*, chapter 1, pages 1–31. Addison-Wesley, Reading, Mass, 1995.

[16] Y. Gourhant and M. Shapiro. FOG/C++ : a Fragmented-Object Generator. In *Usenix C++ Conference*, April 1990.

[17] A. S. Grimshaw. Easy to use object-oriented parallel programming with mentat. *IEEE Computer*, pages 39–51, May 1993.

[18] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds, Jr. Legion: The Next Logical Step Toward a Natiowide Virtual Computer. Technical Report CS-94-21, University of Virginia, June 1994.

[19] G. Hamilton and P. Kougiouris. The Spring Nucleus: A Microkernel for Objects. In *1993 Summer USENIX Conference*. USENIX, June 1993.

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

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

[22] P. Homburg, M. van Steen, and A. S. Tanenbaum. Distributed Shared Objects as a Communication Paradigm. In *Second Annual ASCI Conference*, pages 132–137, Lommel, Belgium, June 1996. ASCI.

[23] P. Homburg, M. van Steen, and A.S. Tanenbaum. An Architecture for a Scalable Wide Area Distributed System, October 1995.

[24] N. Hutchinson and C. L. Jeffery. An efficient implementation of distributed object persistence. Internal Emerald document, july 1989.

[25] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[26] J. F. Karpovich. Support for Object Placement in Wide Area Heterogeneous Distributed Systems. Technical Report CS-96-03, University of Virginia Department of Computer Science, January 1996.

[27] P. Kougiouris and G. Hamilton. Support for Space Efficient Object Invocation in Spring. In *A Spring Collection*. SunSoft Inc., September 1994.

[28] H. M. Levy and E. D. Tempero. Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation. *Software - Practice and Experience*, 21(1):77–90, January 1991.

[29] M. Lewis and A. Grimshaw. The Core Legion Object Model. Technical Report CS-95-35, University of Virginia, August 1995.

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

[31] F. Manola. MetaObject Protocol Concepts for a RISC Object Model. Technical Report TR-0244-12-93-165, GTE Laboratories, Waltham, MA, December 1993.

[32] F. Manola and S. Heiler. A RISC Object Model for Object System Interoperation: Concepts and Applications. Technical Report TR-0231-12-93-165, GTE Laboratories, Waltham, MA, August 1993.

[33] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An Overview of the Spring System. In *Compcon Spring 1994*, February 1994.

[34] J. R. Nicol, C. T. Wilkes, and F. A. Manola. Object Orientation in Heterogeneous Distributed Computing Systems. *Computer*, 26(6):57–67, June 1993.

[35] O.M. Nierstrasz. A Survey of Object-Oriented Concepts. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. ACM Press, 1989.

[36] Object Management Group. The Common Object Request Broker: Architecture and Specification, version 2.0. Technical Report PTC/96.03.04, OMG, July 1995.

[37] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley, New York, 1996.

[38] M. Papathomas. Concurrency Issues in Object-Oriented Languages. In D. Tsichritzis, editor, *Object-Oriented Development*, pages 207–246. University of Geneva, July 1989.

[39] S. R. Radia, G. Hamilton, P. B. Kessler, and M. L. Powel. The Spring Object Model. In *Usenix Conference on Object-Oriented Technology*, June 1995.

[40] S. R. Radia, P. W. Madany, and M. L. Powell. Persistence in the Spring System. In *3rd Workshop on Object Orientation in Operating Systems*, December 1993.

[41] S. R. Radia, M. N. Nelson, and M. L. Powell. The Spring Name Service. Technical Report SMLI-93-16, Sun Microsystems Laboratories, October 1995.

[42] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A General-Purpose Programming Language. *Software – Practice and Experience*, 21(1):91–118, January 1991.

[43] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *6th International Conference on Distributed Computing Systems*, Boston, MA, May 1986. IEEE.

[44] M. Shapiro. A Binding Protocol for Distributed Shared Objects. In *International Conference on Distributed Computing Systems*, Poznan, Poland, June 1994.

[45] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. SOS: An Object-Oriented Operating System - Assessment and Perspectives. *Computing Systems*, December 1989.

[46] B. Steensgaard and E. Jul. Object and Native Code Thread Mobility Among Heterogeneous Computers. In *15th ACM Symposium on Operating Systems Principles*, pages 68–78, December 1995.

[47] A.S. Tanenbaum, H.E. Bal, S. Ben Hansen, and M.F. Kaashoek. Object-based Approach to Programming Distributed Systems. *Concurrency: Practice and Experience*, 6(4):235–249, June 1994.

[48] C. Tomlinson and M. Scheevel. Concurrent Object-Oriented Programming Languages. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 5, pages 79–124. ACM Press, 1989.

[49] M. van Steen, F. J. Hauck, and A. S. Tanenbaum. A Model for Worldwide Tracking of Distributed Objects. In *TINA '96 Conference*, Heidelberg, Germany, September 1996.

[50] M. van Steen, F. J. Hauck, and A. S. Tanenbaum. A Scalable Location Service for Distributed Objects. In *Second Annual ASCI Conference*, pages 180–185, Lommel, Belgium, June 1996. ASCI.

[51] M. van Steen, P. Homburg, L. van Doorn, A.S. Tanenbaum, and W. de Jonge. Towards Object-based Wide Area Distributed Systems. In L.-F. Cabrera and M. Theimer, editors, *4th International Workshop on Object Orientation in Operating Systems*, pages 224–227, Lund, Sweden, August 1995. IEEE.

[52] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7–87, 1990.

[53] W. A. Wulf, C. Wang, and D. Kienzle. A New Model of Security for Distributed Systems. Technical Report CS-95-34, University of Virginia Department of Computer Science, August 1995.