# Chapter 2

# Distributed Object Systems: An Evolutionary Perspective

This chapter adopts an evolutionary perspective on the development of distributed object systems over the past decades. We will show numerous approaches on how object-orientation and distribution can be combined, ranging from early projects such as the Emerald language, to the most recent Java-based systems. Our goal will be to identify the general trends in these developments, and we will ultimately show how the Pangaea system is a natural step in the evolution towards handling distribution transparently and automatically.

This chapter is not a general treatise of distributed systems. We deliberately confine ourselves to the relation between *object-orientation* and *distribution*. Object-orientation is a programming paradigm that is widely accepted as a mature approach to software development over a wide range of application areas. While object-orientation is not the only programming paradigm in existence, it is the one that both industry and academia have been most decidedly embracing in recent years, as can be seen from the introduction and widespread adoption of languages such as C++, Java, and C#.

We may therefore consider object-orientation as a given fact, an accepted methodology for software construction. The question is, then, how the additional requirement of distribution can be integrated with this paradigm in a natural way, so that the benefits of object-orientation are maintained. Various approaches have been suggested to achieve this integration. To present and discuss them in a systematic way, we will use the hierarchical scheme shown in figure 2.1.

At the lowest level, every distributed system works by *message passing*. The most straightforward approach to writing distributed programs is, therefore, to make message passing available to the programmer. This is realized in libraries such as PVM (Geist and Sunderam 1992, 1993) or MPI (Snir et al. 1995). Despite having originated from a conventional programming background, both of these de-facto standards have been adapted to object-oriented languages, and Java in particular (Yalamanchilli and Cohen 1998, Baker et al. 1999, Ma et al. 2002). Although PVM and MPI are sophisticated and mature, providing for example group communication and advanced synchronization primitives, there is nothing particularly object-oriented about them. They are low-level interfaces to distribution services, independent of the paradigm of the programming language used.

Similarly, a programmer can choose to access the message passing layer almost directly via *socket communication*. Sockets usually provide connection-oriented, stream-based communication, which allows the programmer to send and receive messages of arbitrary length. Still, the programmer is required to define the format of these messages, and the processing routines to handle them, all by himself.

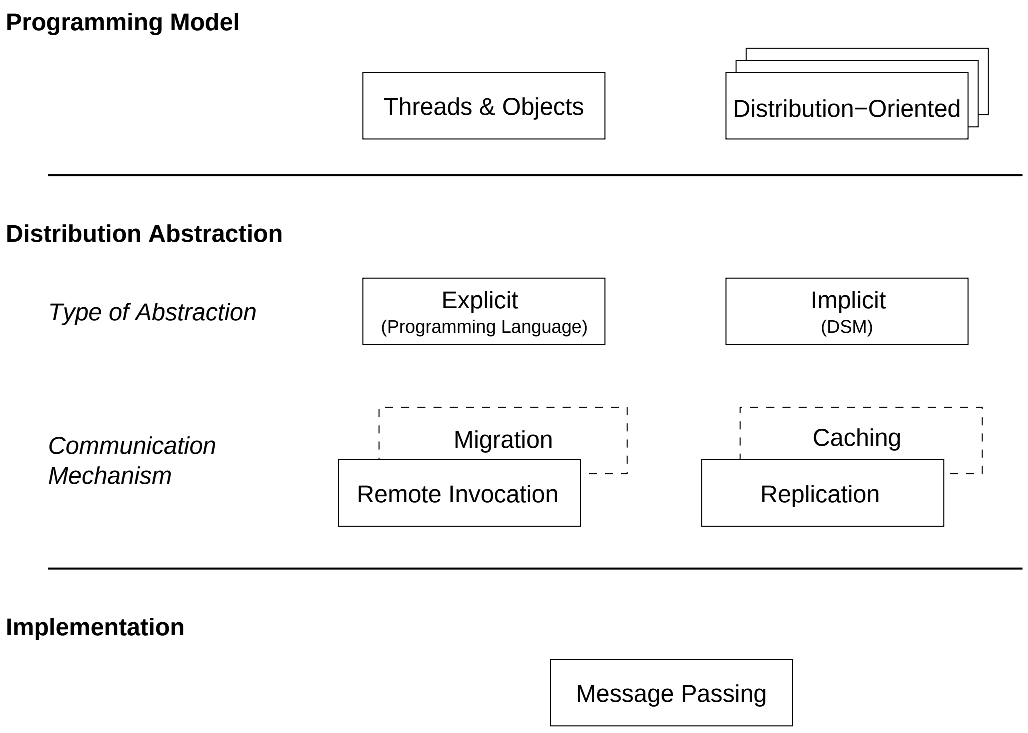


Figure 2.1: Integrating Object-Orientation and Distribution

To achieve a more thorough integration of object-orientation and distribution, the message passing layer must be *abstracted from.* There are two different aspects that characterize this *distribution abstraction layer:* first, it involves the introduction of *communication mechanisms* that are more akin to the object-oriented paradigm, thus being at a higher conceptual level than mere message passing. The basic kinds of these communication mechanisms are *remote invocation*, *migration*, *replication*, and *caching* (we will discuss them in more detail below).

The second characteristic aspect of the distribution abstraction layer is how the new communication mechanisms are made available to the application program. This can be done either *explicitly*, by expressing and controlling them within the programming language, or *implicitly*, by providing an execution environment that handles distribution without the program (and the programmer) being aware of it. The former, explicit approach is also known as the “middleware” approach, while the latter, implicit approach is traditionally referred to as “distributed shared memory” (DSM).

As we will see when discussing individual systems, the choice of communication mechanisms and the type of abstraction (explicit or implicit) are largely independent of each other, and any combination between them can in fact be found in the wild. Traditionally, though, the DSM approach has been more akin to replication and caching, while middleware platforms have mainly been using remote invocation and migration as their communication mechanisms.

On top of the distribution abstraction layer, the actual application program is implemented. In the simplest case, the program adheres to the same *programming model* that is used for a centralized application. The most common object-oriented programming model is that of *threads and objects*, and this is what most distributed object systems suggest to the programmer. There are however other possible programming models, often specific for distributed applications. An example for this are *mobile agents*, which are technically realized via remote invocation combined with object migration; yet they follow a different programming model that is distribution-specific.

In the following, we will describe the evolutionary progress in the development of distributed object systems. To this end, we have selected a number of systems from the literature which we consider important and characteristic stages in this development. This choice is necessarily subjective and could well be argued about in detail. We are convinced, however, that the general picture portrayed by this choice is an adequate one.

We will group systems by their most characteristic feature, the type of distribution abstraction they use. Thus, we will first look at the explicit approach in section 2.1, then at the implicit approach in section 2.2.

## 2.1 The Explicit Approach

An *explicit* distributed object system is a system where remote communication is expressed within the programming language itself. There is an inherent tension in this approach because at the same time, the goal of distribution abstraction is to *hide* the complexities of remote communication from the programmer. Explicit systems do achieve this goal by using traditional object-oriented programming techniques, such as inheritance and the separation of interface and implementation. When applied consequently, this can in fact mean that distribution is *invisible* to the application programmer, although behind the scenes, it may be expressed using the same language. On the other hand, the explicit approach does allow *varying degrees* of distribution transparency, depending on how the abstraction is actually realized.

Historically, explicit systems have been using *remote invocation* as their dominant communication mechanism, although other mechanisms can be implemented explicitly as well. The key idea of remote invocation is to use the same mechanism for remote communication between objects as for local communication: the invocation of methods. In this, remote invocation systems follow the idea of the Remote Procedure Call (RPC), which was introduced as a means to facilitate communication in module-oriented, procedural programs (White 1976, Birrell and Nelson 1984).

In a Remote Procedure Call, the remote target procedure is not called directly, but the call goes to a "dummy procedure" on the caller's machine. This dummy procedure (also called a *client stub*) sends a network message to the callee's machine, including the name of the procedure to be invoked and the actual parameters of the call. (Transporting the parameters via a network message is also called *marshalling* them.) On the callee's machine, the network message is received by an adapter (the *server stub*) which *unmarshals* the parameters and performs the actual call. After that, results are returned to the caller using the same mechanism in reverse.

Remote Object Invocation (ROI) is the object-oriented extension of RPC. Compared to traditional RPC, there are two additional issues involved:

- Stubs are created at the object level, not for individual methods. With ROI, a remote object is therefore realized as a *stub object* or *proxy object* on the client side, and a *skeleton object* on the server side. These objects perform, for each individual method, the marshalling and unmarshalling of parameters.

- Since objects have an identity, they are usually accessed by reference in object-oriented languages. As references are usually implemented as pointers (direct memory addresses) within a program's address space, and pointers cannot readily be used across distribution boundaries, *network references* must be introduced. A network reference is a "pointer" to an object which, in addition to the object's memory address, includes the information at which machine the object physically resides.

Remote invocation systems differ in how similar the semantics of a remote invocation are to a local invocation, and how much work is required by the programmer to actually make an object remotely invokable. A remote invocation mechanism alone, however, is not sufficient for building distributed programs. Objects that should reside on separate nodes somehow need to *get* to these nodes in the first place, and, having been placed onto the desired nodes, the objects need to make initial contact with each other.

There are two general approaches to this:

- One may concentrate on the task of *binding to an already existing instance* on another host. This can be realized by publishing the instance's network reference to an external facility, usually called a *name server*, from which a client may receive the reference if it knows the name under which it is stored. With this approach, a distributed application consists of a set of stand-alone programs that are started independently of each other.

- A more integrated solution is *remote object creation:* Here, the statement that creates an object is annotated with a specification of which machine the object should be created on. After the remote creation has taken place, the creating object immediately receives a network reference to the remote object, and can henceforth communicate with it. For this to work, it is clear however that the participating machines or run-time systems need to establish their initial contact by some other means on a lower level. (Often a name server is used at this level instead.)

Remote invocation and remote object creation, by themselves, would only allow for static distributions of objects. Remote invocation systems therefore often provide some form of *object mobility* in addition. An object is said to *migrate* from one node to another if it is physically transferred to the new node, and can no longer be accessed at the previous location afterwards. Any references that point to the previous location are subsequently redirected, usually by a forwarding mechanism. Object migration can be *strong*, which means that an object can move even while some of its methods are executing. The execution is interrupted, and resumes at the destination node at exactly the same point. *Weak* migration, on the other hand, means that there must not be an active thread in an object when a migration is initiated.

Transferring an object to another node without invalidating the original object results in a *copy* being created. This is often used as a weak alternative to true object migration. Objects can also be *replicated*, meaning that an object physically resides at more than one node, with the replicas being held consistent in some fashion by the run-time system. In the special but common case where an object (or the state of a class) is *immutable*, there is no need for a consistency protocol and replication can be performed at no cost at all.

Object *caching* is a variant of replication where replicas are created on demand, usually for a short period of time. Object replication and object caching are less common in explicit systems based on remote invocation, but essential for implicit (DSM) systems. We will therefore discuss them more thoroughly in section 2.2.

The number of systems we chose to portray the explicit approach is fairly large, reflecting the dominance of this paradigm within both academia and industry. To structure the presentation, we therefore group these systems into three further categories. Section 2.1.1 describes *early systems*, which we define loosely as projects dating before the wide-spread adoption of the explicit approach in the form of *industry standards*, which are described in the following section (2.1.2). Finally, section 2.1.3 covers *current research platforms* from the late 1990s onwards. Although it has not been an intention in selecting them, one characteristic feature of them is that they are all Java-based.

### 2.1.1 Early Systems

The early explicit distributed object systems often involve a special programming language that was designed with easy distribution in mind. Some of these systems are module-based or object-based rather than object-oriented, but we still include them here because they develop important concepts that can likewise be found in later, object-oriented systems.

#### Emerald

*Emerald* (Jul et al. 1988) is a distributed object system that was developed at the University of Washington, comprising the Emerald programming language and compiler, and a corresponding distribution platform implemented on VAX and Sun 3 computers. Although being dated as early as the mid-1980s, the Emerald system implements the remote invocation model more thoroughly, cleanly, and elegantly than most later projects. This is reminiscent of the conceptual clarity of early object-oriented languages such as Simula-67 and Smalltalk, which are still not matched by many of the more recent languages such as C++ and Java.

The Emerald language is an object-based programming language with neither classes nor inheritance. It was specifically designed so that the language semantics allow for a distributed implementation.

An Emerald object consists of four components (Jul et al. (1988), p.112):

- A unique network-wide name;
- A representation, that is, the data local to the object, which consists of primitive data and references to other objects;
- A set of operations that can be invoked on the object;
- An optional process.

The remote invocation model is realized to such a degree in Emerald that it is completely invisible to the programmer: any object can potentially be invoked remotely, with identical semantics as in a local invocation. In particular, the local parameter passing semantics are retained for remote invocations. To achieve this, any object reference that appears as an argument in a remote invocation is converted into a network reference.

Emerald objects are *mobile* in the strong sense: the system keeps track of any active invocations of an object; when an object is moved, it is ensured that all activation records within the system are updated correctly.

The programmer can control the locations of objects using a number of primitives provided by the language:

- "**locate** X" – return the node where object X resides
- "**move** X **to** Y" – colocate X with Y
- "**fix** X **at** Y" – fix X at a particular node
- "**unfix** X" – make X mobile again after a fix
- "**refix** X **at** Z" – atomically perform an unfix, move, and fix at a new node

A common problem with mobile objects is that it often does not make sense to move a single object alone. There are often internal objects on which an outer object critically depends, and these should be moved along with the containing object. To allow this, Emerald provides an "**attach**" modifier for instance variables. Any objects stored in variables marked with this modifier are moved along with the containing object. (This is a transitive but not a symmetric relation: an internal object itself may be moved explicitly, and no attempt will be made to move the containing object with it.)

In addition to these primitives, Emerald allows parameter objects to be passed *by move* or *by visit*. These passing modes are semantically equivalent to pass-by-reference, but, as their names imply, *pass-by-move* lets the parameter object be moved to the site of the callee, while *pass-by-visit* means that additionally, the object is moved *back* to the caller when the invocation returns. These passing modes, which are specified by annotating the formal parameter declarations of a method, allow for a more structured control of object location, as compared to using the other mobility primitives directly.

There are no further distribution-related constructs. Internally, there are three different categories of Emerald objects:

- *Global objects* are objects that can potentially be invoked remotely.
- *Local objects* are objects for which the compiler can infer that they will never be invoked remotely.
- *Direct objects* are objects, the data area of which is physically embedded into a containing object. For example, primitive data values (which are nonetheless considered objects in Emerald) fall into this category.

The implementation of Emerald seeks to minimize the performance impact of distribution on local operations. The reasoning behind this is that distributed operations, such as a remote method call, are orders of magnitude slower than corresponding local operations. Therefore, the authors argue, there is likely no noticeable impact if the duration of distributed operations is increased by a few CPU cycles, while these few CPU cycles do matter tremendously when spent in the local case.

For this reason, Emerald uses *direct memory addresses* within objects, which means that local operations incur zero overhead. However, when objects are passed across distribution boundaries, these direct addresses are translated on the fly to substitute them with references to proxy objects within the caller's address space.

In a case study performed by the authors, an electronic mail system was implemented in Emerald, where individual mail messages are represented as objects. The authors were able to make these messages mobile, and hence let them move from the mail sender to the mail receiver, simply by adding the word "**move**" in two places of the program, which resulted in a performance gain of 22% for a simulated workload of mail traffic.

### Distributed Smalltalk

Distributed Smalltalk (Bennett 1987) upgrades the traditional Smalltalk environment (Goldberg and Robson 1983) to a distributed object system.

One characteristic feature of Smalltalk is that there is no distinction between an application and the programming environment — the programmer modifies the programming environment so that it *becomes* the application. This in turn means that every Smalltalk "application", along with the programming environment, has the property of being *reactive*: every object within the system can be presented to the user for inspection and modification. "Every object" includes ordinary instances, but also classes ("objects that describe behavior", in Smalltalk terminology), and even low-level implementation objects such as stack frames.

The task of turning Smalltalk into a distributed object system is therefore different and more difficult than with most other object-oriented languages. Keeping with Smalltalk's philosophy, it means to convert a single-user, single-address space programming environment into a multi-user programming environment where every aspect of the system is always ready for interactive inspection and instant modification.

In particular, this means that compatibility between classes on different machines becomes an immediate and urgent problem. Several possible solutions are discussed in Bennett (1987), and the one finally implemented in Distributed Smalltalk is a conservative one. It requires that classes and instances be *co-resident*: each object must have its implementation locally available, which in turn means that object mobility within the system is restricted. For system classes, the issue is mitigated because system classes are replicated on each machine, and to move an object of a system class only involves a superficial compatibility check (since the system class could have been modified locally on one of the hosts nonetheless). User-defined classes, on the other hand, are not replicated, and if an object of a user-defined class is to be moved, its class (and possibly the classes from which it inherits) is *moved* to the destination as well (otherwise, for example the values of class variables would no longer be unique per class).

The remote invocation mechanism of Distributed Smalltalk exploits the fact that Smalltalk is a weakly-typed language: there is only a single class of proxy objects, which redefines the **doesNotUnderstand:** method so that it forwards the actual message selector and its arguments to the remote system. This works well for all methods that are called via the standard lookup mechanism of Smalltalk, but it does not work for some basic "no-lookup" primitives such as the "==" operator. Distributed Smalltalk solves this problem by rewriting the source code so that, for example, "==" is replaced with **EqualEqual**, an equivalent method that falls under the standard lookup mechanism and is provided by Distributed Smalltalk.

Other areas that are given special attention in Distributed Smalltalk include:

- *Garbage Collection.* Distributed Smalltalk lets each host use its own local garbage collector, and *prevents* objects from being reclaimed if they are remotely, but not locally referenced. This is realized by a *RemoteObjectTable* object on each host, which contains a local reference to each object that is known to another host. Two separate algorithms are then employed to turn distributed garbage into local garbage so that it can be reclaimed: one algorithm is very fast but cannot detect cycles of garbage that span multiple hosts, while the other can detect such cycles but is comparatively slow. For details, see Bennett (1987).

- *Error Reporting and Remote Debugging.* The error reporting facilities of Smalltalk have been thoroughly adapted to the distributed case. If an error is signalled in a remotely invoked object, the error is reported back to the host that issued the invocation. Here, a modified debugger is activated that allows the programmer to inspect the stack frames of both the local and the remote process.

- *Node Autonomy.* Distributed Smalltalk allows the "owner" of a workstation to exercise control over what objects may or may not be remotely accessed. For example, the *display bitmap* is an object that should not normally be accessible to outside users, or at least only in a controlled manner. For the latter case, the system allows the user to specify a surrogate object (termed an "agent") to which all remote invocations are delegated. The surrogate object may then decide which invocations should be forwarded to the real object, and if so, how. An example for this would be to restrict remote users so that they have only access to a reserved area on the local machine's display bitmap.

The author reports performance figures that are typical for remote invocation systems: a remote invocation in Distributed Smalltalk takes about three orders of magnitude longer than a local invocation. Of this additional time, only one fourth was attributable to actual network communication.

## DOWL

DOWL (Achauer 1993) is an object-oriented programming language and run-time environment that is based on similar ideas as Emerald, however it extends them in a number of ways. DOWL is a distributed extension of the Trellis language and its corresponding environment (O'Brien et al. 1987). Trellis, by itself, is a strongly typed, object-oriented language with multiple inheritance. There is a "multiple-activity" system that enables concurrent programming. Similar to Smalltalk, Trellis features its own programming environment that is itself written in Trellis, comprising a window system and an incremental compiler.

Similar to Emerald, DOWL allows for fully transparent remote invocation of objects, the semantics being identical to the local case. Any object, except for instances of some built-in types, can become remotely invokable; this happens dynamically at run-time (via substitution of proxies). No changes in the source code are necessary.

While (almost) any object can be invoked remotely, types can also be marked as *replicated*, which means that instances of such a type are *copied* to other address spaces. Hence, each address space will have its own local replica of such an object. (There is however no consistency protocol in DOWL. If centralized semantics are to be preserved, then replication must therefore only be used for immutable objects.)

As in Emerald, DOWL offers constructs to control object location and mobility. Unlike Emerald, however, these constructs are mostly not primitives, but integrated into the object model. This means that, for example, it is possible to change a type's migration characteristics by overriding an inherited operation.

Nodes are represented as objects in DOWL. Furthermore, each object has an instance variable named **$location** (instance variables are named *components* in DOWL terminology, but we use the more common object-oriented term to avoid confusion). Assigning a **$Node** object to an object's **$location** variable causes the object to migrate to that particular node (this is *strong* migration; it can also happen while an operation on the object is active). This way, it is also easy to co-locate two objects, by assigning the value of one **$location** variable to another.

Migration can be inhibited by *fixing* an object to a particular node. This can be done at the type level, using the attributes **$fixed** and **$mobile** (which means the opposite), or for an individual object using the instance variable **$fixed_at**. Assigning a node object to this variable has a similar meaning as moving the object via **$location**, but the latter will be ignored if the object is already fixed, while it causes an exception with the former. Fixing an individual object with this mechanism is temporary (assigning **Nil** unfixes it again), while the type-level attributes cause all objects of a type to be permanently fixed or mobile.

Objects can be grouped together using an **$attach** attribute for instance variables, as in Emerald. Two further attributes, **$move** and **$visit**, can be used to achieve pass-by-move and pass-by-visit parameters, also as in Emerald. However, these attributes are more general than that: unlike in Emerald, **$move** and **$visit** can be applied to *any* variable, not just to formal parameters. For example, when an instance variable is marked as **$move**, then any object assigned to that variable is moved to the variable's containing object along with the assignment. (The author points out that **$move** and **$visit** are actually "syntactic sugar", as their effect could be achieved with primitive migration operations as well. However, they allow for a more *structured* way to control object mobility within a program.)

Several non-trivial applications were implemented in DOWL, or upgraded to DOWL from Trellis. The largest and most challenging of these was the Trellis developing environment itself, comprising about 65,000 lines of code. The only problem that had to be solved in order to turn it into a distributed application was to handle predefined types and system facilities, such as the window system and file I/O, correctly. This was basically achieved by marking some types as *replicated* or *fixed*. Altogether, only about 100 lines of code had to be changed in order to make all editors, browsers, inspectors, etc. operate on remote objects instead of local ones.

In summary, DOWL appears as a natural, evolutionary advance over the Emerald language, the ideas of which underlie both systems. DOWL achieves a better integration of mobility constructs with the object model, and enhances location control toward a level that would be required for the construction of large distributed systems.

## HERON

Heron (Finke et al. 1993, Wolff and Löhr 1996) is an object-oriented distribution platform based on the Eiffel language. Despite its focus on Eiffel, Heron has been designed with heterogeneity in mind, and has in fact been demonstrated to interoperate with Pascal programs. Heron follows the remote invocation model, paying particular attention to a number of issues that are hardly dealt with elsewhere in the literature.

To begin with, Heron insists on complete distribution transparency, allowing no distribution-related changes or constructs in the source code at all. Instead, the process of distributing a program is performed entirely by an external configuration utility and a proxy generator. In this, Heron goes even further than, for example, Emerald or DOWL, which do have some distribution-related primitives. The reason we still consider Heron as an *explicit* platform (where distribution is handled within the programming language) is that the configuration description still needs to be written by the programmer; it has only been moved outside of the program for separation of concerns. Since Heron uses an existing language, Eiffel, which was developed without distribution in mind, the goal of complete transparency could not fully be achieved (see Wolff and Löhr (1996) for a complete discussion of transparency limitations).

Unlike many other distributed object systems, Heron features *distributed objects* in the sense that the implementation of a single object can be distributed (spread out) over multiple machines. (An object that only resides on one machine is, strictly speaking, not distributed.) There are two cases of this:

- Heron allows *remote inheritance*, which means that parts of an object that belong to different classes can reside on different machines. This feature is particularly necessary because of Eiffel's peculiar way to use inheritance where in other object-oriented languages, aggregation or delegation would be used. For example, in Finke et al. (1993), a *Client* class inherits from a *Servers* class in order to obtain remote references to the server objects.

- A *dispersed object* is an object that appears to the outside world as a single object, but which is internally realized as multiple instances *of the same class* that reside on different hosts. An example for this is a *Set* object which is made up of several sub-*Set* objects which are not visible to the user.

To configure a program for distributed execution means to split it into *components*, which are defined in Heron's configuration language. Statically, a component is a subset of the classes that make up the application, plus stub and skeleton classes needed for interfacing with other components. Technically, a component is realized as an Eiffel program that can be started on a given host and which interacts with other components on other hosts. There may be multiple instances of any given component at run-time. To distinguish this dynamic aspect of a component from the static aspect referred to before, a run-time instance of a component is also called a *compostance*, while the static "bundle" of classes is called a *compilent*. Within each compostance, objects of the classes that the compilent contains may be created.

A component can either be instantiated (loaded) when it is first accessed by another component, or it can be declared as a *server* component, which means that it is started once and then shared by all other components. This way, public servers can be realized.

Within the configuration description, a component $C$ may specify that objects of a class $s$ should be created in another component $D$. This way, HERON realizes remote object creation without actually modifying the creation statements in the source code.

When referring to another component in the configuration file, this does not actually involve the host or filename from which the other component will be loaded at run-time. These references are resolved in a separate step by the HERON linker.

### Equanimity

*Equanimity* (Herrin and Finkel 1991, 1993) is a software package that can split centralized programs into client and server parts, where the actual distribution boundary between client and server is not fixed, but dynamically adjusted at run-time to improve performance. Equanimity is module-based rather than object-oriented, but deserves to be mentioned here because it already deals with many of the issues found in distributed object systems. Unlike most of the other systems we are discussing, particularly the early ones, Equanimity does not use a special programming language; it can, at least theoretically, be used to distribute existing programs.

The package consists of a set of tools built upon and extending standard UNIX facilities, such as the dynamic linker *dld*, and *Sun RPC*. The unit of distribution is a *module*, which the authors define as *"a self-contained collection of subroutines and their associated data"*. Technically, an Equanimity module is a UNIX compilation unit represented as an object file.

Modules are transformed into remotely callable units using the standard RPC facilities. Additionally, modules can migrate between client and server; this is hidden from callers by a "local/remote switch" (*LR switch*), which is a software wrapper that decides whether a subroutine is locally available or must be called remotely (see fig. 2.2). The LR switch also gathers statistics about calls to the module, and may initiate module migration if it seems likely to improve the overall application performance.

The decision when to migrate can be based on varying strategies, such as

**network-load,** which minimizes the number of bytes transferred over the network by observing calls to and from each module, and migrating the module if the number of bytes transferred remotely exceeds that of locally transferred bytes by a certain threshold,

**network-load, call-frequency normalized,** which extends the above by taking into account the actual number of calls between each module (many calls with few parameters are more expensive than few calls with many parameters because of the fixed cost of each call),

**CPU-load,** which seeks to migrate modules to the CPU with the most power to run them; this is based on the speed of each CPU and its currently measured load, and a programmer-provided estimate of the CPU time needed to run each subroutine of a module,

**hybrid strategies** consisting of weighted combinations of the above.

Module migration is *weak*: the LR switch counts how many times a module has been entered and exited by threads of control; a module can only migrate if no thread currently occupies it.

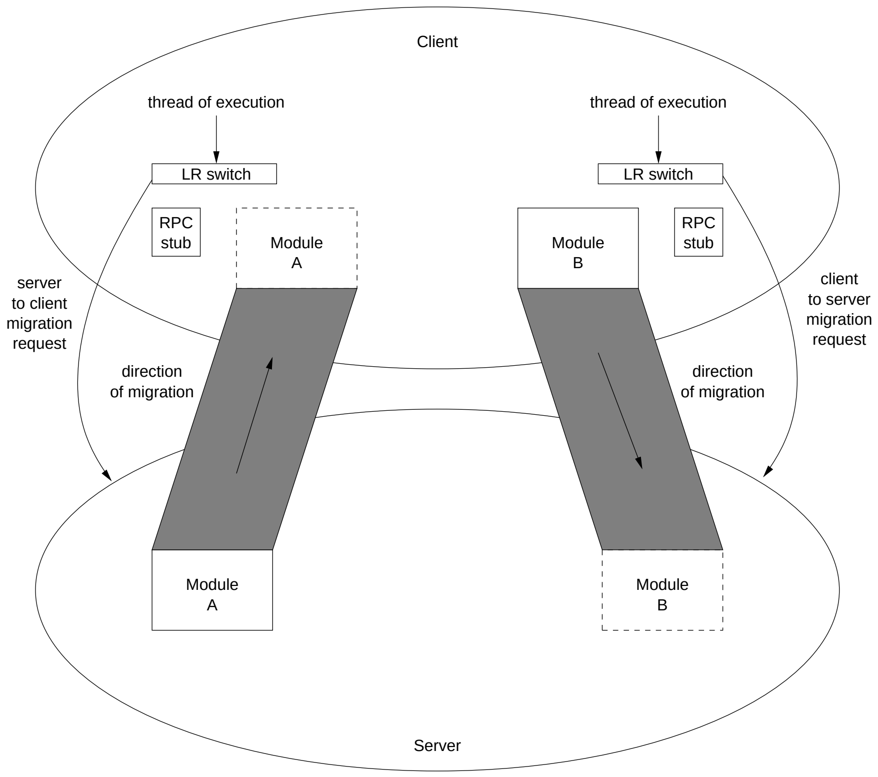


Figure 2.2: The Equanimity System

There are facilities for migration between heterogeneous architectures (essentially realized by keeping binaries for all architectures around, and using a machine-independent data transmission format). It is also possible to use different *versions* of a module at the same time.

Equanimity only splits a centralized program in two halves with a dynamic boundary; it does not convert it into a multi-threaded client/server application where many clients access a single server. If several clients are launched, a separate server is spawned for each of them.

Generation of RPC stubs and LR switches is semi-automatic; the authors estimate that a fully automatic stub generator is feasible, but would be as complex as a compiler. Also, Equanimity cannot convert all kinds of subroutines into remotely callable ones: the only allowed parameter mechanisms are pass-by-value and pass-by-value/result; addresses (pointers), and heap-allocated data structures cannot be used as parameters.

Thus, Equanimity will *in practice* not likely be able to distribute an arbitrary existing program. The programmer is restricted to a very small set of language constructs for any remotely callable modules, and hence, distribution must be designed-in from the initial phases of program development.

The authors present several case studies. One of them is an Equanimity wrapper around an X window system server (see fig. 2.3). A client program that is linked to this wrapper can migrate parts of its code to the server machine, so that graphics-intensive operations are executed locally on the server.

A recurring pattern of all the case studies is that they have strictly hierarchical, layered architecures, where the *fan-out* of each module is higher than its *fan-in* (one call to a subroutine triggers several calls to modules of lower layers). At least under the network-load strategy, the optimal distribution boundary for such a program is clear: all modules, except for the root module at the top of the hierarchy, must be migrated to the server, and this is precisely what Equanimity does.

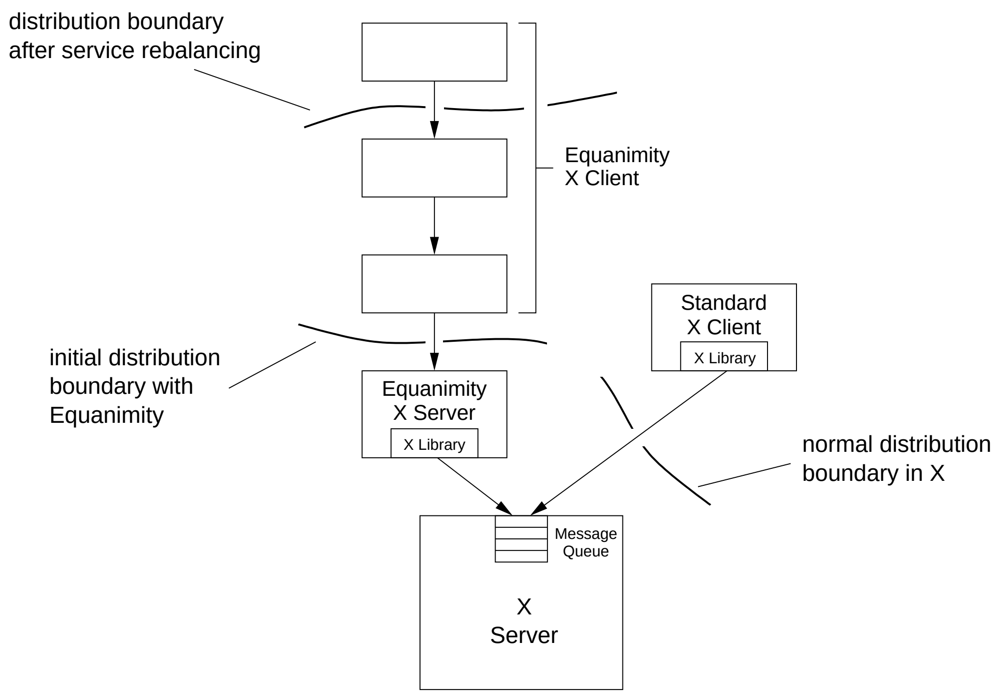


Figure 2.3: An Equanimity X Server

### 2.1.2 Industry Standards

Industry standards of the explicit approach to distributed object systems have been defined from the early 1990s onwards. In this section, we will cover the main three ones of them, Java/RMI, CORBA, and C#.

#### Java/RMI

Java/RMI (Wollrath et al. 1996, Sun Microsystems 1997) is Sun's native mechanism for remote method invocation on the Java platform. It is thus an industry standard, and also the basis of almost all of the more experimental Java distribution platforms discussed further below.

To make a class remotely invokable in Java/RMI, the methods of the class need to be declared in an interface which extends *java.rmi.Remote*. On the client side, the remote object is accessed through this interface; this allows the system to actually delegate the call to a stub (see fig. 2.4). The stub, together with the skeleton on the server side, is automatically generated by the RMI compiler. The compiler further requires that each method of the remote interface be declared to throw *java.rmi.RemoteException*. If a communication error occurs in a remote call to that method, this exception (a subclass instance, actually) is thrown to the client. Since *RemoteException* is a checked exception, the calling method in the client must provide a handler for it, or explicitly propagate it further up to its own callers. On the server side, the actual implementation object needs to be defined as an extension of *java.rmi.server.UnicastRemoteObject*, or an equivalent class, which "provides the framework to support remote reference semantics."[1]

In order to make initial contact between clients and servers, a simple bootstrap naming server, the RMI "registry," is provided. There is no built-in construct for remote object creation. Two other important features of Java/RMI are distributed garbage collection and code downloading, which allows clients and servers to interoperate even if they do not share a file system, or have

[1]Java/RMI documentation of *java.rmi.server.RemoteServer*, the superclass of *UnicastRemoteObject*.

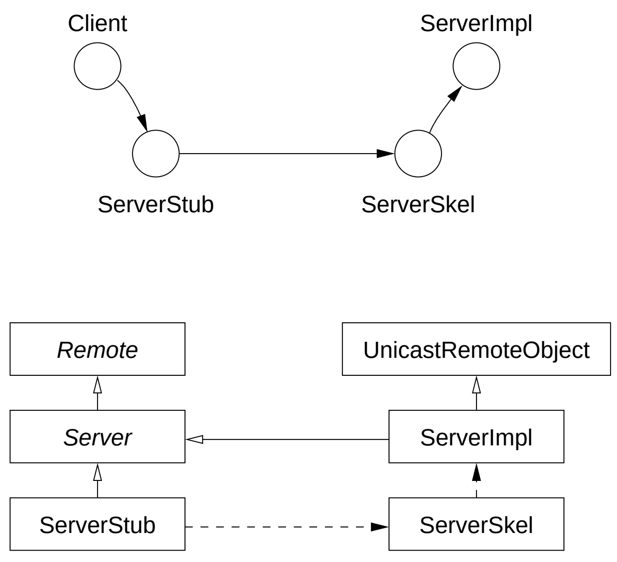


Figure 2.4: Remote Method Invocation in Java/RMI

local access to the same codebase.

The parameter passing modes of remotely invoked methods are slightly different from standard Java semantics. In the local case, objects are always passed by reference. In the remote case, an object is only passed by reference if its class is itself a remotely invokable class. If it is not, then the class must be serializable, and the object is passed *by value* through serialization. Objects of types that are neither remote nor serializable cannot be passed in remote invocations (an exception is thrown in this case).

This diversion from standard Java semantics is motivated by performance reasons. It is clear that pass-by-reference can only be realized remotely via *network references*, and that the parameter object must itself be remotely invokable so that the callee, after having received the network reference, can actually access it (see fig. 2.5). This, however, requires further remote invocations back from the callee to the caller's host. For performance, it would often be better if the parameter object could actually be *moved* to the callee. In the absence of true object migration (and language constructs such as pass-by-move in Emerald), a kind of second-best guess is to pass the object *by value*, so that a copy is created at the callee's site, which can then be accessed locally (see fig. 2.5). For a detailed analysis of the performance issues, see Spiegel (1998).

The price to be paid for this weak replacement of true object migration is the different semantics of remote invocations. This is particularly odd as one of the goals of the RMI mechanism is to make remote calls transparent, in the sense that they are syntactically indistinguishable from local calls. And since Java/RMI does indeed succeed at that, the difference in semantics becomes a particularly mean trap for the programmer. As it turns out (Brose et al. 1997), it is possible to construct situations in which the actual semantics of a given call is totally unpredictable, because it cannot be foreseen whether the object that a given reference points to is local or remote.

As discussed in Spiegel (1998), passing objects by value remains an important opportunity for optimization in distributed programs, even if a platform does also support true object migration. There are situations where it is semantically safe to use it (e.g. for immutable objects), and the Pangaea system does indeed exploit such opportunities.

From what has been said, it becomes clear that Java/RMI requires a lot of explicitly distribution-related programming. For one thing, the use of specific RMI-related interfaces

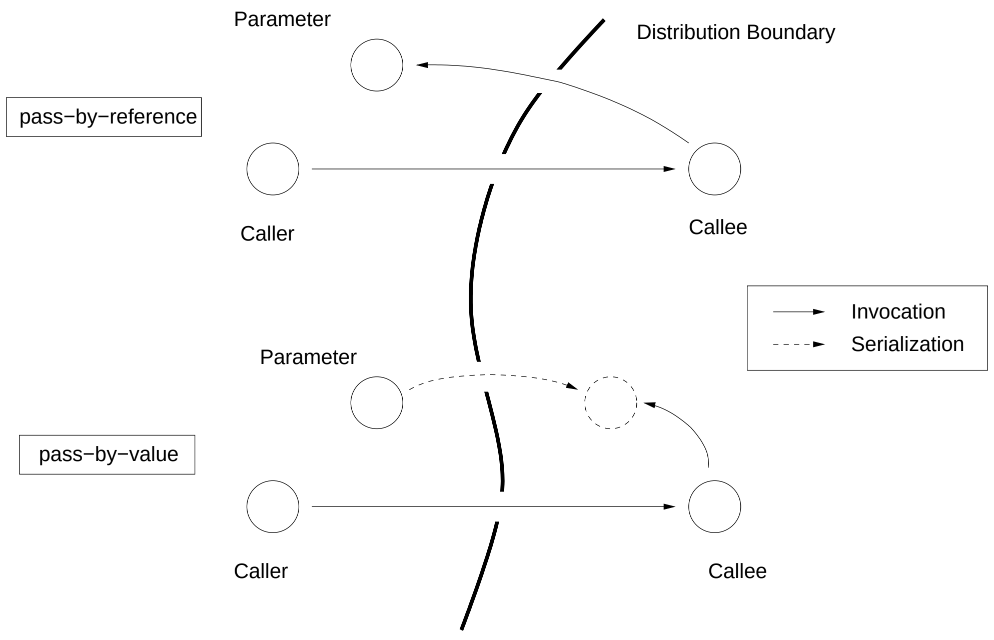


Figure 2.5: Remote passing mechanisms

and superclasses means that the inheritance hierarchy of remotely invokable objects is dictated by the RMI architecture, and not by the application. Second, the fact that every remote call might fail with the checked exception *java.rmi.RemoteException* means that this exception must always be handled explicitly on the client-side — even if all the application can do in response is to exit cleanly, and require human intervention. The result is that the code on the client-side gets drastically longer, due to the presence of exception handlers, and lacks clarity.

This is, however, in part, no accident. The designers of Java/RMI have stated explicitly (Waldo et al. 1994) that they consider distribution transparency a bad idea, and that a programmer should always be aware of whether a given method call will be local or remote. We will discuss this claim in detail in section 2.3. For the moment, suffice it to say that Java/RMI does indeed make distribution issues, and remote calls in particular, painfully obvious in the program code. It could only be argued that it does not go far enough in this regard: almost the only thing that still does not differentiate local calls from remote ones is the actual syntax of the call.

The decidedly non-transparent nature of Java/RMI has motivated several researchers to build distributed Java systems with a higher degree of transparency, among them JavaParty, Manta, and Doorastha, which we will discuss below. These platforms often use Java/RMI as the basic communication mechanism behind the scenes, reaping the benefits of distributed garbage collection, code downloading and remote exception handling. They do attempt, however, to make it much less apparent in the source code whether an object is remote or not.

## CORBA

CORBA (the "Common Object Request Broker Architecture") is a widely used industry standard for distributed applications. The CORBA standard (OMG 2000) is defined by the OMG, an open-membership, not-for-profit consortium of major companies in the computer industry. Several implementations of the CORBA standard exist and are in wide use today, such as *Orbix*[2],

[2]`www.orbix.com`

*Visibroker*[3], *JacORB*[4] (Brose 1997), and *MICO*[5] (Puder and Römer 2000).

CORBA is an interoperability layer between different hardware platforms, operating systems, applications, and programming languages. The standard defines an object-oriented model of the entities in such settings, and provides the infrastructure for them to communicate, following the remote invocation model. A CORBA application thus consists of a number of CORBA objects (which can be anything from entire applications to fine-grained objects at the programming language level); these objects communicate via an "Object Request Broker" (ORB), which is an infrastructure layer typically implemented as a user-space library (see fig. 2.6).

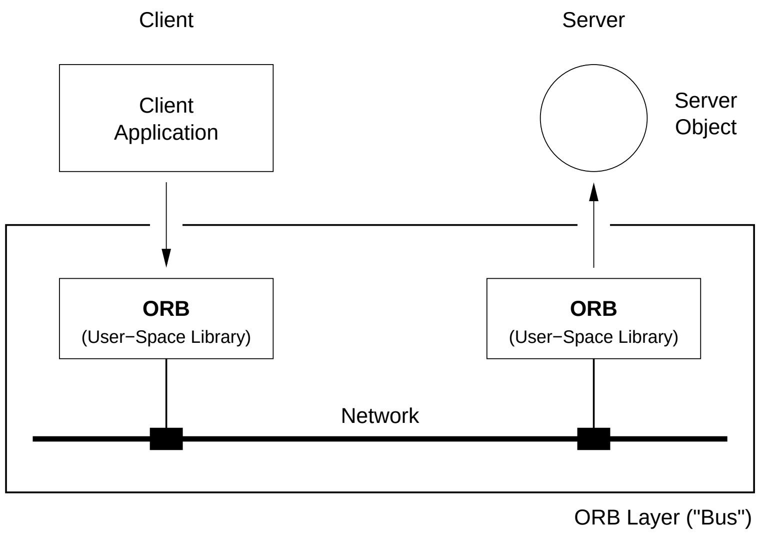


Figure 2.6: Client, Server, and the ORB Layer

Because CORBA is language-neutral, it defines its own *interface definition language* (IDL). To create a CORBA object (i.e. one that is accesible via the ORB infrastructure), the programmer describes the object's interface in CORBA IDL and provides an implementation of that interface in an arbitrary programming language. The IDL description is then used to generate appropriate stubs and skeletons (see fig. 2.7).

Although not a full programming language, CORBA IDL is an object-oriented language, the syntax and object model of which are loosely based on C++. Its core constructs are:

- *Interface types*, which define the operations and publically visible attributes of remotely invokable objects (for attributes, accessor operations are generated, so attributes are actually just a shorthand notation). There is also a special kind of "oneway" operations, which are executed asynchronously at run-time. Interface types can be related by multiple inheritance.

- *Value types*, which define objects that can be passed to other CORBA objects by value. The reason for providing this construct is the same as with Java/RMI: it is a simple substitute for true object migration (see the previous section for a discussion of this). CORBA value types are slightly more general than Java/RMI's mechanism, though: Similar to "expanded" types in Eiffel, it is possible for a value type to inherit from an interface type and vice versa. In a given method call, the actual parameter passing mechanism is then

[3]`www.borland.com/visibroker`
[4]`www.jacorb.org`
[5]`www.mico.org`

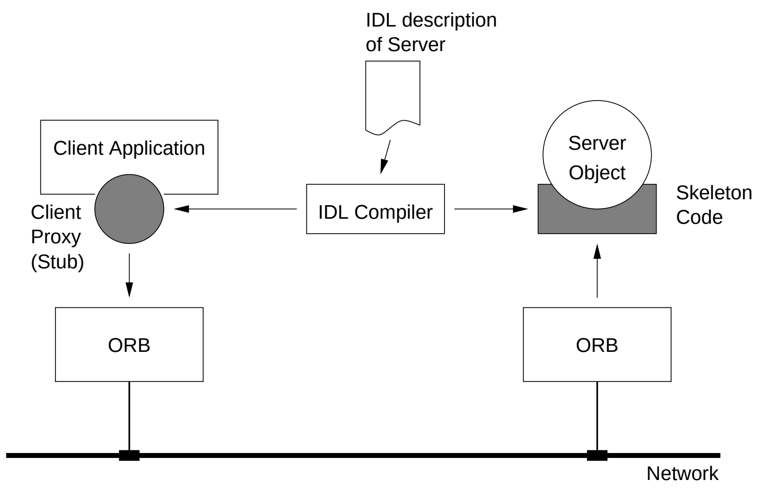


Figure 2.7: Generating Stub and Skeleton from IDL

determined by the types of the formal and the actual parameter. For example, if an actual parameter of a value type is passed to an operation that expects an interface supertype of it, then the parameter is passed by reference instead of by value. In effect, this means that passing mechanisms can be decided *per operation* instead of *per type*, as under Java/RMI.

The correspondence of IDL constructs to the constructs of a given implementation language is described in formal "language mappings." Such mappings exist for all major programming languages, including C, C++, COBOL, Ada, Smalltalk, and Java. It is a natural consequence of this generality that no single programming language can be mapped entirely seamless to CORBA IDL. This is especially true for non-object-oriented languages like C and COBOL, where the object-oriented structure of IDL needs to be simulated by conventional language constructs. But even for languages such as Java, which are conceptually very close to IDL, many disparities arise (Brose 1998). The more important ones of these are:

- Strings and arrays are base types in IDL, but object types in Java. One consequence of this is that null references to strings and arrays cannot be expressed in IDL, or passed in remote invocations mediated by the ORB.
- IDL provides three parameter passing modes *in*, *out*, and *in out*, the latter two of which have no correspondence in Java. As a result, parameters with these modes need to be wrapped into special "holder objects" on the Java side.
- Method names are case sensitive in Java and can be overloaded, both of which is not true in IDL.

Writing a distributed application using CORBA thus requires the programmer to adapt to the programming model imposed by CORBA IDL. A CORBA object will look very different from an ordinary Java object for local use, which in turn means that distribution must be designed into the application from the very beginning. This is further reinforced by the fact that the ORB requires various inheritance relations to be fulfilled by the application code (see fig. 2.8). Similar to Java/RMI, this means that the inheritance hierarchy suggested by the application logic will often be distorted (the *tie approach* illustrated in fig. 2.8 has been introduced to mitigate this).

CORBA does, however, go beyond being only a remote invocation facility. For one thing, it should be noted that CORBA objects as we described them so far are not simple remotely

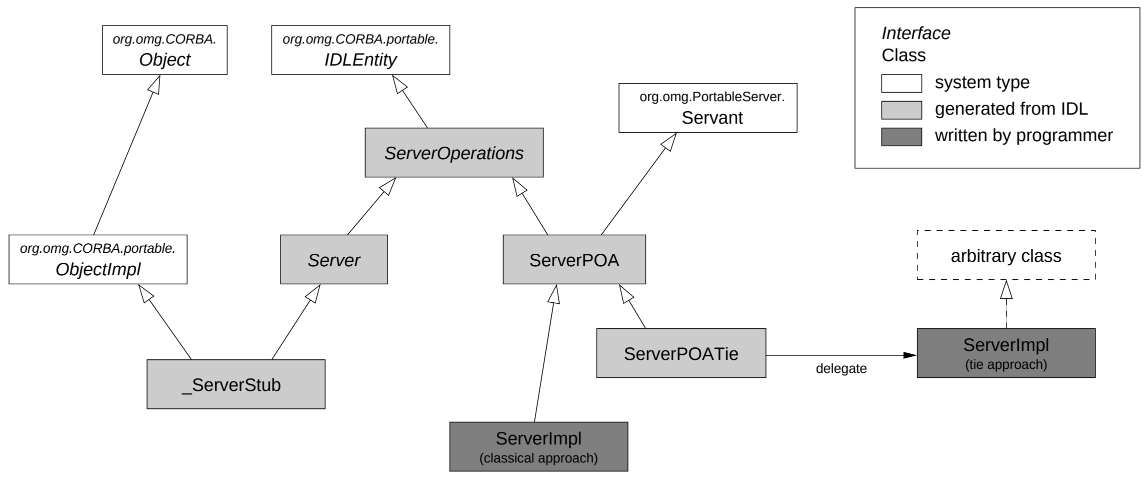


Figure 2.8: Inheritance Hierarchy for Interface Type *Server*

invokable instances: The Portable Object Adapter (POA) allows the programmer to specify policies how an object should behave when invoked by multiple clients. For example, a separate instance may be created for each client or a single instance might serve all requests; it may also be specified whether requests should be serialized or may be executed in parallel.

Additionally, CORBA defines an infrastructure framework for enterprise applications in the form of *services*. These include a *Naming Service*, *Trading Service*, *Transaction Service*, *Persistence Service*, *Security Service*, *Event Service*, and others. All of these services are provided in the form of APIs, generic client/server architectures, and programming conventions. This means that although, for example, event handling does transcend the traditional object-oriented programming model, the CORBA services do not affect the actual programming model of implementation languages. This is a pragmatic solution, the obvious drawback of which is that it leads to an awkward “simulation” of features such as event handling, which more advanced programming languages will provide as a first-class language construct.

In summary, CORBA is a pragmatic approach to providing a distribution infrastructure for enterprise applications, where the main focus is on interoperability between heterogeneous platforms and programming languages. As such, distribution transparency is only a minor objective of CORBA. Similar as with Java/RMI, pretty much the only thing that is indeed distribution-transparent is the actual invocation of a remote object. In practically all other areas, distribution-related issues are completely visible in the source code. A CORBA application must necessarily be completely different from a centralized program that performs the same task.

### C#

C# is Microsoft’s attempt at providing a next-generation object-oriented language for Internet applications. The language is similar in spirit to Java, but adds constructs such as enumeration types and value types, reference parameters, and variable-length argument lists (for a feature-by-feature comparison of C# and Java, see Obasanjo (2001)).

Similar to Java’s RMI middleware, C# also provides a native distribution mechanism that is commonly referred to as “remoting” (Rammer 2002). Like Java/RMI, it is an implementation of the remote invocation model, but it seeks to avoid some of the shortcomings of Java’s approach. In particular, the following similarities and differences can be noted (Löhr 2002):

- As in Java/RMI, classes of remotely invokable objects must inherit from a special supertype, *System.MarshalByRefObject*.

- In C#, however, an object and its proxy do not need to share a common interface. Proxies in C# are introduced transparently, and can be cast to the actual type of the object that they refer to.
- Additionally, C#'s proxies are not generated by a separate, compile-time utility such as `rmic` in Java/RMI, but created dynamically at run-time using class metadata.
- There are no checked exceptions such as RMI's *RemoteException* in C#; the programmer is therefore not forced to provide handlers for communications failures.
- C# programs do not need an external naming service (like the `rmiregistry`) to publish object references. Binding to existing instances can be accomplished by letting the local run-time system contact the remote run-time system, and there is also a remote object creation mechanism that works similarly.

Despite these improvements, C#'s "remoting" suffers from some of the same shortcomings as Java/RMI. Ironically, these shortcomings are manifest in the awkward name "remoting" itself, which perpetuates the misunderstanding that "remoteness" were a *property* of an object, while it is in fact a *relation* between two objects A and B. In other words, a "remote" object (even if it was "remoted" by .NET) is usually *local* to other objects at the same time.

The semantics of a remote invocation, however, are slightly different from a local one, similar to Java/RMI. If an object does not inherit from *MarshalByRefObject*, but is serializable, then it is passed by value instead of by reference. Additionally, C#'s new reference parameters, when used for serializable objects, are implemented as pass-by-value/result.

While these different semantics must not only be kept in mind by the programmer when dealing with "remoted" objects, it is also possible to construct cases where the programmer has no chance of knowing whether an object is *really* remote, and accessed via a proxy, or not. Invocation semantics can thus be *unpredictable* (Brose et al. 1997, Löhr 2002).

In summary, C#'s "remoting" appears as an incremental improvement over Java/RMI, but not a revolutionary or evolutionary one. Although it offers a higher degree of transparency, it suffers from some of the same principal shortcomings, and, attempting to become an industry standard, also provides only a conservative set of features, lacking, for example, object migration or replication, which are common on recent research platforms.

### 2.1.3 Current Research Platforms

We use the term *current research platforms* for explicit distribution platforms that were developed *after* the introduction of the two early industry standards, CORBA and Java/RMI. This research is at least partially motivated by the conservative nature of the standards, which these experimental systems seek to transcend.

#### JavaParty

Contrary to Java/RMI's approach, where the fact that a class is remotely invokable is almost deliberately underlined in the source code, JavaParty (Philippsen and Zenger 1997) sets out to make this as unintrusive as possible. A new class modifier, `remote`, is introduced into the language, and a modified Java compiler translates such classes into Java/RMI "remote" classes, while also adapting any calls from client code. As a result, making a class remotely invokable requires no effort within the source code at all, neither on the client side, nor on the server side (except for inserting the new modifier). To illustrate this, Philippsen and Zenger (1997) created distributed versions of the Salishan problems (Feo 1992) both in plain Java/RMI, and with the

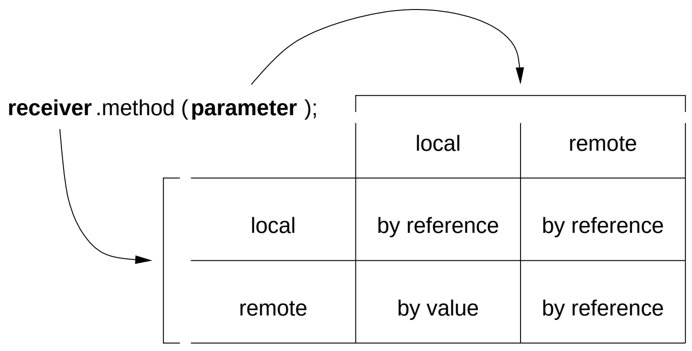
JavaParty compiler. To use Java/RMI, the source code had to grow by 66.2% with respect to a centralized version of the programs, while in JavaParty, the number of lines remained constant. (Actually, 2.2% of the programs' lines had to be slightly adapted, while under Java/RMI, 75.9% of all lines had to be added, changed, or deleted.)

JavaParty also allows more language constructs to be used remotely than Java/RMI does:

- Not only methods, but also *fields* may be accessed remotely; this is achieved by generating additional accessor methods and converting client-side accesses to method calls. (There are special methods to simulate constructs such as *o.i++*.)
- The *static* part of a "remote" class is kept as a single instance on one node of the system; it may also be accessed remotely. (Java/RMI does not allow remote access to the static part of a class.)

At run-time, the system is realized as a cluster of cooperating JVM processes, which must be started before any program can actually be executed. There is thus no need for explicit name server interaction in the program code. Whenever an object of a remote class is created, the run-time system decides on which JVM it should be allocated; this is configurable both externally, and via explicit calls to the run-time system from within the source code. JavaParty also allows *object migration* via a call to the run-time system. Although JavaParty's documentation is silent about this issue, we have found experimentally and by looking at JavaParty's source code that object migration is *weak*: it fails if there is an active method on an object.

Although JavaParty obviously offers a much higher degree of distribution transparency than Java/RMI does, it must be noted that the semantics of its "remote" types are quite different from standard Java. As implied by the underlying Java/RMI technology, the passing modes of parameters are affected by whether the receiving class and/or the parameter class are marked with the `remote` keyword (it is not relevant whether any of these objects is actually on a remote, i.e. distant node). The resulting parameter passing modes are the following:


**receiver** .method ( **parameter** );

| | local | remote |
|---|---|---|
| local | by reference | by reference |
| remote | by value | by reference |

An obvious way to restore standard Java semantics would be to make all classes of a program "remote". This is, however, impossible for two reasons: (a) system classes cannot be made "remote" because the source code is not available, and (b) the performance penalty would be prohibitive, since calls to "remote" classes are several orders of magnitude more expensive than local Java calls. The other option is for the programmer to be aware of the different semantics and adapt to it. This, however, means that it is not generally possible to distribute programs that were written without distribution in mind, at least not by simply adding the `remote` keyword to a few classes. Also, some consequences of the JavaParty semantics are very difficult to program with. One of them is that it is impossible to pass local references to "remote" objects at all, i.e. it is impossible to pass an array reference or a reference to a system object to a remotely invokable object, not even from the local host (fig. 2.9).

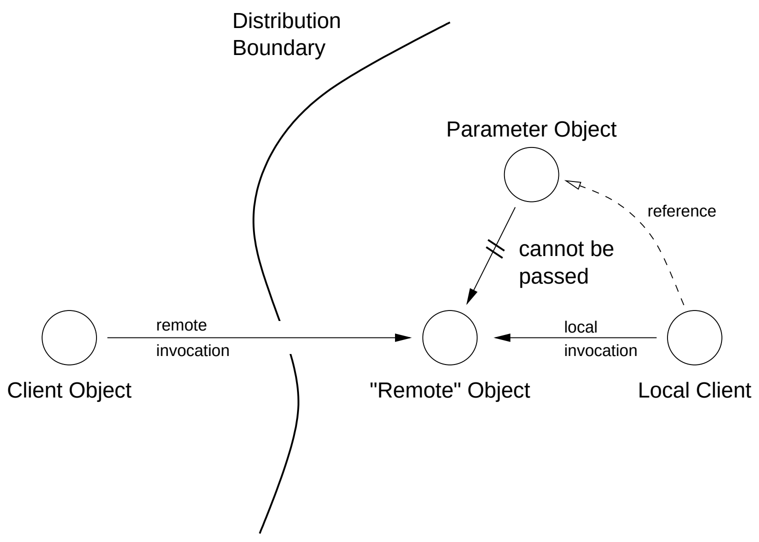


Figure 2.9: JavaParty: Cannot pass local objects to remote objects by reference

Two additional projects related to JavaParty that should be mentioned are an optimized re-implementation of RMI's serialization mechanism, named KaRMI (Philippsen et al. 2000), and an effort to use static analysis to automate object placement decisions (Philippsen and Haumacher 2000). We will return to KaRMI in our case studies in chapter 4, and the static analysis approach when discussing our own algorithm in section 3.1.

## Manta

*Manta* (Maassen et al. 2001) is a platform that follows the same programming model as JavaParty does, by introducing a `remote` modifier for Java classes. The semantics of it are only slightly different from those of JavaParty. In particular,

- remote field access is not possible under Manta; it is caught as a compile-time error, and
- within "remote" classes, calls to *this* object, as in

  ```
  remote class A {
    void methodA (myObject obj) { ... }
    void methodB (myObject obj) {
      this.methodA (obj);
    }
  }
  ```

  are treated as *local calls* under Manta, whereas JavaParty translates them into remote calls via the RMI protocol stack. The Manta approach is faster of course, although it could be argued that it is better to maintain semantic consistency as in JavaParty.

Unlike JavaParty, Manta is a native compiler. It targets x86 architectures, and is mainly used on the Distributed ASCI Supercomputer (DAS-2), located at the Vrije Universiteit Amsterdam and three other Dutch universities. The local clusters of this wide-area system consist of up to 72 Pentium III PC boards, connected via high-speed backbones (Myrinet and Fast Ethernet). The underlying communications technology is the Panda layer that was developed in the Orca project (see page 41). Since this layer is actually capable of *replicating* objects, an extension to Manta could be built that takes of advantage of this for replicating *Java* objects. We will discuss it in the next section.

### RepMI

*RepMI* (Maassen et al. 2000), which stands for "Replicated Method Invocation," is a subsystem of the *Manta* platform (see previous section) that allows the programmer to *replicate* objects. In principle, this would not entail a diversion from Java's standard programming model, if replication were completely hidden from the application code. RepMI does, however, impose a non-standard programming model in order to replicate objects efficiently.

The key idea is to replicate certain specially marked objects or groups of objects on all nodes of a parallel application. Invocations of read-only methods of these objects can then be carried out locally, while invocations of write methods are broadcast to all replicas using totally ordered group communication (which is provided by the Panda software layer upon which Manta and RepMI are built). RepMI thus uses an *update* protocol (as opposed to an *invalidation* protocol) with *function shipping.*

Replicating individual objects leads to a high communication overhead when complex data structures, such as graphs or linked lists, are to be replicated. For this reason, RepMI uses *clusters* of objects as the unit of replication. A cluster is a programmer-defined collection of objects with a single entry point, called the *root* object, and an arbitrary number of objects reachable from the root, called the *nodes.* To define clusters, the programmer must mark an application's classes with the special interfaces *manta.replication.Root* and *manta.replication.Node.* The compiler then ensures that references to node objects remain *private* to the root object. As a consequence, the root object is the only object of a cluster that is externally visible.

RepMI classifies the methods of root and node objects into read and write methods. This analysis is carried out partly in the compiler, and partly in the run-time system, because it is generally undecidable at compile-time due to polymorphism. This information is then used to carry out read operations locally, and invoke the broadcast mechanism for write operations.

To ensure that the semantics of the centralized program are not affected, the compiler imposes several further restrictions on cluster objects:

- *No remote references.* Cluster objects (roots and nodes) cannot contain references to remote objects, because this could lead to the *nested invocation problem*: If a method of a replicated object $a$ calls a write method $m$ of a non-replicated, remote (i.e. shared) object $b$, then all replicas of $a$ would independently invoke $m$ on $b$, leading to multiple state changes. The RepMI compiler therefore disallows remote references in clusters; this also implies that root objects can only have primitive or array or node type parameters.

- *Restrictions on the use of special interfaces.* Classes cannot implement both the *Root* and the *Node* interface, because that would make it difficult to cleanly separate different clusters from each other. As a result, there may be no references from a node object back to the root object of its own cluster. Another consequence is that root and node objects may not themselves be remotely invokable.

- *No static variables.* There may be no static variables in root and node objects, as these could also be accessed from outside the cluster.

- *Only calls to "well-behaved" methods.* Within root and node objects, only "well-behaved methods" of other classes may be called. A well-behaved method is a method which deterministically produces identical results on all machines. This means that the implementation of well-behaved methods must not depend on static variables or methods, random generators, I/O, or the local time.

Clusters are always replicated on *all* nodes of the computing environment. Technically, the replicas are created immediately when the root object is created on one of the nodes. The application code on a remote node, however, can only access its local replica after the creating node has passed a reference to it. (The fact that replicas are created on all nodes, and not only on those that actually need them, is not a principal restriction of RepMI. It only has not been implemented otherwise because for the kind of small-scale parallel applications for which the platform is intended, the existing behaviour is well-suited.)

Two case studies are presented in Maassen et al. (2000). The first is a branch-and-bound implementation of the Traveling Salesperson Problem (TSP). In this program, each processor needs to access a *Minimum* object to compare its current solution to the best solution that has yet been found. Read accesses to this object are much more frequent than write accesses. In a naïve implementation, the *Minimum* object is simply a remotely invokable object shared among all processors, but this creates a prohibitive bottleneck, resulting in no speedup at all. The authors then created (1) a manually optimized version of the program, where the current minimum value is explicitly transferred to all processors using remote method invocations, and (2) a program where the *Minimum* object is replicated using RepMI. Both programs significantly outperformed the naïve implementation, with the RepMI version achieving only slightly less speedup than the manually optimized program.

The second case study is a program solving the All-Pairs Shortest Paths Problem (ASP). Here, a *Matrix* object needs to be worked on by all processors in a way similar to the TSP example. Following the same strategy as for the TSP program, the authors found that the RepMI version achieved an even higher speedup than the manually optimized version, due to the use of the efficient broadcast mechanism supplied by the Panda layer (which was not available to the purely RMI-based, manually optimized version).

#### Javanaise

Javanaise (Hagimont and Louvegnies 1998) is a Java platform where distribution is handled entirely via object caching; the unit of caching being a *cluster* of objects. A cluster is defined as a root object along with any private objects that it contains. Via caching, the clusters are shared between the separate parts of an application, and the run-time system can also make them *persistent* by storing them on disk.

The caching mechanism is implemented via proxy objects within the application program itself (and code transformations are used to substitute proxies for the classes written by the programmer). According to our classification, Javanaise is therefore an *explicit* system, although in an unusual sense because it uses caching, rather than remote invocation as its communication mechanism (at the implementation level, Javanaise processes communicate via sockets).

An additional, unusual feature is that application deployment is based on applets and code downloading. A new client can “join” an existing application by connecting to it via a web browser; the local program is then executed as an applet within a web page. (The primary application domain of Javanaise is collaborative work on an Internet scale.)

Javanaise clusters are defined by an external configuration description, specifying which classes are roots of clusters (called “cluster classes” for short). One requirement that a cluster class must fulfill is that the types of any reference parameters of its methods are also cluster classes. This means that clusters can only import or export references to other clusters, but not to local, intra-cluster objects. The authors claim that clusters can readily be found in normal application designs, and need only be identified during configuration. (Additionally, the types of all objects within a cluster need to be serializable so that clusters can be transferred over the network.)

After cluster classes have been identified, interfaces are generated for them (and the original classes renamed) to allow substitution of proxies at run-time. In a second step, these cluster interfaces are then annotated by the programmer to specify synchronization and consistency requirements. In the implementation reported in Hagimont and Louvegnies (1998), it is only declared whether methods are readers or writers, which allows for a simple invalidation-based consistency protocol. Other consistency protocols would be possible, the authors claim, possibly via different annotations.

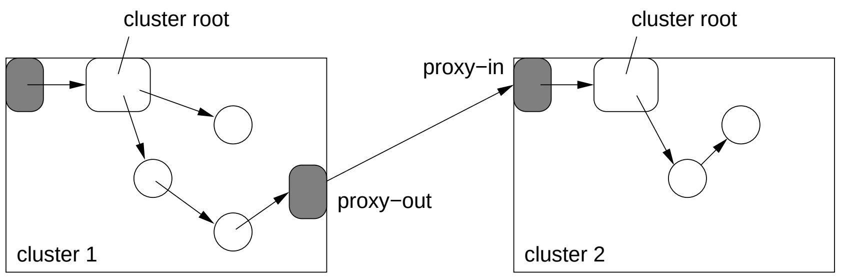


Figure 2.10: Proxy Mechanism in Javanaise

Figure 2.10 shows the proxy mechanism of Javanaise in detail. Any reference to another cluster is realized via two proxies: a *proxy-out* object in the originating cluster, and a *proxy-in* object in the cluster that is referenced. (Note that both clusters reside in the same address space — there are no remote references in Javanaise.) If there are multiple references to a single cluster, then each of the referring clusters has its own proxy-out object, but there is only a single proxy-in object in the cluster being referred to.

The proxy-out object contains a reference to the proxy-in object, and normally forwards all method invocations to it. The proxy-in object, on the other hand, contains a reference to the actual root object of the cluster, and forwards all method invocations to that object. The references within the proxy objects, however, can also be null, and this means that the corresponding cluster is not currently cached in local memory and needs to be fetched from where it resides. To make this possible, each cluster has a globally unique identifier, which is also stored in the proxy objects.

Invalidating a cluster on a given node can thus be accomplished by assigning null to the reference within the proxy-in object. In this way, a strict coherence protocol can be implemented (where each cluster always resides on a single node only), but also more relaxed forms of consistency such as entry consistency.

### Doorastha

The *Doorastha* system (Dahm 2000a,b) follows a similar path as JavaParty and Manta do, using Java/RMI as an underlying distribution technology, but providing a higher-level, more transparent programming model. As in JavaParty and Manta, classes can be specially marked as "remotely invokable", but Doorastha gives the programmer finer control over this and the resulting semantic effects.

Doorastha does not change the Java language itself. Instead, classes are marked with special comments (tags) that are recognized by the Doorastha compiler. This has the advantage that Doorastha programs can also be compiled with a traditional compiler, resulting in centralized execution. It is also possible to mark the byte code of system classes externally, i.e. without actually modifying their source code.

Remotely invokable classes are called *global* classes, which means that instances of them can be accessed from everywhere within an entire distributed system. The term "global" is more accurate than "remote" in JavaParty and Doorastha, since "remoteness" is a relation between objects, not a property of a single object or class. This also helps to underline the point that "global" objects do behave differently than "local" ones, even when they are not called remotely.

To make a class remotely invokable, the programmer marks it with the tag *¡globalizable¿*. As the word implies, instances of such classes are not necessarily global, but can be turned into global instances dynamically at run-time. This happens if and only if a reference to such an instance is actually passed to a remote node; only then is a stub and skeleton created, which results in higher performance for globalizable objects that are only used locally.

Classes may also be marked as *copyable* or *migratable*. Instances of copyable classes may be passed to remote nodes by value, via serialization. To mark them copyable may be regarded as an indication of the programmer that it is semantically safe to do so, or that he is aware of the potential consequences. Migratable classes, on the other hand, are a subset of globalizable classes. Instances of migratable classes are remotely invokable, but may also move from one node to another to improve performance. (Migration is essentially *weak* in Doorastha, with some support for "semi-strong" migration of threads. For details, see Dahm (2000b)).

The annotations of classes specify how their instances *may* be used in a distributed setting. The actual behaviour at run-time is determined by further annotating individual method parameters and instance variables. There are three possible parameter passing modes: *by-refvalue*, *by-copy*, and *by-move*. The first, *by-refvalue*, is the standard Java passing mode (references to objects are themselves passed by value). *By-copy* means to pass an object by value via serialization, and *by-move* means to pass it by refvalue, but to move it to the remote receiver at the same time, as in Emerald (see page 17). The same three modes may also be applied to individual instance fields, indicating how a field should be treated when the containing object is copied or moved: for example, if the containing object is moved to another node, any fields of it that are annotated as *by-copy* are not moved along, but copied to the receiving node via serialization.

The Doorastha compiler and the run-time system check whether the tags of classes, parameters, and fields are compatible (e.g. only instances of globalizable classes may be passed to remote objects by refvalue). The compiler performs static checking of these constraints, and generates warnings if the constraints are not guaranteed statically, but might still be fulfilled at run-time. For example, the declaration:

```
public class Alpha { ... }

public /*: globalizable */ class Beta {

  public void method (Alpha a) { ... }

}
```

generates a compiler warning, because instances of class *Alpha*, which is not globalizable, are passed to possibly globalized instances of class *Beta by refvalue* (the standard Java passing mode). However, this is only a warning because at run-time, the parameters might still be of a globalizable subtype of *Alpha*.

Similar to JavaParty, Doorastha's execution environment is realized as a group of cooperating JVM processes which must be started and registered before any program can be executed on them. It is thus possible to implement remote object creation, and the programmer may request it by adding a special tag to allocation expressions:

```
Beta b = new /*:remotenew :host="eagle"*/ Beta();
```

```
import do.shared.*;

public class Simple_Parallel {

  public static void main (String[] args) {

    Array tasks = new Array (N);
    Array data  = new Array (N);

    for (int i=0; i<N; i++) {
      tasks.add (new My_Task(), i);
      data.add (new Param(), i);
    }

    Par par = new Par (tasks, data);
    par.call ();
  }

}
```

Figure 2.11: A Do! Program

Doorastha is one of the back-end platforms used by Pangaea. We will therefore revisit Doorastha in chapters 3 and 4.

### Do!

The *Do!* system (Launay and Pazat 1997, 1998) is a framework for parallel programming in Java. The key idea is to provide the programmer with a set of *collection* data types that can be used in a distributed setting. For parallel execution, the elements of such a collection are mapped onto distinct processors, and it is the implementation of the collection class that handles any remote communication. A program typically defines a collection of tasks and a collection of data. At execution time, the tasks are combined with the data elements and distributed using an "Operator" design pattern, resulting in an SPMD-like computation that follows the "owner-compute rule" (the processor on which a data element is mapped is the only processor that can modify the data).

The listing in Fig. 2.11 shows the setup of a simple distributed computation. The *Array* data type is one of the special Do! data types; it is used here to create distributed collections of both tasks (threads) and of data (the classes *My_Task* and *Param* are user-provided and not shown here). To execute the collection of tasks on the data, a *Par* object is created in the second-to-last line, and then invoked.

The communication mechanism that is used within the distributed collections relies on Java/RMI. Apart from this "implicit" remote communication, the Do! system also supports explicitly distributed programming, correcting for the deficiencies in Java/RMI in much the same way as JavaParty, Manta, or Doorastha. The programmer can mark certain classes as remotely accessible (implementing the marker interface *Accessible*); a special compiler is then used to transform these classes into RMI "remote" classes, preserving local parameter passing semantics via code transformations (no details are given).

## ProActive

*ProActive* (Caromel et al. 1998) is "a Java library for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework" (from the project home page). ProActive was formerly known as *Java//* ("Java Parallel"), and is based on previous work on *Eiffel//* (Caromel 1993) and *C++//* (Caromel et al. 1996).

ProActive is a Java platform that adheres to the remote invocation model, but it suggests a different programming model than standard Java, based on an *active object pattern*. An active object is generally defined as an object with its own thread of control. In ProActive, this concept is applied to remotely invokable objects. An active object serves remote invocations asynchronously: control immediately returns to the caller, while the actual request is stored in a queue and processed later by the active object's own thread. The fact that the invocation is handled asynchronously is hidden from the caller via *transparent futures*: any object returned by a remote call is transparently substituted with a placeholder object; the result is only retrieved when the caller actually accesses this object.

The active object pattern can thus introduce concurrency, and hence, parallelism even into strictly sequential programs. If there are enough places where computations are called (triggered) well before the results are actually used, then an overlap of communication and computation results in the distributed case, and speedup may occur. It is also possible, though, to program active objects explicitly, thus expressing other forms of concurrency.

ProActive is implemented as a pure Java library without any language extensions or special execution environment. The transformation of objects into remotely invokable, active objects occurs transparently, on the fly, at run-time, by means of a meta-programming library and Java's built-in reflection facilities.

Making objects *active* can be accomplished in one of the following ways (taken directly from Caromel et al. (1998)). A standard object creation expression:

```
A a = new A ("foo", 7);
```

is augmented by adding a node to instantiate the object on. There are three different ways to do this:

- instantiation-based

  ```
  Object[] params = { "foo", new Integer(7) };
  A a = (A)Javall.newActive ("A", params, myNode);
  ```

- class-based

  ```
  class pA extends A implements Active {}
  Object[] params = { "foo", new Integer(7) };
  A a = (A)Javall.newActive ("pA", params, myNode);
  ```

- object-based

  ```
  A a = new A ("foo", 7);
  a = (A)Javall.turnActive (a, myNode);
  ```

It is thus possible to integrate remote method invocation and active objects very seamlessly into standard Java programming. The activity of an active object is realized by a transparently-added thread of control that executes a special method, *live*. The default implementation of *live* (which is added transparently to the active object itself) looks as follows[6]:

```
public void live (Body myBody)
{
  while (true)
  {
    myBody.serveOldest();
    myBody.waitARequest();
  }
}
```

which means that requests to the object (remote invocations) are handled in a FIFO manner (the *Body* object is an object of the framework that owns the actual queue of requests). The programmer may also override the *live* method to specify different handling policies. For example, the following code realizes the classical bounded buffer algorithm, where *put* operations are only allowed if the buffer is not full and *get* operations only if the buffer is not empty:

```
class BoundedBuffer extends FixedBuffer implements Active
{
  public void live (Body myBody)
  {
    while (true)
    {
      if      (this. isFull ())  myBody.serveOldest ("get");
      else if (this.isEmpty()) myBody.serveOldest ("put");
      else myBody.serveOldest();
    }

    myBody.waitARequest();
  }
}
```

One of the main design goals of ProActive has been to create a library-based solution in 100% Java, with no additions to the language, no special compiler, and no special execution environment (it is indeed possible, and one of the stated goals, that objects can be turned active even without access to their source code). The price that Caromel *et al.* pay for this commitment is that their system cannot handle all Java constructs. In particular, instances of `final` classes (which includes all arrays) cannot be reified by the metaprogramming library; the same is true for values of primitive types. The programmer is thus somewhat restricted in his choice of language constructs, although these restrictions can easily be overcome within the language itself (e.g. by using the standard wrapper objects for primitive values). It could be argued that ProActive's limitations must eventually be blamed on the not fully object-oriented nature of Java itself.

The case studies discussed in Caromel et al. (1998) are (1) a distributed matrix-vector product computation, and (2) an interactive, collaborative ray tracer application where several users can concurrently modify a scene and observe the results on their screens. The matrix-vector product program shows performance comparative to other distributed Java platforms: the main bottleneck is identified as the serialization mechanism used by the underlying Java/RMI. The authors conclude that unless this is enhanced, "it is today hard to achieve speedup on a network of workstations where the communication/computation ratio is too high." The collaborative ray tracer shows how complex interactions and synchronization constraints can be expressed by the

[6]Caromel et al. (1998) do not show the *public void* keywords before *live* when they present this example, but since this would be illegal Java, and since one of the other examples further down in the paper does have the modifiers, we assume it is simply an oversight.

*live* method of a central dispatcher object. It is noted that in a straightforward design of this program, it would have been natural to make instances of a class inheriting from *java.awt.Frame* active. This approach was then abandoned because *java.awt.Frame* is “a heavy-weight class with a lot of public methods.” Apparently, it was necessary to work around this (by introducing another, user-written receiver object for the remote calls) because the generated stubs for the AWT-based class would have been too large, and the resulting overhead too high.

## 2.2 The Implicit Approach

Rather than to deal with distribution from within the programming language, it can also be handled *implicitly* by the execution environment. The advantage of this is that it achieves complete distribution transparency from the programmer’s point of view. The disadvantage is that because the abstraction is made *below* the programming language, the resulting efficiency can be poor.

Historically, the implicit approach was first implemented in *Distributed Shared Memory (DSM)* systems (Li and Hudak 1989). The idea of DSM systems is to provide programs with a shared address space, which is transparently realized via message passing and data caching. When a processor attempts to access a data item that is not in local memory, it is transparently fetched from the node where it resides, and then accessed locally on the processor that needs it.

While distribution is thus invisible to the programmer, realizing the shared memory abstraction can be quite expensive, involving much more communication and computation overhead than with remotely invoked methods. In order to achieve reasonable efficiency, it is often necessary to introduce a “relaxed” *consistency model*, where objects are allowed to be temporarily out of date, in a well-defined fashion.

DSM systems can be *hardware-based* or *software-based*. The former approach is mainly used for high-end supercomputers, while the latter approach allows to create a DSM abstraction on ordinary hardware, such as loosely coupled PCs and workstations, and even for machines on wide-area networks such as the internet.

The unit of distribution and caching is usually a memory *page* of fixed size; we call such DSM systems *page-based*. This, however, may lead to the problem of *false sharing* when unrelated data items happen to be stored on the same memory page. To alleviate this, one approach is to make the shared memory *object-based*, which means that individual objects, rather than pages, are transferred and cached. Since data items within an object are usually closely related, false sharing is reduced.

When objects are *cached*, a given object may exist, at least temporarily, in multiple replicated instances. To keep these replicas consistent, various protocols can be used. The two general approaches are:

- *write-invalidate*, which means that when one replica is written (changed), then all other replicas are invalidated and need to be re-fetched by their corresponding hosts, and

- *write-update*, which means that any change to one of the replicas is written through to all the other replicas using a broadcast mechanism.

A variant of both of these protocol types is to allow *multiple writers* at the same time, which is realized by techniques called *twinning* and *diffing* that reconcile and merge concurrent changes.

The strongest form of consistency between all replicas would be if they had identical values all of the time. It is impossible to achieve this, due to the non-existence of a global time. The question, then, is what weaker form of consistency is realized by a given protocol. The following forms of consistency are generally distinguished (sorted from stronger to weaker categories):

- *Sequential consistency* means that the effect of a program execution on a replicated object is the same as that of an equivalent sequential execution on a non-replicated object.
- *Causal consistency* means that the effects of causally dependent write operations are observed in the same order by all nodes. A weak form of causal consistency results if only the effects of write operations *of any single process* are observed in the same order by everyone; this is called *pipelined RAM consistency (PRAM consistency).*
- *Weak consistency* is achieved if synchronization operations are sequentially consistent, and each synchronization operation of a process only completes if (a) all writes of the process to external replicas have completed, and (b) all writes of other processes to this processes' local replicas have also completed ("synchronization forces temporary consistency").
- *Release consistency* is a variant of weak consistency where replicas are only updated when synchronization objects (locks) are *released.* In the basic form of release consistency, the release operation only terminates after all replicas have acknowledged the update operation. *Lazy release consistency*, on the other hand, is a variant where this acknowledgement is not sent; instead each process, when it *acquires* a lock, requests the updates from the process that was the last to *release* that lock.

For a thorough discussion of these consistency models, and further weak consistency models, see Tanenbaum (1995).

Caching is however not the only way to handle distribution implicitly. Remote invocation is another option, and has in fact been used on newer systems, often in combination with caching and replication schemes. This underlines our thesis that the type of abstraction (implicit or explicit) in a distributed object system is independent from the actual communication mechanisms used.

### Orca

Orca (Bal and Kaashoek 1993, Bal et al. 1998) is an object-based programming language and distributed run-time environment. Although being described by its authors as a DSM system, its initial design dates back to before the term DSM had actually been coined. Many of the implementation decisions in Orca are therefore somewhat unusual when compared to other DSM systems; they do however fit well with our classification of Orca as an implicit distributed object system.

The Orca language was developed specifically to facilitate parallel programming and allow special kinds of compiler and run-time optimizations. The Orca run-time system is entirely software-based and, at least in its second version, has been implemented with the explicit goal of portability. Recently, parts of the Orca infrastructure have been re-used to build the Manta platform with its replicated method invocation (RepMI, see pages 32 and 33).

The unit of distribution in Orca is the *shared object.* A shared object, defined in the Orca language, encapsulates a number of data items and makes them accessible only via the object's operations. There is no inheritance or polymorphism in the Orca language, and it has in fact been further simplified: each operation can only access exactly one object; it is not possible to build arbitrary graphs of objects via references. Operations are atomic, which means that Orca objects are implicitly "synchronized" and resemble monitors.

A shared object is either stored on a single machine (and invoked remotely via RPC), or replicated on all machines of the system. Coherence between the replicas is achieved using a *write-update* protocol with *function shipping* (when a write operation is invoked on a replica, the same operation is invoked on all other replicas so that their state changes correspondingly). Coherence between the replicas is guaranteed because the update protocol uses a *totally ordered broadcast* mechanism provided by the system.

The decision whether to replicate an object, or where to store it if it is not replicated, is taken dynamically by the run-time system, guided by analysis performed in the compiler. The idea is that if an object has a high read/write ratio (it is much more often read than written to), then it is worthwhile to replicate it. If the ratio is low, it should be stored on a single machine (because the many write operations would incur a high broadcasting overhead otherwise).

To estimate the read/write ratio, the compiler computes, for each process, the pattern of object invocations it is likely to perform at run-time. These patterns are represented in an augmented regular expression syntax such as:

**A\$W; [ B\$R | {C\$W} ]**

which means that the process will perform a write operation on object **A**, followed by either a read-only operation on object **B** or a repetition of write operations on object **C**. To estimate control flow, the compiler assumes that each branch of an if-statement is taken 50% of the time, and that loops are iterated "many" times, with a configurable meaning of "many" (16 is reported in Bal and Kaashoek (1993)).

At run-time, the compiler's estimates are used to decide on an object's initial placement or replication. The run-time system, however, keeps track of the actual invocations performed during execution, and gradually decreases the influence of the compiler estimates by means of an aging mechanism. If the read/write ratio passes a certain threshold, a non-replicated object may become replicated and vice versa. Non-replicated objects are stored on the machine that accesses them most frequently; if this machine changes, then non-replicated objects can effectively *migrate* to another machine as well.

The performance of the Orca system was studied with a suite of ten typical parallel programming applications, including the All-Pairs Shortest-Paths problem (ASP), a Barnes-Hut simulation, and the Travelling Salesman Problem (TSP). To evaluate the effect of the compile-time and run-time analyses, four different versions of the programs where created:

- a *manual* version where all objects were optimally placed and/or replicated by the programmer
- a *static* version where only the estimates by the compiler where used for placement decisions
- a *dynamic* version where only the run-time statistics where used
- a *combined* version where both the compiler estimates and the run-time information where used

The result is that the automatic placement decisions at compile-time and run-time (*combined*) come very close to the optimal placement specified by the programmer. For eight of the ten programs, the speedup is within 1% of the optimal version, while for the other two, the *manual* speedup is 8.7% and 4.6% higher, respectively.

The *static* and *dynamic* programs reveal that the run-time statistics are more important than the compiler heuristics. Only for one of the ten programs was the *combined* version slightly faster than the *dynamic* version with no compiler support. For another program, the *combined* version was slightly slower because the compiler made a wrong initial placement decision for certain objects; this decision subsequently had to be corrected by the run-time system. If the compiler heuristics *alone* are used for placement decisions (the *static* programs), then the performance can degrade significantly if the compiler guesses wrong. The static version of one of the ten programs had prohibitive performance, two others could not be completed at all due to memory problems because of wrong placement decisions.

In summary, the Orca system shows that automatic placement decisions can be as good as manual object placement by the programmer. The most significant drawback of Orca is that it uses a special-purpose programming language that is not fully object-oriented, and has been kept simple precisely to facilitate static analysis.

### Java/DSM

Java/DSM (Yu and Cox 1997) was one of the first DSM systems for Java. It is built on top of the TreadMarks DSM platform (Keleher et al. 1994), as a modified Java 1.0.2 virtual machine that places the heap into the shared memory region provided by TreadMarks.

TreadMarks, by itself, is a software-based, page-based DSM system that runs on standard Unix operating systems and workstations. To reduce communication overhead, especially by tackling the problem of false sharing, TreadMarks uses a lazy implementation of release consistency, and features a multiple-writer protocol. With these techniques, high speedups could be achieved for a number of standard parallel computing benchmarks.

Java/DSM does not introduce any new optimizations that would specifically exploit or support the object-oriented nature of Java. Instead, the authors focus on heterogeneity (allowing changes of data representation within objects from one machine to another) and distributed garbage collection.

The case study that is reported for Java/DSM is a distributed spreadsheet application that allows collaborative work. No performance figures are given for this application, but the authors demonstrate how the shared memory approach results in much easier programming than using Java/RMI.

### Hyperion

Hyperion (Antoniu et al. 2000) is a Java DSM platform that is built on top of an existing run-time system named PM2 which provides lightweight threads, inter-node communication primitives, and a generic DSM layer. Hyperion uses a special compiler that translates Java bytecode to C code with embedded calls to the PM2 run-time system; a traditional C compiler is then used to translate this C code into native machine code. The Hyperion/PM2 platform itself runs on several UNIX variants using a variety of network protocols.

The DSM layer of PM2 is page-based and allows for arbitrary consistency protocols that can be implemented by customizing a *DSM page manager* and *DSM communication module.* Hyperion takes advantage of this to implement the special, “relaxed” consistency protocol that is allowed by the Java language specification (Gosling et al. 2000). Rather than requiring sequential consistency, a Java implementation is free to provide a thread with its own local memory in which it may cache objects. At synchronization points (i.e. when a Java lock is aquired or released), these cached objects must be flushed into main memory. A Java programmer is thus able to require stricter forms of consistency by using the Java synchronization primitives, while a minimal use of synchronization enables a more efficient implementation under the “relaxed” consistency model.

In a case study, the authors compare the performance of a multi-threaded program running under Hyperion to that of sequential versions of the same algorithm, using both Java (which is subsequently translated into intermediate C code), and hand-written C code. The program studied solves the minimal-cost map-coloring problem using a branch-and-bound approach. The execution times (in seconds) for the sequential versions are as follows:

| program version | time | factor |
|---|---|---|
| Hand-written C | 63 | 1.00 |
| Java via Hyperion/PM2 | 324 | 5.14 |
| Java via Hyperion/PM2, in-line DSM checks disabled | 168 | 2.67 |
| Java via Hyperion/PM2, array bounds checks also disabled | 98 | 1.56 |

The DSM access checks do incur a significant overhead; removing them saves nearly 50% of the execution time. The authors speculate that a DSM system that relies on *page-faults* rather than in-line access checks might give better results; see also our discussion of Jackal (page 46) which also uses access checks, but attempts to optimize many of them away in the compiler.

However, a significant amount of performance loss, when compared to the C implementation, is also due to the Java language itself, e.g. its strict bounds-checking of arrays. But even without these checks, a penalty of about 56% remains, which must be attributed to the object-oriented programming paradigm. (The authors conclude that a comparison to hand-written C++ "would probably be more fair to Hyperion".)

The times for parallel execution of the multi-threaded version do show a significant speedup, however. The execution time on a single node is already slightly lower than that given for the sequential version above (273 s), because the multi-threaded version follows a more efficient search path for the particular problem at hand. As more nodes are added, the system achieves between 78% and 90% of the ideal speedup.

### Juggle

Juggle (Schröder and Hauck 1998, 1999) is described by its authors as "a distributed virtual machine for Java". It allows the distributed execution of standard, multi-threaded Java programs without any changes in the source code. This is realized by reimplementing some of the JVM's byte code instructions to potentially access data on a remote host. According to our classification, Juggle therefore is a distributed shared memory system, although in an unusual sense.

In Juggle, method invocations are always local, while data is potentially accessed remotely, by means of the redefined byte code instructions. Unlike traditional DSM, accessing a remote data item does therefore not result in a fault and a subsequent caching of the data item in local memory; it is rather realized as a *remote invocation at the JVM level.* This may of course result in poor performance if a method performs several accesses to instance variables. Juggle therefore allows objects (i.e. their instance variables) to *migrate* to other machines, or let them be *replicated*, under the control of a runtime system that monitors which thread accesses an object most frequently.

The authors do however not give any details about the actual migration or replication mechanisms used; it is therefore unclear whether migration is strong or weak, or what consistency model and update policies underlie the replication system. As a matter of fact, the authors admit that while they did already have a working implementation of Juggle's virtual machine, the actual distribution of programs was only simulated.

In a case study they perform, the authors therefore demonstrate Juggle's effectiveness merely by counting the number of local vs. remote accesses to instance variables, with and without migration and replication enabled. The program studied is a ray tracer that renders a given scene; at run-time, it consists of roughly 100,000 objects of which 188 are actually "distributed".

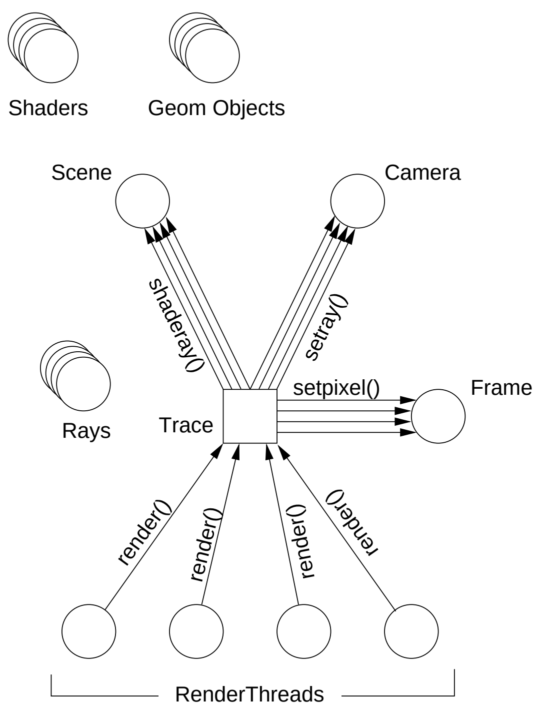


Figure 2.12: Call Graph of the Juggle Ray Tracer

While a distributed execution without migration or replication enabled resulted in $5.5{\cdot}10^6$ remote accesses, this number could be reduced by a factor of 20 with migration and replication switched on.

It is instructive, however, to take a closer look at the ray tracer program itself.[7] The main rendering loop of the program looks as follows:

```
public static void render(Frame f, Scene sc, int sx, int sy, int ex, int ey) {
  ...
  Camera cam = sc.cam;
  for (int y = sy; y < ey; y++) {
    for (int x = sx; x < ex; x++) {
      Ray r = new Ray();
      cam.setray (r , ( double)x / six, (double)y / siy);
      Color c = sc.shaderay (r);
      f.setPixel (x, y, c);
    }
  }
}
```

The *render()* method is executed by each of the parallel threads, with each of them using different parameters *sx*, *sy*, *ex*, and *ey* to indicate which part of the scene should be rendered. The *render()* method itself, however, is *static*, which means that it only exists in a single instance according to Java's centralized programming model. In Juggle, however, we have seen that methods are replicated on every node of the system, and it is clear that the programmer, in writing the ray tracer, took advantage of that implementation technique. This is further underlined if we consider the other objects involved in the computation. Figure 2.12 shows a

[7]I would like to thank Michael Schröder for giving me access to the source code.

schematic call graph of these objects.

The graph shows that in addition to the static *render()* method, there is also only a single instance of the *Scene*, *Camera*, and *Frame* classes, "shared" by each of the rendering threads, with the *Scene*'s *shaderay()* method actually performing most of the rendering work.

Again, this structure only allows for parallel execution if the code of these objects is internally replicated on each node. A perhaps more object-oriented design would be to encapsulate the rendering logic in objects local to each thread (which could then be distributed instance-wise), and let them operate on a shared, but passive data structure describing the scene (cp. the ray tracer in our second case study in chapter 4). On the other hand, it could be argued that it is good object-oriented practice to make the data objects themselves "intelligent" and let them do the actual work.

Either way, it is obvious that the ray tracer, as it is written, can only be distributed efficiently on a platform with implicit code replication. There is no way to achieve anything similar on a pure remote invocation platform without replication.

### Jackal

Jackal (Veldema et al. 2001a,b) is a DSM compiler and run-time system for Java that targets x86 architectures under Unix. The focus of Jackal is to provide aggressive optimizations at both compile-time and run-time. It is a software-based, *fine-grained* DSM system, where *fine-grained* refers to the unit of caching, named a *region* of 64 bytes. Objects are stored 64-byte-aligned; an object may occupy one or more regions, but there can never be more than one object in a given region. This eliminates the problem of false sharing between objects, and Jackal may therefore also be considered an *object-based* DSM system.

Cache coherence is realized via software access checks inserted by the compiler. The consistency protocol uses a home-based approach, where the master copy of each object is kept at the node that created the object. The protocol allows multiple writers and uses an invalidation scheme. Jackal takes advantage of the "relaxed" memory model allowed by the Java language specification, in that regions are only flushed to or from main memory (the home node) at synchronization points. The run-time system, however, goes even further than that and minimizes communication by a technique called "adaptive lazy flushing". It exploits the fact that flushing can be avoided if a region is only accessed by a single processor, or if it is only *read* by the threads that access it.

Jackal's *compiler optimizations* seek to eliminate superfluous in-line access checks. The compiler employs a number of existing, conventional techniques (common subexpression elimination, full redundancy elimination, and array aggregation), and introduces two novel approaches, *object-graph aggregation* and *computation migration.*

*Object-graph aggregation* allows the compiler to detect situations where a root object references other objects, and accesses them immediately after a call to the root object. These secondary objects can then be fetched along with the root object, and the compiler may remove any access checks for these secondary objects. To find graphs of related objects, the compiler uses the traditional technique of heap approximation (Ghiya and Hendren 1998, Whaley and Rinard 1999).

*Computation migration*, on the other hand, means to augment the DSM approach with remote invocation: a computation is said to *migrate* to a remote node if it invokes code on that node, thereby suspending the local thread until the remote code has been executed and control returns. This technique, which is equivalent to a remote procedure call, is used in two situations in Jackal: First, to execute a critical region that is protected by a lock on the home node of the lock, and second, to co-locate the execution of a thread constructor with the data created by that constructor. Unlike traditional RPC or Remote Method Invocation, it is however not the user-defined methods that are called remotely. In Jackal, it is possible to execute only part of a method remotely (e.g. a critical region). The compiler extracts this region and wraps it into a newly generated, remotely invokable method, passing all the relevant variables to it as parameters.

A suite of four applications was studied to analyze the impact of Jackal's optimizations. On the one hand, it was found that the access checks added by the compiler *without optimization* increase the sequential execution time by a factor of up to 5.5 (2.2 on average). This is a very similar value as that reported for the Hyperion system, see page 43. When enabled, Jackal's compile-time optimizations reduce this overhead to a mere 0.1 on average.

To evaluate parallel performance, the programs running under Jackal were compared to hand-optimized RMI versions of the same algorithms, where the programmer has complete control over communication patterns. Even with the access-check elimination optimizations switched on, the performance of the DSM programs was generally poor when compared to the RMI versions. Only when computation migration (compile-time) and lazy adaptive flushing (run-time) was enabled, a "reasonable to good" performance could be achieved. Of these latter optimizations, computation migration (i.e. remote invocation on a DSM system) was shown to have the most significant effect, without which the run-time optimizations would have been almost useless.

### cJVM

The *Cluster Java Virtual Machine*, or cJVM (Aridor et al. 1999, 2000), is a Java DSM system that flexibly chooses among various communication mechanisms, including remote invocation and caching both at the class, object, and field level. The decision when to use which mechanism is taken at run-time, based on speculative just-in-time analyses performed at the byte code level.

The goal of cJVM is to execute unmodified Java programs on clusters of workstations, without a program being aware that it is executing on a cluster. cJVM realizes this by means of a modified implementation of the Java Virtual Machine, in which some of the byte code instructions are re-implemented so that they can potentially operate remotely. There is thus no special compiler in cJVM; it can execute arbitrary Java byte code.

Communication in cJVM is based on remote invocation and proxies, although in a generalized sense: there are special kinds of proxies that execute methods locally, rather than actually invoking the master object remotely. This optimization is possible if (a) a method is entirely *stateless* (it does not read or write any of the receiving object's state at all), or (b) the method only *reads* some of the fields of the receiver (these fields are then cached within the proxy). A cJVM proxy contains implementations for all three of the above cases (remote invocation, stateless method, and read-only method), and can choose either of them for each individual method. This choice can be based either on just-in-time analysis of the application's byte code as it is loaded, or on observations of an object's communication behavior at run-time.

*Caching* thus happens in cJVM at various levels of granularity:

- The state of *classes* (static fields) is speculatively cached on every node, based on the observation that this state is usually not modified after class-loading time. If this speculation turns out to be wrong (a static field is indeed modified), then all replicas are invalidated and the value is henceforth accessed remotely on its master host.

- For objects that can statically proven to be *immutable*, the entire state is cached within the proxy. All operations thus execute locally, and there is no need for an invalidation protocol. (Immutability is detected at class-loading time by analyzing the byte code.)

- Additionally, cJVM can cache individual fields of an object if these fields are *read-only in practice*. A field is read-only in practice if, in a particular run of the program, the field is not modified after the object containing it has at least one proxy. At run-time, a read-only in practice field can be implemented as *read-locally*, meaning that it is cached within the proxy. Whenever a mutable class is loaded, cJVM marks all non-static, private fields as read-locally. If a read-locally field is modified after it is cached, the replicas are invalidated and the field loses its read-locally status. The read-locally status is kept at a *per class, per field* level, based on the observation that usually, all instances of a class will behave similarly within a program.

Based on the above caching policies, cJVM can thus execute some methods locally on the proxy. In particular:

- *Class methods* are always executed locally, since usually the static fields that they access will be available locally too. If a field is not cached, then only that particular field is accessed remotely, but the methods still executes locally.
- *Stateless methods* (detected by a simple load-time analysis) are always executed locally.
- Methods that only access *read-locally* fields are also executed locally. If only one read-locally field is invalidated, though, the method will henceforth be executed remotely.

A further class of optimizations concerns *object placement*. To ensure that the master copy of each object is placed onto the most suitable node (where the least remote communication results), cJVM identifies *factory methods* and attempts *single chance migration*.

A *factory method* is a method that creates an object which it then returns. There is a high probability that the invoker of a factory method will subsequently use the returned object, therefore it will usually be best to place this object on the invoker's node. cJVM uses a simple, flow-insensitive, non-conservative analysis at the byte code level to identify factory methods. To ensure correct object placement, factory methods are then always executed locally at the invoker's site. This way, the invoker's site is where the master copy of the returned object will be created. Note that cJVM can potentially execute *any* method locally, because the implementation of each individual byte code instruction has been modified so that it can operate remotely if needed.

*Single chance migration* tries to identify cases where an object goes through two distinct, non-overlapping phases in which it is used by two different threads. For example, it is common in many programs that one thread creates an object and interacts with it to set it up, then passes it to another thread that exclusively uses it for the rest of its lifetime. In this case, it is beneficial to *migrate* the object's master copy from the first thread's node to that of the second thread.

cJVM tries to identify these cases by (1) finding out which classes might be eligible for the optimization, and then (2) detecting cases where an instance of an eligible class enters the second phase of its lifetime. Initially, the system considers all classes eligible, excepting (among others) those that are not *relatively encapsulated*: a relatively encapsulated class is one whose instances are "not too dependent" on other objects which may be left behind if an instance is moved. At run-time, the list of eligible classes is then pruned whenever the system detects that an object is accessed remotely after it has been migrated (the object's class is then no longer eligible for the optimization). A migration is triggered whenever a node receives an object it has never seen in response to a remote request, and the class of this object is still eligible for the optimization.

In Aridor et al. (2000), a data warehouse application is reported as a case study. The program is based on a common benchmark for enterprise-type applications and comprises approx. 10,000 lines of code. Executing this program under cJVM, an efficiency of 80% could be obtained on a four node cluster. The optimizations employed by cJVM were shown to reduce the number of remote messages significantly, down to about 10% compared to the non-optimized version.

## 2.3 Conclusions

Based on the individual systems that we discussed in the previous sections, a number of general trends in the development of distributed object systems can be identified.

The platforms with explicit abstraction within the programming language differ in how much technical detail they do make visible to the programmer, and how similar distributed programming and local programming are on these platforms. In other words, they differ in the amount of distribution transparency they provide.

In the one extreme, the technicalities of remote invocation or object migration are completely hidden from the programmer, and only the *location* of objects needs to be controlled. This can be done either by language primitives as in Emerald, or externally, in a separate configuration language as in HERON. At the other end of the spectrum are platforms such as CORBA and Java/RMI that require a lot of distribution-related programming, ultimately achieving only shallow distribution transparency at the syntactic level of individual method calls, with the semantics of remote invocations being actually different.

As a general trend, it can be observed that those platforms featuring their own programming language that was specifically designed for distribution achieve the highest degrees of transparency (Emerald, DOWL, also Orca). Whenever an existing language is used, compromises need to be made due to the fact that the language was not designed with distribution in mind. A striking example for this is Distributed Smalltalk which was — as the author himself claims — heavily influenced by the contemporary Emerald project. Adapting the special Smalltalk programming environment and philosophy to distribution, however, posed major technical difficulties and only led to modest results.

The lowest degree of transparency is achieved by non-experimental, industry-standard solutions like CORBA and Java/RMI. While this can certainly be attributed to the inherently pragmatic nature of real-world standards, there are also more far-reaching considerations behind this which we will examine in the following section. A striking fact is, however, that research projects in the past five years, although they have been trying to improve on the deficiencies of the industry standards, have not reached the level of integration that was achieved already in the very early projects such as Emerald. One reason for this is of course, as outlined above, that they are dealing with existing languages (mostly Java nowadays) that do not lend themselves to easy distribution due to their design.

On the other hand, with implicit systems (where distribution is handled in the run-time system), transparency is simply not an issue. By their nature, distribution is completely invisible to the programmer, which makes them appealingly elegant. The big problem of these platforms is their lack of efficiency, which researchers are trying to compensate for by ever higher degrees of clever automatic optimizations. Despite this, implicit platforms have not really been put to use outside of academia yet.

Another clear trend in these developments is that the communication mechanisms become independent from the type of abstraction. While in the past, the explicit approach was invariably tied to the remote invocation model (maybe combined with object migration), and implicit systems predominantly used caching and replication techniques, there are now all sorts of "cross-over" efforts: implicit systems using remote invocation as in Jackal and cJVM (although Orca already did this ten years ago), and caching on explicit platforms as in Javanaise.

### 2.3.1 Distributed Computing: A Note on A Note

In 1994, Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall wrote an internal memorandum at Sun Microsystems, titled *A Note on Distributed Computing.* This paper has received considerable attention far beyond Sun, not least because it served as the blueprint for the development of Java's standard distribution technology, Java/RMI. In 1997, the *Note* was officially published in a Springer collection on "Mobile Object Systems". We briefly referred to this paper already in our discussion of Java/RMI (see page 24), but will have a detailed look at its line of reasoning now.

Waldo et al. argue that distributed and non-distributed programming cannot be unified, and they do so with considerable rhetoric effort.

> Every ten years (approximately), members of the language camp notice that the number of distributed applications is relatively small. They look at the programming interfaces and decide that the problem is that the programming model is not close enough to whatever programming model is currently in vogue (messages in the 1970s [...], procedure calls in the 1980s [...], and objects in the 1990s [...]). A furious bout of language and protocol design takes place and a new distributed computing paradigm is announced that is compliant with the latest programming model. After several years, the percentage of distributed applications is discovered not to have increased significantly, and the cycle begins anew.
>
> A possible explanation for this cycle is that each round is an evolutionary stage for both the local and the distributed computing paradigm. The repetition of the pattern is a result of neither model being sufficient to encompass both activities at any previous stage. However, (this explanation continues) each iteration has brought us closer to a unification of the local and distributed computing models. The current iteration, based on the object-oriented approach to both local and distributed programming, will be the one that produces a single computational model that will suffice for both.
>
> A less optimistic explanation of the failure of each attempt at unification holds that any such attempt will fail for the simple reason that programming distributed applications is not the same as programming non-distributed applications. Just making the communications paradigm the same as the language paradigm is insufficient to make programming distributed programs easier, because communicating between the parts of a distributed application is not the difficult part of that application (op.cit., pages 4-5).

The difficult part, Waldo et al. continue, lies in four distinct areas where the local and the distributed case are separated by insurmountable differences.

- *Latency.* A remote method invocation takes between four and five orders of magnitude longer than a local method invocation, and the current trends in both processor speed and network latency suggest that this will not fundamentally change in the future. As a consequence, Waldo et al. argue, not paying attention to distribution from the earliest phases of development may lead to designs with insurmountable performance problems. It must be decided right from the beginning "what objects can be made remote and what objects must be clustered together" (op.cit., page 5).

- *Memory access.* Direct memory addresses are not valid outside a single address space. Waldo et al. conclude that if local and distributed computing are unified, this means that programmers must not use address-space-relative pointers. However, this restriction could only be *enforced* if the ability to get address-space-relative pointers were completely removed from the programming language. This, on the other hand, would require programmers to learn a new style of programming, and thus give up the complete transparency between local and distributed computing.

- *Partial failure.* In a distributed system, some components, such as a network link or an individual node, may fail while others still function normally. This is different from the local case, where failures at the system level are always total. Programmers thus have two options: they can either ignore the possibility of partial failure, resulting in each partial failure being unhandled and catastrophic, or they must enhance all of their interfaces to report partial failures adequately, and make all of their code prepared for these events. This, however, would mean that local computing becomes more like distributed computing, and not the other way round.

- *Concurrency.* A similar argument can be made for concurrency (parallelism). Unlike local objects, Waldo et al. say, distributed objects must always be prepared for truly parallel invocations. In a distributed system, there is an actual indeterminacy in the order of method invocations, while in the local case, the programmer has complete control over invocation order when desired. Additionally, synchronization becomes much more difficult in a distributed system, because there is no single point of resource allocation or synchronization. Under a unified model, the burden to handle this complexity would have to be placed on all objects, not just on those where it is actually required.

None of these arguments against distribution transparency holds up under closer inspection. This is of course obvious for the argument regarding *memory access*. Since Waldo et al. wrote their paper, we have seen the introduction of main stream languages with all direct memory access constructs removed (Java and C#). This has not happened for distribution reasons, but because of the insight that direct memory access is dangerous and error-prone by itself. For example, most security incidents on the Internet are buffer-overflow attacks, caused by the availability of general address arithmetic in the implementation languages used. The widespread adoption of Java, especially among former C and C++ programmers, shows that it has not been overly difficult for them to adjust their habits. The memory access argument can thus be dismissed.

Neither is there much merit in the *concurrency* argument. Concurrent code that only works in a pseudo-parallel, time-sliced mode is simply wrong concurrent code. It is true that the actual parallelism in a distributed setting often uncovers concurrency bugs that were not apparent during time-sliced execution. This is, however, not an excuse for writing incorrect code. (It may be added that programmers are getting better at this, now that main stream languages tend to have some form of concurrency support built in.) Finally, it is definitely not the case that all objects must bear the burden of concurrency. It is quite common practice to make critical system classes non-reentrant for the sake of efficiency (e.g. Java's collection classes or the Swing toolkit). This does place an additional burden on the programmer to follow certain guidelines when using such objects in a concurrent setting, but experience has shown that this is absolutely doable.

The *latency* argument hinges, for a large part, on the assumption that there must be a static, one-way decision "what objects can be made remote and what objects must be clustered together". While this is true for platforms such as CORBA and Java/RMI, which require significant programmer effort to make an object "remote", other platforms let objects easily move around within the system, and to make an object remotely invokable can be as simple as a run-time operation that may happen at any time without programmer effort (e.g. in Doorastha and ProActive, but this was in fact already demonstrated in Emerald in 1988).

Given a highly distribution-transparent platform, it is therefore possible to select distribution strategies late in the development process, or even later, at run-time. A deeper question remains, however: *It could be that distribution requirements need to be taken into account at the architectural level.* While one design might be easy to distribute efficiently, another might not be efficiently distributable at all.

An example is the TSP program studied in the RepMI project (see page 34). Here, the currently best route is stored in a *Minimum* object accessible to all processors. If this is indeed realized as a single object, the latency of remote invocations causes the performance to degrade prohibitively. With a different design, where there is no *Minimum* object, but the minimum value is broadcast to all processors explicitly, much better performance results. *Almost the same performance,* however, was achieved by replicating the *Minimum* object transparently using the RepMI facility.

This suggests that the more sophisticated distribution platforms become, the more likely distribution requirements will not need to be incorporated at the design level. If, on the other hand, distribution does need to influence the design in a particular case, there is still no reason why the distribution aspect should not be handled as transparently as possible. (Even if I need to make different objects so that I have fewer remote invocations, there is no reason why the remote invocations shouldn't be transparent nonetheless.)

Last but not least, the issue of *partial failure* is indeed difficult. There are however answers to it, if only partial ones.

- First, there are entire application domains where partial failure simply isn't an issue. An example is scientific computing. If one of the processors of a supercomputer fails during a program run, there is usually no point in trying to mask this failure – one simply discards the program results, repairs the fault, and starts over. In a domain such as this, it is perfectly acceptable to require the entire distributed system to be fault-free, just like a single computer. If a fault does happen, the application should, if possible, be shut down in an orderly manner, but this is something that can well be done by the distribution platform, not by the application.

- Second, there are other domains where partial failure indeed cannot be ignored. Client-server computing on an Internet-wide scale is one of them. But while the problem must be acknowledged, we don't believe that the solution put forth by Waldo et al. is acceptable. In Java/RMI — the brainchild of the authors of the *Note* —, any remote operation may throw a *RemoteException* that signals a communications failure (which may also be due to a problem at the remote host itself). There are 19 immediate subtypes of this exception type, including, for example, *NoSuchObjectException*, *ServerError*, *ServerException*, *ServerRuntimeException*, *SkeletonMismatchException*, and *UnknownHostException.* Moreover, since *RemoteException* is a "checked exception", the programmer *must* provide a handler for these exceptions with every remote call. What, however, is a program supposed to do in response to an *UnknownHostException*? In the vast amount of cases, there will still be no option except to shut down the application. Sophisticated retry and replication schemes that could mask the failure, on the other hand, are too complicated to be implemented within the application logic. There is a need for infrastructures that can mask failures, but for very practical reasons, these need to be shielded from the application logic as far as possible.

Despite the decided tone of their paper's opening, none of the concrete, technical arguments raised by Waldo et al. holds up under closer inspection. It would of course not be logically sound to infer from this that their initial thesis — that local and distributed computing cannot be unified — is wrong. It means, however, that we are on our own to evaluate it.

From our understanding of the history of computing, it is a history of *abstraction.* Abstraction, in turn, means to *unify* disparate low-level concepts under new higher-level concepts. In this way, *processor registers* and *main memory locations* have been unified under the concept of *variables.* The different *increment* instructions on different processors have been unified in C's *++* operator. *Paging strategies* and *memory architectures* have been unified by *virtual memory.*

An important thing to note about these unifications is that usually, they don't result from subsuming one of the exisiting concepts under another, also existing concept. Instead, they work by establishing a *new* concept that is capable of covering all the existing ones. The implicit assumption of Waldo et al., that distribution transparency means to make distributed computing just like local computing, is therefore questionable. In fact, when local and distributed computing are unified, neither of them is likely to come out entirely unchanged. An example of this is the trend towards abstract *references* in programming languages (which can be implemented as addresses within main memory, but also as network references), along with the abolishment of unshielded address arithmetic.

It could be that the final argument in favour of distribution transparency is that people are simply *trying.* Ever since the introduction of Java/RMI, researchers have been dissatisfied with its non-transparent nature, and sought ways to improve on it. We have discussed many of these projects in the previous sections, and observed how they have actually come quite far in unifying local and distributed computing.

### 2.3.2 The Need for Static Analysis

If distribution is a low-level concept, and ought to be hidden under an abstraction layer as far as possible, the question which abstraction is the *right* one still remains. We have seen that both the explicit approach (handling distribution from within the programming language) and the implicit approach (handling distribution within the execution environment) do provide such an abstraction.

Implicit abstractions are more complete than explicit ones because they hide the *entire* distribution aspect from the programmer. Implicit Java platforms are often described as a means to consider a cluster of workstations as a *single* JVM. By contrast, explicit systems do require some work on the part of the programmer to make objects remotely invokable, to identify clusters, and to arrange their placement, e.g. via remote object creation. This amount of work is becoming less and less with more recent platforms, but it is nonetheless there.

Implicit abstraction tends to be less efficient than explicit abstraction, though. We have seen this demonstrated in the Jackal case studies (see page 47); there is also a study that compares Hyperion (page 43) and Manta (page 32) in this regard (Kielmann et al. 2001). The general observation is that the implicit platforms, at best, come *close* to the performance of equivalent programs written for explicit platforms. The reason for this is that with explicit platforms, the programmer has finer control over the communications patterns within an application. With implicit systems such as Distributed Shared Memory, there is usually more network communication and protocol overhead than would strictly be needed, due to the problem of false sharing and the fact that an abstract consistency model needs to be maintained, not one that is defined by the logic of the application.

Due to the nature of implicit abstraction, there is hardly anything that the programmer can do to optimize performance (except choosing a different object structure that maps better to the underlying system). The only way to compensate for the performance loss is therefore to perform extensive analysis both at compile-time and run-time, and use the results to enable all sorts of run-time optimizations. Orca, Jackal, and cJVM are examples of platforms that do this.

There is a higher potential for optimization with explicit platforms, because they allow finer control over an application's communication patterns. This potential for optimization, however, is usually left entirely to the programmer (unlike with implicit systems, it *can* be left to the programmer because at least aspects of the distribution are usually visible in the source code, and can be controlled programmatically).

As a result, the programmer must decide

- *where* to place objects,
- *which* objects must be remotely invokable,
- *when* to use object migration or object caching.

Having decided on these issues, the programmer must then *implement* them on the distribution platform at hand, which can be more or less work, depending on the degree of distribution transparency provided by that platform.

The *Pangaea* system that we will describe in the remainder of this thesis is meant to fill this gap. It is a distributing compiler, the frontend of which uses static analysis to find distribution strategies for a program under the remote invocation model. The backend is a code generator that can *implement* these strategies on a given distribution platform. At run-time, this is complemented with a generic migration subsystem that can move objects to other locations based on their actual communication behavior.