# Distribution and Inheritance in the HERON Approach to Heterogeneous Computing

S. Finke, P. Jahn, O. Langmack, K.-P. Löhr, I. Piens, Th. Wolff

Institut für Informatik, Freie Universität Berlin, lohr@inf.fu-berlin.de

## Abstract

*HERON is a platform for object-oriented distributed computing in an open systems environment. We try to achieve a degree of distribution transparency previously known only from special distributed programming systems, while at the same time accommodating heterogeneous, autonomous computer systems. Distributed programs are written in Eiffel. The Eiffel language system is not modified: HERON employs proxies for remote object invocation and a flexible configuration procedure for building servers and distributed programs. In addition to regular objects, two kinds of distributed objects are supported by the proxy generator: dispersed objects and objects fragmented by remote inheritance. They contribute to distribution transparency both for distributed programs and for client/server systems.*

## 1 Introduction

Distributed *object-based* systems have been the subject of many research and development projects during the last decade [Chin/Chanson 91]. The object paradigm is attractive not only for homogeneous distributed systems but also for distributed computing in heterogeneous environments. This is witnessed by the standardization work on Open Distributed Processing, ODP [ISO 90], by the effort of Unix International towards a comprehensive distributed computing architecture, ATLAS [UI 91], and by the CORBA architecture developed by the Object Management Group [OMG 92].

Object *orientation* includes *inheritance* which is usually regarded as a language feature, not to be reflected on the system level. ODP, ATLAS and CORBA do mention inheritance, but its final role is not clear yet. In particular, it is not easily seen how inheritance relates to distribution; a similar situation is found in *concurrent* object-oriented programming where the question of how inheritance relates to concurrency is not completely settled.

The object is usually considered as the natural distribution unit. But imagine a complicated application program that is distributed among heterogeneous systems because of its very heterogeneity. It may then be desirable that a superclass and a subclass need not be available on the same system, giving rise to the notion of *remote inheritance*. Investigating remote inheritance is one of the goals of a project called HERON which aims at creating a platform for developing and executing distributed object-oriented programs in heterogeneous environments.

### 1.1 A language approach to open distributed computing

The problem of how to support distributed computing can be tackled from either of two sides. The *system approach* proceeds bottom-up, building a distributed operating system or a network operating system [Tanenbaum/van Renesse 85] which provides suitable abstractions like distributed virtual memory or remote procedure call. The *language approach* starts top-down, choosing or developing a high-level programming language and striving for an efficient distributed implementation [Atkinson 91].

Although in principle both approaches should be able to arrive at the same programming environment for the application-level programmer, the different biases of the approaches have remained visible in real projects. The system approach insists that the very notion of open systems requires a language-independent distributed computing platform in the tradition of network operating systems; so it faces the problem of how to choose the right abstractions for supporting a broad range of object-oriented languages. Consequently, it has difficulties in coming up with readily usable programming environments for true high-level languages (i.e., other than C) with complete distribution transparency [Horn 89] [Sun 90] [Lyons 91] [OSF 93].



0-8186-3770-6/93 $3.00 © 1993 IEEE

In contrast, the language approach tries to support one specific language, employing a runtime system that is responsible for distribution issues [Black et al. 87] [Bennett 90] [Achauer 91] [Gunaseelan/LeBlanc 92]. Limited openness has typically been the drawback of this approach. The problem is not so much the concentration on one language; after all, mixed-language programming can be supported independently of distribution. But existing systems suffer from lack of portability. Either they offer a special distributed language or they rely on specific operating system support, or both. So they fall short of openness in the sense of ODP.

HERON follows the language approach but emphasizes use of a normal programming language and full accommodation of heterogeneity. The language chosen is *Eiffel* [Meyer 92]. We do not want to embark on a language discussion here. We consider Eiffel to be the technologically most advanced object-oriented language to date. It is available on different systems and is easily interfaced with a low-level language like C. (We do have a problem with Eiffel presently - the transition from version 2 to version 3 of the language.)

We have extended Eiffel towards a concurrent language CEiffel [Löhr 92]. CEiffel will be used, like Eiffel, for distributed applications on the HERON platform. Applications programming is distribution-transparent both in Eiffel and in CEiffel.

### 1.2 Private versus public objects

The client/server paradigm for distributed computing has traditionally likened the server to a *module* or heavy-weight object, available in one instance only; this module may be responsible for managing other, more lightweight objects which are created on demand. The object paradigm, however, prefers to treat all objects on the same footing. Although the two views do not strictly exclude each other, the former distinguishes between two kinds of objects while the latter does not make such a distinction.

We maintain that the difference between the two views is technical rather than conceptual. With respect to distribution, we may observe big differences between remote procedure call for long-lived modules and remote object invocation for transient objects [Levy/Tempero 91]. But these differences are due to the different lifetimes and availabilities of modules versus objects, not to a conceptual difference between module-based and class-based architectures.

HERON insists on one notion of object - that of Eiffel. Invocation of an object is independent of the object's location *and* lifetime. Thus, no difference is made in the program code between the invocation of a local private object, a remote private object, or a remote public object. A program does not have a notion of server; a server is just an entity containing remotely accessible public objects. Whether it is implemented as a heavy-weight process or as a passive address space or even as collection of persistent objects depends on the local operating system.

This approach differs from existing approaches where the programmer has to distinguish between small-grained objects and large-grained servers. A typical example is Distributed Eiffel, a homogeneous system running on top of the Clouds distributed operating system. The language is an Eiffel extension and distinguishes between Eiffel objects and Clouds objects (= passive address spaces).

The situation is more obscure with DCE, the OSF Distributed Computing Environment. DCE contains the Network Computing Architecture/System, NCA/NCS [Lyons 91], an RPC platform for heterogeneous environments. As NCA is independent of any particular programming language, it employs an interface description language for defining module interfaces. A module can operate as an object manager, and several modules can be combined in a server. (NCA's claim to be object-*oriented* is certainly not justified.)

A similar approach to distributed computing, although not as sophisticated as NCA, is Open Network Computing, ONC, by Sun Microsystems [Sun 90]. Both DCE and ONC are claimed to be incorporated into the ATLAS effort mentioned above. ATLAS is a long-range, ambitious endeavour towards global interoperability on the basis of object-orientation.

HERON aims at supporting truly distribution-transparent object-oriented programming, accommodating both private and public objects in a heterogeneous environment. Transparency is made possible by configuration which allows to identify both private distribution components and public servers.

### 1.3 Overview

Section 2 of this paper will give a flavour of the power of the HERON approach by means of a simple example. Section 3 deals with the problems encountered with remote inheritance and distributed objects and presents the solutions adopted. Section 4 will present our approach to configuration and binding of distributed Eiffel programs. We will discuss implementation issues and summarize the essentials of HERON in section 5.

## 2 Distribution transparency

In language-independent systems like ONC and NCS, invocation is not really transparent, and RPC programming is still an art [Corbin 91] rather than a craft. Transparency has been the hallmark of the language approach where local and remote object invocation are usually indistinguishable in a program.

A crucial point, however, is how to get hold of an object in the first place. By using a *create* operation we obtain a *private object*. The object can be either local or remote; the creation operation must know somehow whether the object is to be created remotely, and if so, where. Does the notion of distribution transparency make sense at all here? Then there is the case that we want to get hold of an already existing *public object*. Here we are not concerned with the location of the object, but some *binding* procedure must be used for obtaining a reference to it. Can this binding be distribution-transparent, or should it be? We have to investigate these questions before considering a typical example for distributed applications.

### 2.1 Creation and placement of private objects

The basic form of explicit object creation in Eiffel is the creation statement

```
!!object.init(arguments)
```

where `object` is an entity of a reference type associated with a certain class `C` and `init` is a creation routine of that class (possibly parametrized). The transparency requirement forbids any tampering with the syntax. All the information needed for deciding where the object is to be placed must therefore be derived from the given information, i.e.,

- the underlying class,
- the creation routine,
- the arguments.

We will examine these in turn.

Distribution-transparent object-oriented programming in open environments requires the usage of *proxies*, akin to stubs in traditional RPC systems. A remote object is reached via a proxy object which is an instance of a *proxy class*. The configuration procedure, to be discussed later, may or may not substitute a proxy class for the original class. If `object` is associated with the original class, a local object will be created. If not, a local proxy will be created; the `init` code in the proxy class determines the location for the object and causes the object to be created at that location. The details are again dependent on the configuration procedure.

So the programmer can influence but never completely determine the location of a remotely created object. If independent placement of two objects must not be precluded, *different classes* can be chosen for them: e.g., `C` might be used for local objects whereas a subclass `B`, containing nothing but the inherit clause, can be used for remote objects if replaced with a proxy class. Different subclasses allow for different locations.

Less obvious, although possible in principle, is having the placement depend on the creation routine. A class can have several creation routines in Eiffel; objects created by *different creation routines* may be placed in different locations (again dependent on configuration).

In many cases, the *arguments* to a creation routine naturally govern the choice of location. Transparency demands that no extra arguments are introduced just for the sake of object placement. But this is often not necessary. A typical example is the creation of a *channel* in opening a file in a distributed file system. A creation routine `open` has a string argument which represents the file name. The location of the file and thus the location of the channel is derived from the given string. The client code is not concerned with this and just contains

```
ch: Channel;
.......
!!ch.open(filename);
```

The `Channel` proxy determines the right location; in this case, the channel may even be created locally, despite the existence of a proxy. There is, of course, no way to completely automate the generation of the proxy's `open` routine. The proxy refers to a localization routine which will be plugged in at configuration time. We will come back to this later.

### 2.2 Binding of public objects

A public object can be bound to a program either statically or dynamically; with the latter technique, the program is in control, which allows for data-dependent binding.

*Static binding* is performed when a distributed program is configured. It has the effect that public objects are a priori available in the program. As opposed to module-based languages, many object-oriented languages deny the existence of static objects, and so does Eiffel. This view is inconsistent with static binding of public objects. The problem is not specific to distribution; it also arises in interfacing the language with the local operating system, in particular the I/O system.

Eiffel solves it with a trick, the so-called *once* functions. A once function remembers the value delivered upon its first invocation. Subsequent calls always produce that same value. Fortunately, this is not a burden for the I/O programmer as it is hidden behind the scenes. Eiffel has a library class `Standard_files`, and this class features once functions that deliver references to the standard I/O channels. The point is that *any* invocation of one specific function produces the *same* reference. Likewise, we can provide a class, say, `Servers` with functions that deliver references to (proxies for) public server objects. The configuration procedure is responsible for establishing the proxies and thus for binding specific servers residing somewhere in the network.

Imagine a `Servers` class that features two once functions supporting services taken from DCE and ONC, resp.: `lb` delivers a reference to an object of type `LocationBroker`; `nfs` delivers a reference to an object of type `NetworkFileSystem` (which is a *dispersed object*, see section 3). We can get hold of these objects by inheriting from `Servers`:

```
class Client
inherit Servers --features lb and nfs
feature status: FileStatus;
        object: PublicObject;
          ...
          ...
     status := nfs.status(filename);
     object := lb.lookup(objectname);
     object.operation;
          ...
end  -- Client
```

Note that `lb` and `nfs` refer to public objects, but so does `object`. The very job of the location broker is to serve as a directory for public objects. (Usually not all objects in such a directory are of the same type.)

Lookup services like that of `lb` support *dynamic binding* such as for `object` in the given example. (Caveat: this is not the notion of dynamic binding used in connection with polymorphism.) This functionality is by no means restricted to a central authority, may it be known as Location Broker, Name Server, Directory, Network Information Service or whatever. Application software can establish its own agents for managing objects, enabling newly activated programs to get hold of them. We will see this in the example given in the next section. However, no program can do *all* its binding dynamically. There always has to be at least one *anchor* object that is bound statically. Starting with an anchor we may find references to further objects and thereby perform dynamic binding.

## 2.3 A simple distributed application

Supporting on-line communication between users at different sites is a common source of examples for distributed applications. The "one-way talk" example is about the simplest possible:

> A user can declare that s/he is *listening*, i.e., willing to accept messages from other users on the screen. Other users can then send messages to that user. A user is identified by a user name which is a string.

We solve the problem as follows. There is a `Receiver` object for each listening user. A reference to this object can be found in a directory under the name of that user. The following class can be used for sending messages:

```
class Sender
inherit
 System; --features username: String
 Servers --features addressbook:
         --Directory[String,Receiver]
feature
 send(message, receiver: String) is
 do addressbook
       .lookup(receiver)
       .display(username,message) end
end  -- Sender
```

The operation `display` is featured by the `Receiver` class. It is responsible for displaying the message on the screen that is associated with the owner of a `Receiver` object. Here is the class `Receiver`:

```
class Receiver
inherit
 System; --features username: String
 Servers --features addressbook:
         --Directory[String,Receiver]
creation init
feature
 display(sender, message:
                 expanded String) is
 do print("Message from  ");
    print(sender);
    print(message) end;

 init is
 do addressbook
        .enter(username,Current) end;

 final is
 do addressbook.remove(Current) end
end  -- Receiver
```

The `Servers` class has a very simple structure:

```
class Servers
feature
 addressbook:
       Directory[String,Receiver] is
 once !!Result end;
   .....
end  -- Servers
```

The functionality of the given classes can be validated without thinking about distribution. We may even configure a centralized program that uses `Directory`, `Sender` and `Receiver` objects for testing the respective classes and their cooperation. The typical configuration, however, will be distributed, with several `Sender` and `Receiver` objects residing on different nodes and several `Servers` objects residing in one node of the network. There will be a permanent or even persistent address book, delivered by `addressbook`, and users will be able to create and destroy permanent `Receiver` objects tied to their screens. With a typical operating system, a user who is willing to accept messages will run a background program that contains `Receiver` which may be the only user-written code in that program. Additional code will of course be added by the configuration procedure (see section 4).

# 3 Distributed objects

We have considered an object as an indivisible entity until now. But as the client of an object does not care about its implementational details, the object's representation may be spread over several locations in a distributed system. The most obvious case, splitting an object represented as a linked structure of objects along the links, is supported by several systems. Other forms of *distributed objects* are less common:

- A *replicated object* as supported in the distributed programming language Orca [Bal 91] is represented as a set of identical copies residing in different locations.
- A *dispersed object*, analogous to a distributed module in DAPHNE [Löhr et al. 88], is a container object whose elements reside in smaller container objects in different locations.
- The concept of a *fragmented object* as known from SOS [Shapiro et al. 89] is a generalization of the above concepts. As opposed to those, it requires special programming and thus compromises distribution transparency.

The systems mentioned here are object-based. Object *orientation* gives rise to the phenomenon of *remote inheritance*: if a class inherits from other classes, not all the classes involved may be available in one location. This is a consequence of heterogeneity; but even in a homogeneous setting a class name may be meant to refer to a remote rather than a local instance of that class: recall that a class instance may be bound to local system objects (like `Standard_files` mentioned in 2.2). Another example is the class `Servers` from 2.3 which is remotely inherited by `Sender` and `Receiver`. So we must be prepared to split an object according to the locations of the classes involved.

Dispersed objects are also supported in HERON. Recall the `nfs` example from section 2.2. A client of `nfs` gets hold of the system via an `nfs` proxy. The proxy represents an abstract object and hides an implementation that comprises many concrete objects in different locations. These objects all have the same interface, as reflected in the proxy, but they may have different implementations in a heterogeneous environment.

## 3.1 Proxies and drivers

We have to clarify our notion of *proxy* for non-distributed objects before we tackle remote inheritance and dispersed objects. First, we have to distinguish carefully between a *class* (which is a program text) and its *name* (as given in that text). It is important not to confuse an original class with its proxy class which has the same name. We say that a class B has a proxy class B'. On the server side we have a *driver* class that represents the clients. The driver class for B is denoted B"; its name is irrelevant to this discussion.

As the proxy corresponds to the client stub in RPC systems, the driver corresponds to the server stub. The proxy is *not* a full-fledged part of a fragmented object as in SOS, nor should it be confused with a part of a dispersed object (as suggested by the different notion of proxy in [Notkin 90]).

It is quite possible that a class is available at several sites; this is the rule for standard classes. If both local and remote objects of a class are to be accessible, it is inevitable that access is mediated by the proxy class in both cases, slowing down local operation. This is the price that has to be paid for the open systems approach which prohibits the introduction of a special distributed language or compiler. However, it is not a good idea anyway to remotely pass standard objects by reference rather than by value. So there will be no proxies and thus no actual penalty in most cases.

Remote *creation* is governed by a locally created proxy. As mentioned in 2.1, the chosen creation routine and its arguments may influence the placement decision. This is a must if the class is available also locally.

Now consider the case that B is a subclass of A and both A and B are available on a remote system only. Remote invocation of a B object involves a proxy class B' that has been locally substituted for B. B' is preferably constructed as a subclass of A', the proxy of A. The situation readily generalizes to arbitrary inheritance hierarchies as long as all superclasses of the object to be created are available at the remote site. The original hierarchy is mirrored by a *proxy hierarchy.*

On the server side, the inheritance hierarchy is likewise reflected in a hierarchy of driver classes. A remotely generated creation request is directed to a central *switch*; the switch creates the required driver object which in turn creates the real object and then passes on further calls. Figure 1 shows the classes required on both sides for the remote creation of an object of class B.

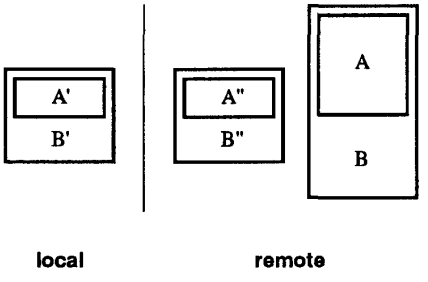


**Figure 1 - Proxy and driver classes for a remote subclass B**

Many standard programming languages, designed for centralized implementation in one address space, cause trouble when it comes to *parameter passing* in remote invocation. Eiffel is no exception. Basically, Eiffel passes *values* as arguments or results between invoking and invoked routines. A value is either a reference to an object or the object itself, i.e., the flat record of its attribute values, possibly including references. The automatically generated proxies and drivers perform the usual marshalling as known from other remote invocation systems, with a special proviso for references: passing a reference to a local object in an argument (or result) causes a driver for that object to be created on the fly; a proxy is created at the destination site, and a reference to that proxy is handed out to the receiving routine.

The problem arises with arrays and strings. To begin with, remotely accessing an array or string object is not a good idea, if only because it makes all local accesses inefficient, as explained above. So arrays and strings should normally be passed by value. But then the Eiffel semantics of passing an array or string object by "value" amounts to passing an object *descriptor*, not the object's elements. Arrays and strings do have redefined copy routines that have the desired semantics, but those do not apply to parameter passing.

Like other projects have done before, HERON trades a little bit of the original Eiffel semantics for ease of programming. When an array or string `x` is passed by value remotely, the effect is that of passing a complete copy which is denoted `clone(x)` in Eiffel. Recall the example given in 2.3, where the arguments to `display` were passed by value. The semantics of the

remote invocation

```
display(username,message)
```

is identical to that of the

local invocation

```
display(clone(username),
        clone(message)) .
```

Note that the remote version of the latter invocation would not solve the problem without the changed semantics because the `clone` invocations just create local copies.

## 3.2 Remote inheritance

Figure 1 showed a simple case of remote object creation involving inheritance. Class B was associated with a remote site where its superclass A was available as well. Now imagine that B is local and A is remote, so that B inherits remotely. Can the object be split in two parts which reside in different locations? The answer is yes, and the proxy/driver technique works smoothly for this example, as shown in figure 2.

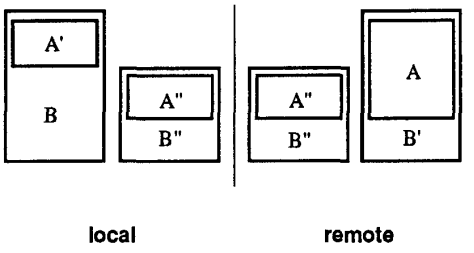


**Figure 2 - Proxy and driver classes with simple remote inheritance**

An A' proxy, inherited by B, may be used both from B and from clients of B. The corresponding driver class is the remote A". Note that B may redeclare routines inherited from A and that routines in A may refer to the redeclared ones. It is for this reason that a remote

B' is required as well. This in turn implies that we need a local driver class B". A remote B" is required, too, if only for the creation of the B' object (not its invocation). And if we want to stick with location-independent, uniform drivers we should use the same B" with parent A" both remotely and locally.

As can be seen from the symmetry in figure 2, the roles of the local and the remote site are easily reversed, only that the right-hand B" is completely superflous in this case. The pattern of proxies and drivers shown in the figure generalizes for arbitrary inheritance hierarchies.

Both remote usage and remote inheritance raise the question of whether *remote access* to attributes should be allowed. HERON does allow it, and the programmer must be aware that remote inheritance can be unexpectedly expensive. It is certainly not a good idea to have a compute-intensive routine work on remotely inherited attributes.

### 3.3 Dispersed objects

There are cases where the information contained in an object can alternatively be distributed among several objects of the *same type*. Container objects are typical examples: the elements of a given set may be contained in one `Set` object or in several smaller `Set` objects. This observation suggests that what is offered to clients as one object may actually be implemented as a collection of geographically distributed objects of the same type. This kind of object is called a *dispersed object* in HERON (see figure 3).

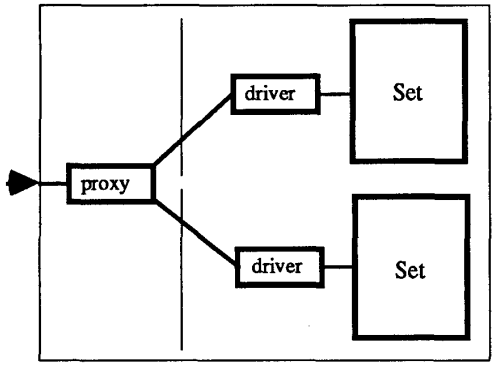


**Figure 3 - Simple dispersed object**

More precisely, a *type* means a specification. The different parts of a dispersed object will often have the same implementation, but not necessarily so. Different classes (of course with identical interfaces) may be used on heterogeneous systems. A typical example for a dispersed object with a possibly heterogeneous implementation is the `nfs` object seen in 2.2.

The crucial point with dispersed objects, as opposed to general fragmented objects, is the absence of any programmed interaction between the object parts. The proxy generator is able to handle dispersed objects. As any remote object is represented to its clients by a proxy, so is a dispersed object. In contrast to a non-dispersed object, though, there is not only one driver, but several of them, one for each object part.

Although transparency is not compromised, a little cooperation from the programmer is inevitable here. As the proxy must be able to determine the actual destination of an invocation, the necessary information must be extracted from the arguments somehow. There is no automatic way to do this because it depends on the semantics of the invoked routine. The proxy produced by the generator cooperates with a programmer-defined class that computes a location and new arguments from the given arguments of each routine. The `nfs` object, e.g., will use some extended naming scheme for identifying files on different systems. Breaking an extended name into a host identifier and a local file name is the task of the mentioned class. There is a whole spectrum of possibilities for name resolution, ranging from static association of a pathname fragment with a host to a completely dynamic approach like consulting a name server. This belongs to the realm of configuration, to be addressed in the next section.

## 4 Configuration

A distributed computation spans several heavyweight processes, mostly in different systems. Each process executes a *component* of a distributed program, also known as a *subsystem* or *virtual node* [Atkinson 91], and is called a component itself (of the distributed computation).

The purpose of *configuration* is to construct and place components, as well as to support their binding. *Construction* amounts to proxy/driver generation, compilation of classes and linking of what a local operating system sees as one program. *Placement* means making a constructed component available for loading on one or more systems, placing it in a file (or in several files). *Binding support* comprises mechanisms built into the proxy classes that support binding clients to remote public objects and dynamic creation of remote private objects at runtime.

## 4.1 Elements of the Heron configuration language

From a local point of view, a HERON component is a local Eiffel program, to be constructed independently of other components. A component may *export* some of its classes so that local instances of such classes are accessible from other components. Corresponding driver classes have to be incorporated into the exporting component. A component may also *import* classes exported from other components so that remote instances of those classes can be accessed. Corresponding proxy classes have to be incorporated into the importing component. Note that an imported class may also be present in the importing component, as mentioned in 3.1.

Describing the structure of a component can be considered as programming-in-the-large [Magee et al. 89]. A configuration language is used for this purpose. We present core elements of the language by means of examples. Consider the following text:

```
component C --component name
export x; y --exported classes
import r;   --imported class
       s@D  --imported class,
            --creation in component D
end
```

The text is processed by the *configurator* which employs the proxy generator and the Eiffel system to produce a local Eiffel program. The above component description does not show all the classes that go into the program: any class that is used or inherited directly or indirectly by `x` or `y` is included in the program. The program also contains driver classes for `x` and `y` and proxy classes for `r` and `s`.

Imagine a scenario where `C` creates remote `s` objects whose invocations may return remote `r` objects, but `C` does not remotely create `r` objects. Then information about the placement of newly created objects is only required for `s`. The above text mentions the component `D`, which is to say: `s` objects have to be created in an instance of `D` that has to be loaded from a certain file on a certain host when it is referenced for the first time.

Now, the description of `C` does *not* contain information about the file and host for `D`. `D` is an *external reference* in `C` that will be resolved by the HERON *linker*. Linking a component `C` amounts to constructing an associated file containing information for the resolution of references to other components at runtime.

The configuration language has additional constructs for locating the classes that go into a component, specifying libraries to be searched, supporting dynamic placement decisions as described in 2.1, etc. It is also possible to declare a component a *main component* by specifying a root class and a creation routine in that class. A main component will start execution with that creation routine. A distributed computation is usually started by a user loading a main component.

## 4.2 Configuration with public servers

In the above example, the first creation of an `s` object causes a fresh instance of a `D` component to be loaded. Binding to an already existing object is not possible in this way. But, as promised in 2.2, Heron does support this kind of binding which is necessary for getting access to public servers. The structure of a server is described in the same way as that of a private component, except for the keyword **`server`** instead of **`component`**. This has the consequence that the component is loaded only once and is shared by all clients.

Recall the "one-way talk" example given in 2.3. The server contains the directory of the listening users. Its configuration description is extremely simple:

```
server Servers
export Servers; Directory
import Receiver
end
```

Then we have two components, `Sending` and `Receiving`; both are main components with a root class:

```
component Sending
root Main.init    -- Main uses Sender
import Servers, Directory @ Servers;
       Receiver
end

component Receiving
root Receiver.init
export Receiver
import Servers, Directory @ Servers
end
```

An important difference between the two components is the presence of an export clause in the latter; this causes the component to wait for invocations when the `init` routine has terminated, rather than to die.

# 5 Conclusion

## 5.1 Implementation

The runtime system underneath the proxies and drivers uses the HERON transport service for inter-component communication. This service is connectionless and supports the unreliable transfer of messages of arbitrary length. The current implementation of the transport protocol uses the UDP datagram service.

The remote operations system on top of the transport service has much in common with traditional RPC protocols. The most prominent differences are due to HERON's object-orientedness. Supporting a *remote invocation* protocol is not enough. *Remote creation* requires a special protocol that takes remote inheritance and dispersed objects into account. In addition, there is a special protocol for *remote copying* and comparing.

Each host participating in HERON has a local *HERON server* with a well-known transport address. This server keeps track of the components loaded at the host and is responsible for loading new components on demand (as mentioned in section 4). Its functionality is akin to that of a combined portmapper and internet daemon in ONC. A newly created proxy gets the transport address of the component where the object is to be created from a HERON server.

The HERON server is not operational yet; components have to be loaded with system startup or by hand. Of all parts of HERON the server is the least portable. This is mainly caused by the differences in program and process management among different operating systems. Note that future operating systems may also provide better support for persistence than current ones. Application servers will be implemented as persistent objects on such a system, implying considerable changes is the HERON server.

## 5.2 Summary

The HERON approach to distributed processing clearly separates three independent concerns:

1. *Programming* - concentrates on the computational aspects of software. It is considered unnecessary or even counter-productive to use a special distributed programming language. Likewise, code written in a regular programming language should not be cluttered with distribution-related library calls.

2. *Configuration* - builds the units of distribution, called components. A component usually contains many objects at runtime, but objects may also span several components. Determining the classes that should be incorporated in a component is governed by criteria which are independent of the computational issues of 1. It is *not* recommended (although possible) to establish a fixed binding between components at configuration time.

3. *Binding* - resolves inter-component references. Components may be bound as early as at configuration time, but will usually be bound much later, e.g., upon first access. As mentioned in 4.1, there is no fixed binding strategy in HERON. There is only one common requirement, imposed by the HERON server, for all techniques that might be employed: a reference to a component must be resolved by a file name.

To conclude, HERON provides a maximum of comfort for programming and configuring distributed applications, but is deliberately open-ended with respect to dynamic binding. How a reference to a component is actually resolved is not part of the architecture. The referring component may consult a local name service that happens to be available, or it may use a standardized service such as the *trader* envisaged for ODP.

**Acknowledgement**

Thanks go to the referees and to Jacek Passia whose critical comments helped to improve this paper. Project HERON is supported in part by the German Research Council (DFG).

**References**

[Achauer 91] B. Achauer: Distribution in Trellis/DOWL. Proc. TOOLS 5, Santa Barbara. Prentice-Hall 1991

[Atkinson 91] C. Atkinson: Object-Oriented Reuse, Concurrency and Distribution: an Ada-Based Approach. ACM Press 1991

[Bal 91] H.E. Bal: Programming Distributed Systems. Prentice-Hall 1991

[Bennet 90] J.K. Bennet: Experience with distributed Smalltalk. Software - Practice and Experience 20.2, February 1990

[Black et al. 87] A.P. Black, N. Hutchinson, E. Jul, H. Levy, L. Carter: Distribution and abstract types in Emerald. IEEE TSE 13.1, January 1987

[Chin/Chanson 91] R.S. Chin, S.T. Chanson: Distributed object-based programming systems. ACM CS 23.1, March 1991

[Corbin 91] J.R. Corbin: The Art of Distributed Applications. Springer 1991

[Gunaseelan/LeBlanc 92] L. Gunaseelan, R.J. LeBlanc: Distributed Eiffel: a language for programming multi-granular distributed objects. Proc. 4. Int. Conf. on Computer Languages, IEEE 1992

[Horn 89] C. Horn, A. Donnelly: Architectural aspects of the Comandos platform. Proc. 2. Int. Workshop on Distribution and Objects. Decus, Karlsruhe 1989

[ISO 90] ISO/IEC JTC1/SC21/WG7: Basic Reference Model of Open Distributed Processing (several documents). ISO 1990

[Levy/Tempero 91] H.M. Levy, E.D. Tempero: Modules, objects and distributed programming: issues in RPC and remote object invocation. Software - Practice and Experience 21.1, January 1991

[Löhr et al. 88] K.-P. Löhr, J. Müller, L. Nentwig: DAPHNE - distributed applications programming in heterogeneous networks. Proc. 8. Int. Conf. on Distributed Computing Systems, San José. IEEE 1988

[Löhr 92] K.-P. Löhr: Concurrency annotations. Proc. 7. OOPSLA, Vancouver. ACM 1992

[Lyons 91] T. Lyons: Network Computing System Tutorial. Prentice-Hall 1991

[Magee et al. 89] J. Mage, J. Kramer, M. Sloman: Constructing distributed systems in Conic. IEEE TSE 15.6, June 1989

[Meyer 92] B. Meyer: Eiffel: the Language. Prentice-Hall 1992

[Notkin 90] D. Notkin: Proxies: a software structure for accommodating heterogeneity. Software - Practice and Experience 20.4, April 1990

[OMG 92] Object Management Group: The Common Object Request Broker: Architecture and Specification. OMG Document #91.12.1

[OSF 92] Open Software Foundation: DCE Application Development Reference. Prentice-Hall 1993

[Shapiro et al. 89] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, C. Valot: SOS: an object-oriented operating system - assessment and perspectives. Computing Systems 2.4, December 1989

[Sun 90] Sun Microsystems: Network Programming Guide. No. 800-3850-10, 1990

[Tanenbaum/van Renesse 85] A.S. Tanenbaum, R. van Renesse: Distributed operating systems. ACM CS 17.4, December 1985

[UI 91] Unix International Europe: UI-ATLAS Distributed Computing Architecture. September 1991