Distrib. Syst. Engng **6** (1999) 3–12. Printed in the UK PII: S0967-1846(99)04179-0

Jonathan: an open distributed processing environment in Java

Bruno Dumant, François Horn, Frédéric Dang Tran and Jean-Bernard Stefani

CNET/DTL/ASR, France-Telecom/CNET, 30–40 Rue du General Leclerc, 92794 Issy Moulineaux, Cedex 9, France

E-mail: Bruno.Dumant@cnet.francetelecom.fr

Received 11 February 1999

Abstract. This paper describes a minimal and modular Object Request Broker (ORB) framework from which it is possible to build highly flexible ORBs supporting the introduction of arbitrary binding mechanisms between interacting objects. We show that such a framework consists essentially of extending the Java notion of object reference to make it distributed. Jonathan is a Java implementation of such a framework, featuring a CORBA 2.0 'personality' and several different binding factories. It could be easily extended with new binding factories and personalities (e.g. a RMI personality) or scaled down to fit particular needs.

1. Introduction

The success of Object Request Brokers (ORBs)† in telecommunications essentially depends on the possibility to adapt them to the specifics of telecommunication systems, and in particular to the support of interactive multimedia services.

This implies that ORBs should be uniformly available across traditionally separate systems, ranging from low-end network equipment such as routers and cross-connects to high-end information processing intensive nodes such as those supporting network operation and management functions and operators' information systems.

Another strong requirement for telecommunication ORBs is their support of various binding and interaction models.

The term *binding* should be understood as both the process of associating or interconnecting different objects of a computing system according to a specific communication semantics, and as the end result of this process. Binding implies setting up an access path between objects, which in turn typically comprises locating objects, checking access rights and setting up appropriate data structures to enable communication between objects. Even in the standard client–server case, there is a wide variety of communication semantics that reflect different application requirements. For instance, servers may be persistent, replicated, or may use caches with various consistency policies to improve performance, availability, etc. Other forms of bindings include, for example:

- Multimedia stream bindings with various communications topologies: one-to-many, many-to-many, etc.
- Group bindings, with various communication semantics and dependability properties.
- QoS-constrained bindings whose life cycle and resource multiplexing policy are controlled by the application of e.g. a client–server binding over a dedicated ATM connection with guaranteed bandwidth.

Lastly, telecommunication systems are typically open, real-time systems, and it is crucial that ORBs achieve real-time behaviour and performance.

In this paper, by real-time systems we understand systems whose users are allowed to specify timeliness and throughput quality of service (QoS) requirements and to obtain guarantees about the fulfillment of these requirements. The nature of guarantees provided may vary from best-effort (where the system provides no quantitative guarantee of how well or how often it will meet application QoS requirements) to deterministic (where the system guarantees that application requirements will be strictly met throughout the lifetime of the application).

An *open* real-time system is a system whose set of supported applications is not known beforehand and which may vary over time. The main consequence of this is that strong (i.e. stronger than best-effort) guarantees can be provided only via some form of run-time admission control. Providing timeliness and throughput guarantees in an open distributed real-time system is a formidable task; to the best of our knowledge, there is currently no comprehensive and systematic approach to the problems at hand. Even in the case of relatively simple communication facilities, such as a channel supporting communication between a pair of objects, deriving e.g. admission control tests for guaranteed end-to-end delay bounds involves recent advances in scheduling theory. Also, it seems unlikely that a single scheduling policy or even task model will be applicable in all cases and all

† The term 'ORB' should be understood in a loose way throughout this article as a distributed object-oriented system, not necessarily a CORBA compliant platform.

application domains. For these reasons, a real-time ORB should, as a general principle of separation between policy and mechanism, refrain from embodying any particular resource management policy. Instead, a real-time ORB ought to be flexible enough to accommodate such different policies and should provide direct control of system resources such as processors, memory and communication to applications.

Architectures like CORBA or JAVA-RMI do not seem to be particularly adapted to telecommunication systems. A large part of these architectures remains monolithic and disallows adapting or extending the internal machinery short of resorting to proprietary extensions. In terms of computing and network resources, they offer very little control of how resources are allocated and multiplexed. As a result, it is not clear how to provide the adaptations required to support multimedia and real-time applications or to scale down these architectures to fit resource-constrained devices such as portable phones or interactive TV set-top boxes.

The work pursued in the context of the ACTS ReTINA project [13] is to specify and implement a flexible ORB architecture whose machinery is exposed to the systems or application programmer.

The prime characteristic of this architecture is the ability to plug arbitrary forms of binding policies between objects beyond the implicit binding model for client–server interactions assumed by standard architectures like CORBA or RMI. This goal has been achieved by designing a minimal ORB kernel whose role is to provide a generic environment that arbitrary *binding factories* can use to create and manage specific bindings.

Using this approach, a CORBA compliant ORB can be built as a particular 'personality' (i.e. a set of APIs and language mappings), thus decoupling the specifics of the CORBA API from the personality-independent kernel interface. This is also true for a non-CORBA personality like Java-RMI.

The aim of this paper is to provide a clear description of the ORB kernel architecture, and to show how a CORBA ORB can be built on top of it. To illustrate this, we shall use the Jonathan Distributed Processing Environment (DPE), developed in Java at CNET, that implements the specifications of the ORB kernel and a CORBA personality.

The ReTINA architecture also features a communication framework allowing the modular construction of protocols (suited for RPC or stream styles of interactions) and the reuse of protocol components between binding implementations. This framework, implemented in Jonathan, is very similar to the *x*-kernel [11] framework, and the reader is referred to [2] for a more detailed description.

Lastly, the ReTINA architecture proposes a resource framework, featuring a number of abstractions to manage real-time resources, that provide direct control of system resources to applications. The resource framework has not been implemented in Jonathan since it requires accessing the underlying system-level resources, and this is not provided by the current implementations of the Java virtual machine.

This paper is structured as follows: section 2 describes the ORB kernel architecture, and how it is implemented in Jonathan; section 3 describes in more detail how a CORBA ORB may be built on top of the Jonathan kernel, and in particular how a binding factory able to create RTP multicast channels can be built in this context; section 4 compares the performance of Jonathan with that of commercial Java and C++ ORBs. Finally, section 5 concludes by providing more information about the current status of the Jonathan DPE, comparing with related works, and sketching future research.

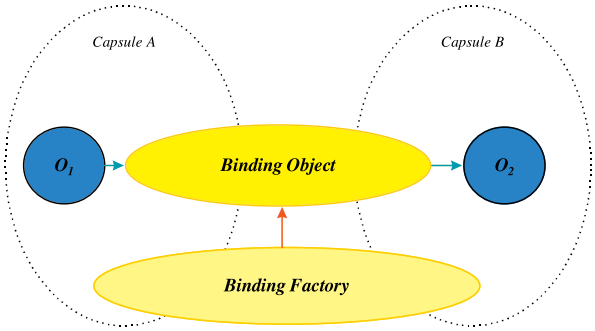

Figure 1. A point-to-point binding.

2. A flexible ORB framework

2.1. Architectural model

At least three problems must be addressed in an object method invocation: referencing the target, checking its type and, finally, accessing it and interacting with it. The role of an ORB is to provide solutions to these problems in a distributed setting. Since ORBs may operate over heterogeneous platforms and programming languages, there is a need for an abstract object model that can be mapped on specific platforms and languages; Jonathan (like CORBA) is based on the Reference Model of Open Distributed Processing (RM-ODP [6–9]):

- An **object** is an entity containing (encapsulated) information and offering services; RM-ODP objects may be of arbitrary granularity (from one byte to a telephone network, etc).
- RM-ODP objects can only interact at **interfaces**: informally, the interfaces of an object are its access points, which means that all the interactions of an object with its environment must occur at one (and only one) of its interfaces.

2.1.1. Binding objects and binding factories. To address the interaction problem, RM-ODP introduces the notion of *binding object*. Two objects—say O_1 and O_2—may interact in two different ways: either object O_1 directly invokes an operation on O_2, or it invokes the same operation on a *binding object* whose role is to transmit the invocation to O_2 and to return a result if necessary (cf figure 1). In the first case, both objects must belong to the same addressing space (also named *capsules*); in the second case, the interacting objects may belong to distinct addressing spaces.

Binding objects are usually composite objects, distributed over several capsules. Binding objects encapsulate the full end-to-end computational binding between interacting objects, including communication resources, protocols, stubs, etc. In our model, bindings have a type, the type of

a binding representing the protocols used, possible quality of service constraints, or any other kind of binding property. Note that binding objects can represent not only client–server bindings, but any kind of binding including e.g. those described in the introduction.

Bindings are constructed by special entities named *binding factories*. Binding factories have two main roles:

- Create and manage interface *identifiers*; identifiers are introduced in the next section, and this role of binding factories is further discussed in section 2.2.3.
- Establish and manage bindings of a specific type†; this role is further explained in section 2.4.

In its current version, the Jonathan DPE provides three binding factories of different types, two of which create client–server bindings (one uses the IIOP protocol, the second a simpler remote invocation protocol), and the third RTP multicast bindings.

2.1.2. Types and references. To address the typing and referencing problem, RM-ODP introduces the notion of *interface reference*. An interface reference may be seen as a generalized Java object reference. Interface references are characterized by two elements:

- A type:
  Like Java references, interface references have a type, whose role is to allow the creation of safe bindings.
  There is no universal type system, and different type systems (and type conformance rules) may be used for different bindings. In our model, type systems are related to *personalities*: a personality defines the type and binding type systems used by binding factories belonging to it‡. For instance, CORBA is a personality: it defines types by the means of IDL declarations, and binding types by the means of *profile ids*. It is then possible to build various CORBA binding factories, using different protocols, that will use this type system: for instance, all the binding factories provided with Jonathan use the CORBA type system.
  However, an interface reference may have to be manipulated in different personality contexts. We can imagine a capsule featuring two binding factories, the first building CORBA IIOP bindings, the second building RMI bindings, each of them having to manipulate the same interface reference. To make this possible, the interface reference associated type must be defined in a reference, personality-independent, type system. Each personality present in the capsule must provide ways to translate types from its own type system to the reference type system, and conversely (cf section 2.3). In our model, the reference type system is defined by the *ORB kernel*. The same holds for binding types: the ORB kernel defines a reference binding type system, and each personality has to define its own binding type system with the necessary translation functions.
- A set of *identifiers*:
  An identifier is created by a binding factory before the interface reference is sent out of its originating capsule§, and may be used by another binding factory of the same type in a different capsule to unambiguously designate the referred interface.
  An identifier is thus always associated to a given binding type. An interface reference may have identifiers of different types, corresponding to different ways to designate the target interface.
  An identifier is the distributed counterpart of the memory address contained in a Java object reference. Interface references generalize this notion in the sense that *several* identifiers may be associated with one interface reference, and the process of resolving an identifier into an interface residing in a given address space may be much more complex than dereferencing a pointer.

Thus, the three problems stated above (distributed referencing, typing and access) are represented by different abstractions: interface references for the referencing problem, personalities for the typing problem, and binding factories for the access problem. The next sections illustrate these points in the case of Jonathan.

2.2. Distributed references

The first thing an RM-ODP compliant ORB has to define is its own way to map the notion of interface, in a specific implementation language.

When an object oriented language is used, an RM-ODP object is usually *implemented* by a collection of language level objects. Some of these language level objects correspond to access points to the corresponding (abstract) RM-ODP object and as such constitute implementations of its interfaces. One way to concrete the RM-ODP notions of object and interface is thus to see an RM-ODP object as a collection of language level objects, and an interface of this RM-ODP object as a given accessible element of the collection.

In CORBA+Java, or RMI, the objects implementing RM-ODP interfaces have a special type (`org.omg.CORBA.Object` for CORBA, `java.rmi.RemoteInterface` in RMI). Likewise, in Jonathan, an interface is represented by a Java object of type `Interface`:

```
package jonathan.kernel;
public interface Interface {}
```

A reference (in the Java sense) to an object of type `Interface` represents an interface reference.

2.2.1. Surrogates. According to the ODP and CORBA computational models [8, 10], interfaces are passed by reference in invocations. As long as the interface reference remains in its original capsule, there is no need for extra information; its type is defined by the actual Java type of the interface, and there is no need for specific identifiers. But as soon as an interface reference is sent out of its

† In the following, we shall identify the type of bindings created by a binding factory and the type of the binding factory itself.

‡ Plus a set of APIs and language mappings.

§ This is the first invariant that must be maintained by any ReTINA environment in order to maintain the integrity of the associations between identifiers and interfaces, cf section 2.2.3.

capsule, we need a way to associate type information and identifiers with it more explicitly. Since an interface reference may be associated with identifiers created by various binding factories, this method must be normalized to allow interoperability. To achieve this, we introduce the `Surrogate` type:

```
package jonathan.kernel;
public interface Surrogate
extends Interface {
  Type _type();
  IfRef _ifRef();
}
```

A surrogate is an interface reference, containing the information needed to manage distribution. It is characterized by two elements:

- a type;
- an IfRef.

The type information corresponds to the local type of the surrogate†.

The IfRef is the container for the identifiers of the designated interface and its type. IfRefs in Jonathan are structured as follows:

```
package jonathan.kernel;
public class IfRef {
  Type type()...
  Key key()...
  BindingDataSet bindingDataSet()...
}
```

An IfRef contains type information about the designated interface, a *key*, and a set of *binding data*. Identifiers are not represented directly in an IfRef, but by all the possible ⟨key, binding data⟩ combinations:

- The key identifies an interface in the context of its originating capsule; a key is thus a context-dependent name that need not be directly interpretable in the context of the capsule where the caller of operation `key()` resides: it is an identifier whose domain of validity is restricted to the interface's originating capsule; the key can typically be used by any binding factory in conjunction with a hash table to access interfaces in a capsule.
- Binding data are added to an IfRef by binding factories, and contain binding factory specific information. Typically, for remote invocation protocols based on TCP/IP (like Java RMI or CORBA IIOP), binding data consist of a host name and a port number, in which case the binding data directly designate a specific address space; however, it is possible to imagine very different kinds of binding data.

Structuring identifiers as ⟨key, binding data⟩ pairs allows a factorization across different binding factories of the same key for identifying a given interface. In the Jonathan DPE, the kernel maintains a hashtable, associating keys and interfaces, that any binding factory can use. Note that this does not prevent different binding factories from using their own (equivalent of) keys (which should then be hidden in the binding data).

Both IfRefs and surrogates represent interface references, and it would have been possible (though not necessarily convenient) to have only one type for both. In fact, they are not used at the same level:

- Surrogates are used at the application level to support distribution transparency. In particular, a user expects a surrogate to be of the same type as the interface it represents: if an interface is of type `A`, the surrogate type of interest should be described as the generic type `Surrogate<A>`, and be a subtype of `A`. The surrogate interface is in particular implemented by stubs.
- IfRefs are lower level constructs. Each interface reference appearing in an invocation must be marshalled in some concrete on-the-wire representation; IfRefs constitute abstractions of such representations, allowing access to the different elements of an exportable interface reference.

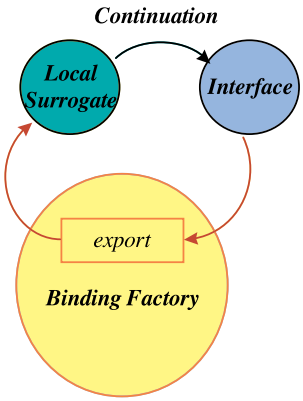

Figure 2. Exporting an interface.

2.2.2. Creating surrogates. Each binding factory must be able to create a surrogate—i.e. a distribution-ready interface reference—for a given interface. The simplest form‡ of this operation is the following (cf figure 2):

```
LocalSurrogate export(Interface itf);
```

The `export` operation takes an interface as parameter and returns a surrogate for this interface. The IfRef of this surrogate contains binding data added by the invoked binding factory§.

The returned surrogate is of type `LocalSurrogate`: The specificity of local surrogates is to be equipped with a *continuation* pointing to the designated interface. This continuation signifies the relation between the IfRef held by the local surrogate and the represented interface.

† In Java (or C++), the type of an object reference is not necessarily the most refined type of the designated object, but must be a supertype of that object. Likewise, the type of a surrogate is not necessarily the most refined type of the interface it represents, but must be one of its supertypes.

‡ More sophisticated `export` methods could manage e.g. offered QoS descriptions or, more simply, type information (like in Jonathan 1.2). Java RMI only provides the simplest `export` form as the `exportObject` method of `java.rmi.server.UnicastRemoteObject`.

§ It may also contain other binding data, if the same interface has been previously exported by other binding factories.

```
package jonathan.kernel;
public interface LocalSurrogate
extends Surrogate {
  Interface _continuation();
  void _continuation(Interface itf);
}
```

2.2.3. Maintaining reference chains. One essential role of binding factories is to make sure that when an interface reference is sent as a parameter in a remote invocation, the invoked object will receive an interface reference that references the right interface.

The above condition may be decomposed into two simpler invariants that binding objects must maintain:

(1) A binding object must marshall an interface reference i into an appropriate IfRef† when that interface reference is passed as an argument to an outgoing invocation.
This means that either i is a surrogate—and its IfRef is used—or it is not (it is simply of type `Interface`), and it must be exported to a binding factory, so that an IfRef representing it is created.

(2) A binding object must unmarshal an IfRef into an appropriate interface reference on receiving an incoming invocation bearing that IfRef.
This interface reference is either a surrogate bearing the received IfRef‡, or the interface reference represented by the IfRef (if it can be proved that it is local).

If these invariants are maintained, every *non-local* surrogate s can be associated with a local surrogate l (possibly located in another capsule) bearing the same IfRef, and the interface represented by s is the continuation of l, or the interface designated by that continuation. Since nothing prevents surrogates from being explicitly exported to binding factories, the continuation of a local surrogate is indeed not necessarily the effective implementation of the designated interface, but may be one of its surrogates.

The above invariants ensure that every surrogate belongs to a *reference chain* (cf figure 3) uniquely identifying a given interface.

A reference chain is constituted by a sequence of interfaces

$$s_n \rightarrow s_{n-1} \rightarrow \cdots \rightarrow s_1 \rightarrow i = s_0$$

such that $i = s_0$ is an interface that is not a surrogate, and $(s_j)_{j\in\{1,\ldots,n\}}$ are surrogates. Interface i is the target of the reference chain, i.e. it is the interface that is designated by the chain.

Each surrogate s_j in the chain is such that either:

(1) $j > 1$ and its continuation is s_{j-1} or,
(2) $j > 1$ and s_j holds the same IfRef as surrogate s_{j-1}.

These two properties directly correspond to the two invariants mentioned above.

† The exact format of encoded IfRefs is protocol dependent. For instance, if IIOP is used, IfRefs are marshalled as Interoperable Object References (IORs).

‡ If a surrogate bearing the same IfRef exists in the capsule, it can be used, otherwise a new surrogate must be created.

2.3. Distributed typing

Several type systems may have to cohabit in the same capsule, and the ORB kernel has to define a reference representation of types (and binding types) that different personalities will have to translate into their own type system.

Jonathan defines a `Type` type:

```
package jonathan.kernel;
public class Type {
  public Type(String str_type)...
  public boolean is_a(Type type)...
  public String to_string()...
  ...
}
```

As Jonathan is written in Java, it is natural that the `Type` objects represent Java types. That is why a `Type` object may be constructed from the string representation of a Java type, i.e. a Java scoped name (like `"jonathan.kernel.Interface"`). The main operation on types is the `is_a` operation that tests whether the target type is a subtype of the argument.

Jonathan also defines a `BindingType` type.

```
package jonathan.kernel;
public class BindingType {
  public BindingType(int id)...
  public int encode()...
  ...
}
```

Binding types in Jonathan are simply represented by integers.

2.4. Distributed access

Binding factories are responsible for the creation and management of bindings of a given type. To create bindings (i.e. instantiate binding objects), they use the identifiers of the interfaces to bind, and in particular the binding data of the appropriate type: binding factories implement binding functions that can resolve such identifiers into effective access chains to the remote interfaces, thus enabling interaction.

2.4.1. Implicit binding. When an IfRef is received in a capsule, and if it cannot be proved that the IfRef corresponds to an interface residing in the capsule, a surrogate must be created. This surrogate may be directly invocable, like in the classical *implicit* binding case in ORBs or distributed object systems; in this case, a communication channel is set up either at the creation of the surrogate, or at the first invocation.

This surrogate must know which binding factory to invoke to set up the communication channel, but there is no reason to impose that the binding factory used to create the current binding (used to perform the invocation) should also be able to bind the new surrogate: the received IfRef may not contain an appropriate identifier.

The task of finding the appropriate binding factory is devoted to the ORB kernel. The kernel maintains a table of the binding factories in the capsule that can be used to implicitly bind surrogates. As each of these binding factories is associated with a binding type, the kernel simply needs to

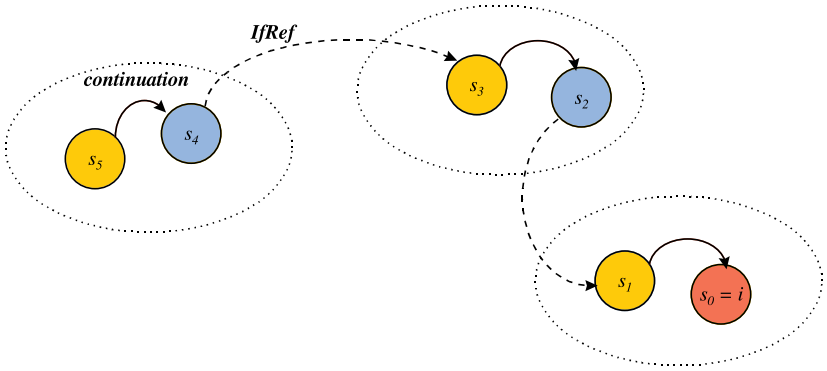

Figure 3. Reference chains.

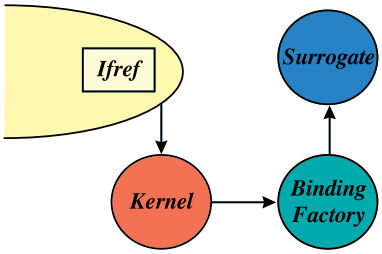

Figure 4. Implicit binding.

know the types of the binding data present in an IfRef to find—if possible—a binding factory able to implicitly bind the corresponding surrogate.

In Jonathan, this is implemented by a method `getInvocable` that takes an IfRef and a type (represented as a string) as parameters, and returns a surrogate (cf figure 4):

```
Surrogate getInvocable(IfRef ir,String type);
```

This method is implemented both by the kernel and by the binding factories. When an IfRef is unmarshalled in a binding object, the binding object invokes the `getInvocable` method on the kernel, that will forward it to an appropriate binding factory, if any:

- If a suitable binding factory can be found, it instantiates a surrogate (usually a stub) that is returned to the invoking binding object.
- Otherwise, the kernel returns `null`. In this case, the binding object that invoked the `getInvocable` method must create a surrogate that is not a stub, but simply an holder for the IfRef: this surrogate cannot be used directly in interactions, but still refers to the right interface, and may be used to *explicitly* establish a binding involving it.

This discussion shows that our framework makes a clear distinction between stubs and surrogates:

- The only role of surrogates is to *reference* remote interfaces, not to give access to them.
- Stubs belong to binding objects, and their principal role is to allow an *access* to a remote interface. Since accessing an interface implies that it is designated, stubs *are* surrogates, but the converse is not true.

2.5. Explicit binding

If the kernel cannot find any suitable binding factory to implicitly bind the surrogate, explicit binding must be used. Binding factories may define a `bind` operation that is used to explicitly create bindings. The form of this operation may be very different from one binding factory to another. In its general form, it takes as parameters a set of interfaces to bind and QoS constraints that need to be guaranteed, and returns a control interface on the created binding object. Note that our framework does not constrain the possible binding scenarios, and that any one can thus be implemented (including third-party binding, cf [3]).

Explicit binding is used by the `Stream` binding factory provided with the Jonathan DPE, to create multicast RTP channels (cf section 3.3).

3. Building a CORBA ORB on top of the Jonathan kernel

Building a CORBA 2.0 ORB on top of the Jonathan kernel means providing:

- A CORBA *personality*, namely the standard set of CORBA APIs (ORB, Object, etc) and an IDL to Java mapping (defining in particular the mapping between the CORBA type system and the Jonathan reference type system).
- CORBA binding factories, and in particular an IIOP binding factory.

3.1. The CORBA personality

The CORBA† personality is implemented as a set of four packages containing the standard CORBA definitions. The only extensions of the CORBA standard can be found in the implementations of `org.omg.CORBA.Object` and `org.omg.CORBA.ORB`.

- `org.omg.CORBA.Object` extends the `jonathan.kernel.Surrogate` interface. It is natural in Jonathan, since the CORBA ORB class must be able to implement the `object_to_string` method that turns an object into

† Jonathan only implements a subset of CORBA 2.0. It is not complete since CORBA features like the `Any` type or the Interface Repository are not implemented. However, the implemented subset is rich enough to implement non-trivial applications.

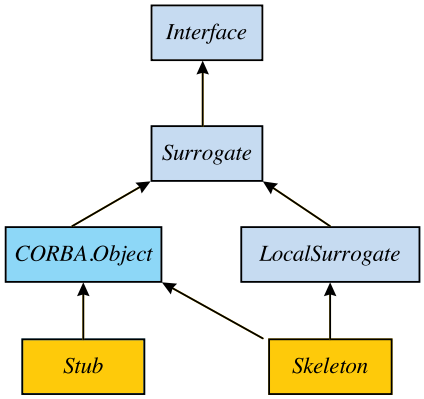

Figure 5. Interface types hierarchy.

a stringified IOR. We have seen above (cf footnote '†' on p 7) that IORs are a mere encoding of IfRefs, and consequently, the existence of the `object_to_string` method implies that it must be possible to associate an IfRef to any `org.omg.CORBA.Object`, which means that CORBA objects are surrogates.

- As stated above, the role of a personality is also to provide a bridge between the personality type system and the reference type system to binding factories. This is expressed in Jonathan by some additions to the `org.omg.CORBA.ORB` class:
  - method `corba2java` turns a CORBA type, described by its interface repository representation, to the corresponding Java type string representation;
  - method `java2corba` performs the inverse transformation.

  The CORBA personality needs not provide translation methods for binding types since binding types are represented by integer constants both in CORBA ('profile ids') and Jonathan.

The Jonathan implementation provides a *stub factory*. In the ReTINA architecture, a stub factory is simply an object whose role is to instantiate stubs and skeletons. As it is a *CORBA* stub factory, it comes with an IDL compiler that generates code for stubs. An interesting property of the current Jonathan stub factory is that it is independent of the underlying protocols, and of the binding factories that may use it. This is how all three binding factories provided with Jonathan (see section 3.2) can use the same stubs, even though they are based on different protocols. A simple smart stub mechanism is also provided.

Like in standard CORBA implementations, the generated stubs are CORBA objects, and thus surrogates. The generated skeletons are also CORBA objects, and implement the `LocalSurrogate` interface. This way, the local surrogate abstraction may be introduced, at no cost, to the system: when an invocation is received from a remote capsule, there is no unnecessary indirection due to the existence of a local surrogate.

The resulting interface types hierarchy is depicted in figure 5.

3.2. Binding factories

CORBA Object Adapters can be understood as server-side interfaces to binding factories. In particular, the `create` operation on the `BOA` is a specific form of `export`.

Jonathan does not provide a standard `BOA` interface, but three binding factories. As we have seen, binding factories are responsible for the creation and management of bindings, and rest on specific libraries (protocols, stub factories, personalities) to achieve this.

Jonathan provides five *independent* protocol packages, built following the binding and communication framework sketched in the introduction: the `protocols.tcpip` package represents the TCP/IP protocol (it is built on top of the `java.net` package), the `protocols.multicastip` package lets the user open multicast channels, the `protocols.rtp` package is a limited implementation of the Real Time Protocol, `protocols.giop` provides an implementation of CORBA's GIOP protocol, and `protocols.miop` provides an implementation of a simpler invocation protocol.

In Jonathan 1.2, all three binding factories use the same personality and stub factory, but use different protocols.

- The IIOP binding factory implements the IIOP protocol, by stacking the GIOP protocol on top of TCP/IP. It allows implicit binding.
  The binding data used by the IIOP binding factory implement the `BindingData` interface, and extend the `org.omg.IIOP.ProfileBody` class; consequently, they contain a host name, a port number, and a key (represented as an array of bytes encoding the IfRef key).
- The JIOP binding factory stacks the MIOP protocol on top of TCP/IP. It is more experimental, and could be modified so that, for instance, two-way operations use TCP connections while one-way operations on the same interface use UDP connections.
- The Stream binding factory stacks the RTP protocol on top of Multicast IP. It is further explained with an example in the next section.

3.3. A CORBA stream binding factory

In this section, we shall describe, using an example, how the stream binding factory provided in Jonathan works, and how it can be used.

The stream binding factory creates objects representing multicast RTP channels. These objects are manifested to the users by surrogates containing a specific multicast address (cf figure 6), and may be dynamically extended to bind different interfaces.

The type of the channel is defined by the user thanks to an IDL declaration. For instance, the following declaration can be used:

```
// IDL
typedef sequence<octet> frame;
interface DataChannel {
  oneway void send(in frame data);
}
```

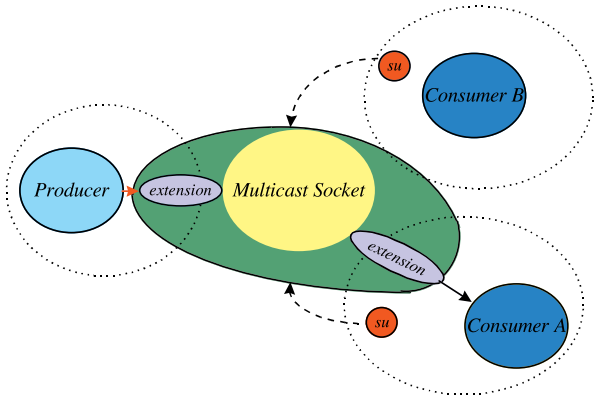

Figure 6. A stream channel.

This declaration defines a `DataChannel` type. Only one-way operations are considered, and trying to use two-way operations results in a runtime exception.

The binding factory can create multicast channels of a given type, simply by providing the type, a class D IP address, and a standard UDP port number; the following invocation on `StreamBF` (the stream binding factory) returns a surrogate of the multicast channel specified in the invocation:

```
// Java
DataChannel channel = (DataChannel)
  StreamBF.getInvocable("DataChannel",
  "224.10.0.0",9000);
```

The IP address and the port number are stored as binding data in the returned surrogate's IfRef. This surrogate is implicitly bound: a producer can immediately emit data in the channel by invoking a method on the returned object:

```
channel.send(new byte[1024]);
```

At the first invocation on the surrogate, the stream channel is extended. The extension is represented in figure 6 by an object in the capsule of the producer. The role of this object is to compose RTP packets with the data sent, and write them to the multicast channel.

Since the object representing the stream channel is a surrogate, it can be used as a parameter in any CORBA invocation (even if another binding factory is used). For instance, it can be registered in a trader under the name `"data_channel"`.

A consumer must be an implementation of the `DataChannel` interface. To receive data, a consumer must first retrieve a reference on a channel (e.g. by invoking the trader where the channel is registered), and then explicitly get bound to the channel. This is achieved by the `bindConsumer` method of `StreamBF`:

```
DataChannel consumer = new DataConsumer();
DataChannel channel =
  Trader.get("data\_channel");
StreamBindingCtl ctl =
  StreamBF.bindConsumer(consumer,channel);
```

The `get` invocation on the trader returns a surrogate of the multicast channel. At this point, the situation is that of consumer B in figure 6: there is a surrogate of the stream channel in the capsule. The `bindConsumer` invocation extends the stream channel with an object (see the situation of consumer A) that reads RTP packets from the multicast socket, and recomposes the original message before forwarding it to the consumer.

`ctl` is a control interface of the binding between the channel and the consumer. It comprises a `release()` method that can be used to stop forwarding data from the channel to the consumer: the `release` method destroys the extension created by `bindConsumer`.

4. Performances

Jonathan has not been developed with performance in mind. However, the performance tests we have conducted show that this implementation compares well with commercial CORBA implementations. This in particular means that the modular and flexible architecture we propose can be implemented at no significant cost in terms of performance.

To compare the different ORBs, a test bed consisting of two nodes connected by a dedicated network has been used. The two nodes are similar (same hardware, operating system and ORB). One of them runs a client, the other runs a server. The tested ORBs are:

C + + : OrbixMT 2.3, Visibroker C + + 3.0,
Chorus ORB 5.0, M3 2.2

Java : OrbixWeb 3.0, Visibroker Java 3.0, Jonathan 1.2

It is beyond the scope of this paper to describe precisely the different experimentations. We will simply give here some figures extracted from the results of one experiment dealing with invocation duration.

A first series of tests deals with two-way invocations with no or few simple parameters (up to five arguments of basic types, or a small string, or a `struct` containing simple members). From these tests, we can infer the following orders of magnitude for simple invocation times (in ms):

Visi C++	Chorus	M3	Visi Java	Jonathan
4.0	4.5	10.5	10.5	12.5
OrbixMT	OrbixWeb			
13	21.5			

These figures have been obtained using 70 MHz Sparc 5 workstations with 64 Mb RAM, i.e., slow machines, connected by a 10 Mb s^{-1} Ethernet network, and using IIOP as invocation protocol. Much better results may be obtained by more efficient configurations. For instance, the invocation time obtained for Visibroker Java on a 200 MHz bi-pentium pro NT4 machine is only 0.6 ms, the client and server using a different processor†. These results show that a performance-oriented C++ implementation is really faster than the best Java implementations. However, this may or may not be significant for a given application, depending on the importance of the communication time w.r.t. computation time, the network configuration, etc. No Just-In-Time (JIT) compiler was used for the Java tests. Using a JIT compiler does not greatly affect these results: an important part of the invocation time is spent in the network layers and the Java/C interface. However, it significantly reduces the marshalling

† Note that the difference in the results is not only related to the speed difference between the machines used, but also to the absence of network latency in the second configuration.

time, and using a JIT compiler becomes really important if the network latency is small, or if there are many or big parameters.

These figures should really be understood as orders of magnitude: sending parameters takes of course more time than sending no parameter. In our tests, we have not observed variations of more than 25% w.r.t. these figures. A typical variation is 5% for a C++ ORB, and 20% for a Java ORB. This difference in the variations is essentially due to the fact that, for instance, unmarshalling a simple type or a string from an IIOP stream when the server and client machine architecture are the same is a very simple operation in C++ (essentially a cast); in Java it always requires byte-level manipulations; it is even worse with strings in Java, since the underlying array of chars must be accessed element by element (or copied), and the characters unmarshalled one at a time. The same kind of argument holds for marshalling.

The performance of Jonathan is quite good if we compare it to the other Java ORBs: it is close to Visibroker Java, and 40% faster than OrbixWeb.

Another test dealt with sending sequences of octet values, which may represent sending a data stream. The results we obtained show that the time to send a sequence grows more or less linearly with the size of the sequence: the performance of an ORB may thus be expressed in bytes ms^{-1}.

Visi C++	Chorus	Jonathan	Visi Java
1432	1333	1105	892
OrbixWeb	OrbixMT		
714	365		

Jonathan appears here as the best Java ORB, which shows that a modular architecture is not necessarily synonymous with bad performance, and close enough to the best C++ implementations.

We have also tested the scalability of ORBs, by trying to run and interact with up to 100 000 servants in the same process. Good C++ ORBs (Visibroker, Chorus) can manage 100 000 servants with no overhead due to the number of servants.

Java ORBs require the maximum size of the memory allocation pool to be changed. For instance, to let a Jonathan server manage 100 000 servants, the virtual machine running it must be allowed to allocate 55 Mb (the default value is 16 Mb), even if it actually only uses half this memory: the virtual memory of Sun's JDK allocates twice the necessary memory for garbage collection purposes†.

Once this is done, the behaviour of the Java ORB is similar to that of the C++ ORBs: the overhead on invocation times when many servants co-exist in the same virtual machine is only related to swap and garbage collection.

5. Conclusion

We have described in this paper a highly flexible and modular DPE architecture organized around a very small binding-independent core. This architecture essentially introduces a distributed reference abstraction properly interfaced with that of binding. This separation enables the encapsulation of reference management and binding management in different objects: the ORB kernel for references, and various binding factories for bindings. It is thus possible to introduce in a modular fashion new bindings within an ORB based on this framework, while retaining interoperability. In the case of Java, the ORB kernel could in fact be completely *integrated* in the virtual machine: surrogates are simply an extension of the notion of object reference; they could be implemented in a reflective Java virtual machine by modifying the implementation of object references, thus enabling a seamless integration of distribution in the language.

The current Java implementation of this DPE architecture—Jonathan—has confirmed that it is stable and generic enough to accommodate arbitrary types of binding and communications mechanisms, while remaining efficient:

- We have been able to build a CORBA personality on top of the DPE kernel, featuring a stream binding factory that shows that, for instance, non-client–server bindings can be developed easily. We could in the same way develop an RMI-like personality.
- The performance tests we made show that this implementation compares well with commercial CORBA implementations like Visibroker or OrbixWeb. Comparison with RMI is harder to analyse: RMI is faster in very simple cases, but may become much slower (up to ten times) if serialization is used. Moreover, Jonathan is able to detect if the client and server are in the same capsule, and invocations are direct calls in this case. RMI does not detect this case, and calls systematically go through all the marshalling and unmarshalling process. This issue may become very important if objects are mobile.
  The modular organization of Jonathan makes it very easy to add specific protocols for real-time, or change the stub factory for a more specialized one, thus letting more performance-oriented components be plugged in. Our experience with Jonathan shows that the modular and flexible architecture we propose can be implemented at no cost in terms of performance.

Jonathan is available freely for non-commercial use. Interested readers may retrieve it from `http://www.infres.enst.fr/~delpiano/Jonathan.htm` or contact the authors to obtain a copy of the software (Java sources are included).

This work has been inspired in part by the Spring distributed system [4] and the SOR system [14]. Spring provides the notion of subcontract that allows application programmers to define new client–server communication mechanisms. SOR provides a flexible binding protocol which can be used to establish arbitrary client–server bindings. The framework proposed here generalizes these approaches to arbitrary types of interactions and in particular to multi-party multimedia communication schemes. It also generalizes more language-dependent ORB designs such as Network Objects [1] and Java-RMI [5]. Our architecture can also be seen as a weak form of the SSP chains framework [12]: in particular, the two reference chains invariants described in section 2.2.3 are the first two invariants required for the proper integration of the garbage collection algorithm of [12]. SSP chains could be developed as a specific binding

† This is true for the 1.1.x versions of the JDK.

factory maintaining the remaining necessary invariants, thus enabling acyclic garbage collection.

Future work on Jonathan will focus on the areas of resource management and admission control in order to provide full support for end-to-end QoS. Reflexivity will be used as the major architectural principle to achieve this goal. Opening the virtual machine in a reflective way would in particular allow the implementation of the ReTINA resource framework in a very flexible and modular way: the ReTINA resource control framework provides applications with access to the system-level resources they require for their execution, and this access is the first step towards quality of service management.

Acknowledgments

Many thanks to Nicolas Rivierre and Fabien Guinet, who carried out the performance tests. This work was partially supported by ACTS project ReTINA AC048.

References

[1] Birrell A, Nelson G, Owicki S and Wobber E 1995 Network objects *SRC Research Report 115* Digital Systems Research Center

[2] Dang Tran F, Dumant B, Horn F and Stefani J-B 1997 Towards an extensible and modular ORB framework *Workshop on CORBA Use and Evaluation (ECOOP'97) (Jyväskylä)* submitted

[3] Dang Tran F, Perebaskine V, Stefani J-B, Crawford B, Kramer A and Otway D 1996 Binding and streams: theReTINA approach *Proc. TINA'96 Int. Conf. (Heidelberg)* (Berlin: VDE) pp 101–113

[4] Hamilton G, Powell M and Mitchell J 1993 Subcontract: a flexible base for distributed programming *Proc. 14th Symp. Operating Systems Principles (Asheville, NC)* (New York: ACM) pp 69–79

[5] Sun Microsystems 1996 Java remote method invocation specification *Technical Report* (Mount View, CA: Sun Microsystems)

[6] ODP Reference Model: Overview 1995 *ITU-T | ISO/IEC Recommendation X.901 | International Standard 10746-1*

[7] ODP Reference Model: Foundations 1995 *ITU-T | ISO/IEC Recommendation X.902 | International Standard 10746-2*

[8] ODP Reference Model: Architecture 1995 *ITU-T | ISO/IEC Recommendation X.903 | International Standard 10746-3*

[9] ODP Reference Model: Architectural Semantics 1995 *ITU-T | ISO/IEC Recommendation X.904 | International Standard 10746-4*

[10] Object Management Group 1995 The Common Object request Broker: Architecture and Specification *CORBA V2.0*

[11] Peterson L L, Hutchinson N, O'Malley S and Abbott M 1989 RPC in the *x*-kernel: evaluating new design techniques *Proc. 12th ACM Symp. Operating Systems Principles (Litchfield Park, AZ)* (New York: ACM) pp 91–101

[12] Plainfosse D 1994 Distributed garbage collection and referencing management in the Soul object support system *PhD Thesis* University of Paris VI, Paris

[13] Website `http://www.chorus.com/Documentation/retina.html`

[14] Shapiro M 1994 A binding protocol for distributed shared objects *14th Int. Conf. Distributed Computer Systems (ICDCS) (Poznan)* (Los Alamitos, CA: IEEE Computer Society Press) pp 134–41