`export-bind`: An Architectural Pattern for Flexible Binding

Sacha Krakowiak[*], Jean-Bernard Stefani[†]

Abstract

Binding, i.e. interconnecting objects to allow them to interoperate, is a key function of any middleware system. In this paper, we present a generic design pattern for binding, based on the two primitives *export* and *bind*. This pattern has been embodied in several software frameworks, and applied to a wide variety of situations, including several communication protocols and different client-server systems. We show that the *export-bind* pattern provides the flexibility needed to develop adaptable middleware. Identifying and documenting this pattern contributes to a better understanding of a fundamental aspect of middleware construction and opens perspectives for its extension to new situations.

Keywords

architectural pattern, framework, binding, ORB, middleware

1 Introduction

Binding, i.e. interconnecting objects to allow them to interoperate, is a key function of any middleware system. Binding covers a wide range of situations, due to the variety of communication semantics that reflect different application requirements, and to the variety of contexts in which it may be applied: point to point or multicast communication, multimedia streams, client-server or peer to peer systems, persistent object systems, mobile applications. This variety calls for the use of generic and modular mechanisms that may be adapted, extended or enhanced, possibly at run time, to respond to changing requirements.

The aim of this paper is twofold:

- to propose and to document a generic architectural pattern for binding in distributed systems, based on two main primitives, `export` and `bind`;
- to demonstrate the usefulness of this pattern by showing how it is put to work in several software frameworks that cover a wide range of situations and are used in actual applications.

The pattern relies on the (possibly recursive) composition of the well-known notions of naming context and binding factory. This results in a general and flexible binding scheme, which allows binding objects (the concrete representation of bindings) to bridge distributed address spaces and heterogeneous storage systems. By documenting this pattern and its applications, we expect to contribute to a better understanding of a key issue

[*]Université Joseph Fourier, Grenoble and INRIA Rhône-Alpes. email: sacha.krakowiak@inrialpes.fr

[†]INRIA Rhône-Alpes. email: jean-bernard.stefani@inrialpes.fr

of middleware design. We hope that the pattern will be applied to new situations, such as mobile applications, which should take advantage of its generality and flexibility.

The pattern is the basis of the design of several software frameworks for communication protocols, client-server systems, and persistent object support, which have been developed within ObjectWeb (a consortium dedicated to innovative open source middleware), and are freely available on the ObjectWeb site [11]. These generic frameworks support a number of "personalities", i.e. specific middleware systems, including several communication protocols, Java RMI and CORBA.

Patterns for distributed systems have been the subject of intensive work in recent years [2, 14]. Most of the patterns developed until now fall in the category of *design patterns*, which cover a basic construction, whereas the `export-bind` pattern is an *architectural pattern*, which deals with overall design, and exploits several design patterns, including `Proxy`, `Adapter`, and `Factory` [6].

The initial base of our work is the ODP model [12, 13], which has directly inspired the design of Jonathan [4], one of the frameworks presented in this paper. Two other frameworks inspired by ODP, FlexiNet [8] and OpenORB 2 [1], implicitly use the `export-bind` pattern, but only apply it to communication services. Early work on a general model for distributed binding is [15].

The rest of the paper is organized as follows. Section 2 briefly reviews the main concepts related to naming and binding. Section 3 presents the `export-bind` pattern. Section 4 details the use of the pattern for communication protocols and for ORBs used in client-server systems. We conclude in Section 5.

2 A Brief Reminder of Naming and Binding

2.1 Names and Naming Contexts

In a computing system, a *name* is an information associated with an object[1] (the name *designates* the object) in order to fulfill two functions:

- to identify the object, i.e. to distinguish it from other objects, so the object can be unambiguously referred to.
- to provide an access path for the object, so the object can actually be used according to its specification.

A *naming system* is the framework in which a specific category of objects is named; it comprises the rules and algorithms that apply to the names of these objects. In a given naming system, a *name space* defines the set of valid names; it is usually organized into naming contexts, which correspond to organizational or structural subdivisions. A *naming context* is a set of associations between names and objects.

When dealing with names defined in different naming contexts, e.g. *NC1* and *NC2*, it is useful to define a new context *NC* in which *NC1* and *NC2* have names (e.g. *nc1* and *nc2*, respectively). This leads to the notion of a *context graph*: designating *NC1* by name *nc1* in *NC* creates an oriented arc in that graph, labeled by *nc1*, from *NC* to *NC1* (Figure 1).

Composite names may now be defined: if *a* is the name of an object *A* in *NC1*, and *b* is the name of an object *B* in *NC2*, then *nc1.a* and *nc2.b* (where the separator "."

[1]In this definition, the term "object" has a general meaning and could be replaced by "resource", meaning anything that we may wish to consider as an independent entity, be it physical or logical.

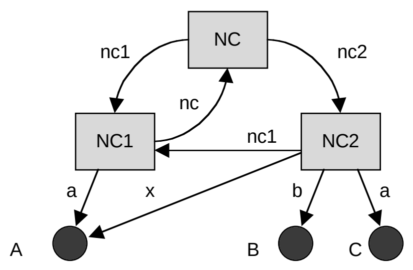

Figure 1: Names and naming contexts

denotes a name composition operator) respectively designate A and B in NC. A name thus constructed, be it simple or composite, is called a *contextual name*: it is relative to some context, i.e. it is interpreted within that context.

In order to determine the object, if any, referred to by a valid name, a process called *name resolution* must be carried out. It proceeds in steps, starting from an initial naming context. At each step, a component of the name (a label in the current context) is resolved. This operation either delivers a result, the *target*, or fails (if the current label is not bound in this context). The target may be either the address of an object, in which case the resolution is done, or a new name (together with a new context), which must itself be resolved. Name resolution may be implemented using either recursion (following the chain of contexts) or iteration (controlled by the initial context).

2.2 Binding

2.2.1 Overview

Binding is the process of interconnecting a set of objects in a computing system. The result of this process, i.e. the link, or access path, created between the bound objects, is also called a binding. Thus a binding may associate one or several sources with one or several targets, allowing them to communicate. Examples of bindings include network communication binding (4.1), client-server binding (4.2), and persistence binding (**??**).

In addition to setting up an access path, binding may involve the provision of some guarantees as to properties of this path. For example, binding may check access rights (e.g. at file opening), or may reserve resources in order to ensure a prescribed quality of service (e.g. when creating a channel for multimedia transmission).

There is an important difference between centralized and distributed systems as regards binding. In a centralized system, a memory address may directly be used to access an object located at that address. In a distributed system, a reference to a remote object (the equivalent of an address), such as [*host network address*, *port number*] is not directly usable for access. One first needs to actually create a binding to the remote object by building an access chain involving a network protocol, as illustrated by the following simple example.

Consider the case of a client to server connection using sockets. Initially, the client knows the name of the server and a port number associated with the requested service. Binding consists in creating a socket on the client host and connecting this socket to a server socket associated with the server port. The server socket, in turn, creates a new socket linked to the client socket (Figure 2). The name of the client socket is now used for accessing the server, while the server socket remains available for new connections.

It should be noted, however, that the binding may only be set up if there is actually

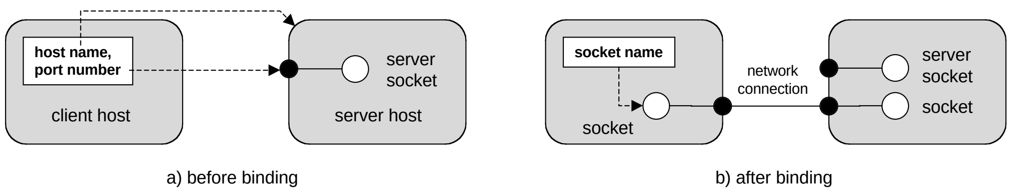

Figure 2: Client to server binding using sockets

a server socket accepting connections on the target port. As a rule, the binding must be prepared, on the target's side, by an operation that enables binding and sets up the adequate data structures.

2.2.2 Binding Objects and Binding Factories

The notions of binding objects and binding factories have been introduced in the Reference Model for Open Distributed Processing [12, 13], as a means for providing a concrete representation of the binding process.

A key feature of the ODP model is to consider the bindings themselves as objects: a *binding object* is an object that embodies a binding between two ore more objects. A binding must therefore have a distinct interface for each object to which it is connected. In addition, a binding may have a specific control interface, which is used to control the behavior of the binding, e.g. as regards the quality of service that it supports or its reactions to errors. The type of a binding is defined by its interfaces.

The notion of a binding factory has been introduced to define a systematic way of setting up bindings. A *binding factory*, or *binder*, is an entity responsible for the creation and administration of bindings of a certain type. Since the implementation of a binding object is distributed, a binding factory may itself be distributed, and usually comprises a set of local factories dedicated to the creation of the different parts that make up the binding object, together with coordination code that invokes these factories. A local binding factory is also a naming context, since it manages the names of the objects to be bound.

In the above socket example (2.2.1), the binding object is composed of the client socket, the server socket, and the connection between those sockets. The binding factory runs the code that creates the sockets by invoking the `accept` and `connect` operations on the server and client side, respectively.

3 The `export-bind` Pattern

3.1 Environment and Constraints

The pattern provides a generic mechanism for creating and managing bindings between objects in a distributed system, in order to allow communication between these objects, subject to the following requirements:

- The objects live in possibly heterogeneous systems, with different representation conventions and communication semantics.
- The pattern should be applicable at several levels (operating system, network, application), and may bridge different levels of organization. The organization structure may be of any form (hierarchical, nested, acyclic graph, etc.).

- Communication may bridge several address spaces, on several machines on a network, or may span several levels of storage hierarchies.

3.2 Pattern Description

The pattern is organized around three main entities: names, naming contexts, and binders. It includes two main primitives, `export` (borne by naming contexts), and `bind` (borne by names). A third primitive, `resolve`, is used for managing composite names. Additional primitives, described in 3.3, allow names to be marshalled and unmarshalled in order to be sent over a network or copied to persistent storage. In this section, the primitives are presented in general terms. Specific implementations adapted to various situations are described with each framework.

3.2.1 Exporting Objects and Resolving Names

To allow the construction of context graphs, such as described in 2.1, names may be associated with chains of naming contexts. Such a chain may be (conceptually) represented as, for instance, `a.b.c.d` in which `a.b.c.d` is a name in the first context of the chain, `b.c.d` a name in the next context, etc. Note, however, that the names do not necessarily explicitly exhibit this concatenated form, which is only presented here as an aid to understanding. Two operations, `export` and `resolve`, are respectively used to construct and to parse such chains.

- `id = nc.export(obj, hints)` is used to "export" an object `obj` to the target naming context, `nc`. This operation returns a name that designates object `obj` in the naming context `nc`. The initial designation of `obj` may have various forms, e.g. `obj` may be a name for the object in a different context, or a low-level name such as a Java reference. The parameter `hints` may contain any additional information used for expressing preferences or for optimization. In many cases, `export` also has the side effect of building additional data structures that are subsequently used when binding to object `obj` (see 3.2.2). The `unexport` operation precludes further use of the target name.

- `next_id = id.resolve()` is used to find the "next" name in a chain. This operation is the inverse of `export`, i.e. if `id1 = nc.export(obj, hints)`, then `id1.resolve()` returns `id`, the name of `obj` in `nc`.

Using the conceptual concatenated representation, if name `id1` is represented as `a.b.c.d`, then `id1.resolve()` returns `id` represented as `b.c.d`, etc. Conversely, calling `nc.export(id1)` where `id1` is represented by the chain `x.y.z` returns a name `id` associated with `nc` and represented as `w.x.y.z`. Calling `resolve()` on a name that is not a chain returns `null`.

3.2.2 Binding Names to Objects

Consider a name `id` in a naming context. If the naming context is also a binder, then a binding may be set up by invoking `id.bind()`. If not, then the name may be `resolved`, returning another name associated with another context, and the resolution is iterated until the naming context associated with the name is a binder (Figure 3). A name that may neither be resolved nor bound is said to be *invalid* and should not be used.

A point of terminology needs to be clarified: "resolving" a name, in the usual sense, refers to the whole sequence represented on Figure 3. We carefully isolate the issues of

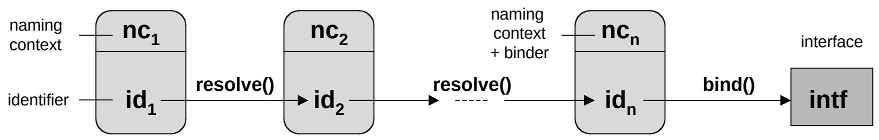

Figure 3: Resolving and binding identifiers

naming (traversing a chain of contexts) from those related to binding proper (setting up the access chain).

If the naming context of a name `id` is also a binder, then

```
handle = id.bind()
```

returns an object `handle` through which the target of the binding may be accessed (the interface of `handle` therefore conforms to that of the target). The target is an object designated by the name. The returned object `handle` may be the target itself, but it is usually a (reference to) a representative of the target (a proxy). A special form of proxy is a stub, which holds a communication object used to reach the target. The `bind` operation may have parameters that specify e.g. a requested quality of service.

In order for `bind` to work, the target must have previously been `export`ed.

3.2.3 Organization of a binder

A common organization for a binder consists of the following elements:

- a name factory for each name type managed by the binder (e.g. a binder for communication has two types of names, for clients and servers, respectively, see 4.1).
- a table of context elements (name-object associations), which is a concrete representation of the contexts managed by the binder.
- a set of references to objects that provide services needed by the binder. This typically includes factories for the objects that make up a binding (e.g. stub and skeleton factories for ORB binders, session factories for communication binders, marshaller factories, etc.).

A simple example of a binder is an adapter (an implementation of the `Adapter` pattern), which manages (and exports) a set of service providing objects (servants) on a server.

3.2.4 Examples

The socket example in 2.2.1 may be described in terms of the pattern: on the server site, `accept` is an instance of `export`, while on the client site `connect` is an instance of `bind`.

Another example is the client to server binding in Java RMI, which is described in detail in 4.2. For a better understanding of the pattern, we present a high-level view of this example.

Java RMI [16] allows a client process to invoke a method on a Java object, the target, located on a remote site. Like RPC, Java RMI relies on a stub-skeleton pair. Binding proceeds as follows (Figure 4).

On the server site (a), the target object is exported in two steps.

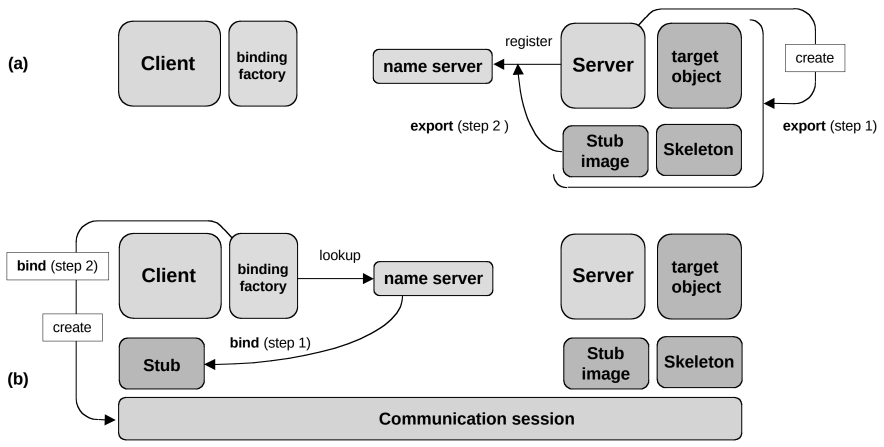

Figure 4: Using the `export-bind` pattern for client-server binding

1. An instance of the target is created, together with the skeleton and a copy of the stub (to be later used by the client).
2. The stub is registered in a name server (the RMI registry) under a symbolic name.

On the client site (b), the client calls a binding factory, which also proceeds in two steps.

1. The stub is retrieved from the registry using the symbolic name.
2. A communication session is created, to be used by the client to invoke the remote object through the stub (the information needed to create the communication session was written into the stub during the `export` phase).

Note that the target object is exported twice: first to a local context on the server site, then to the name server. This is a usual situation: actually `export` is often called recursively, in a chain of contexts.

3.3 APIs

Figure 5 describes the APIs of `Name`, `NamingContext` and `Binder`. While the APIs are presented as Java interfaces, they may be used in a different environment.

The form presented here is subject to variations, depending on the context of use. As a name is linked to a naming context, a name may delegate some of its operations to its naming context, and vice versa.

3.4 Usage Rules

The `export-bind` pattern sets up a binding between separate address spaces and allows for much flexibility as to the actual implementation of this mechanism.

- As regards binding time, the moment of the binding may be determined by the specific framework. As a first example, in Java RMI, the stub and skeleton classes are generated in advance, and the actual stubs and skeletons are created when the target object is looked up in the registry. Another example is binding to a multiparty

```
public interface Name {

 /* Returns the NamingContext that created this name. */
 NamingContext getContext ();

 /* Returns the object designated by the given name
    (may be a Name in another context).
    throws RuntimeException if the resolution fails */
 Object resolve ();

/* Returns an object that gives access to the object
    designated by the target name.
  *   hints: optional additional information.
  * throws RuntimeException if the binding fails */
 Object bind (Object hints);
}
```

```
public interface Binder extends NamingContext {

 /* Returns an object that gives access to the object
     designated by the given name.
  *   n: a name of this context.
  *   hints: optional additional information.
  * throws RuntimeException if the binding fails */
 Object bind (Name n, Object hints);
}
```

```
public interface NamingContext {

 /* Creates and returns a name in this context to designate the
     given object.
  *   o: the exported object ( may be a name of another context).
  *   hints: optional additional information.
  *   throws RuntimeException if theresolution fails
  */
 Name export (Object o, Object hints);

 /* Encodes the name as an array of bytes and returns this array.
  *   n: the name to be encoded (must belong to this context) */
 byte[] encode (Name n);

 /* Decodes the given encoded name and returns that name.
  *   b: an array of byte containing the encoded form of a name
          created by this context.
  *   throws RuntimeException if the given encoded name cannot
          be decoded. */
  Name decode (byte[] b);
```

Figure 5: The APIs for naming and binding

multimedia stream, in which the proxy objects that are used for communication are dynamically created when a provider or consumer connects to the stream.

- As regards the structure of the binding object, an arbitrary number of intermediate naming contexts and binders may be inserted between the source and target address spaces, for different reasons: adaptation layers, additional functions related to QoS (e.g. security, availability, or performance), interposition layers providing device or storage system independence. Therefore a binding may consist of a chain of elementary bindings, built by recursive application of the `export-bind` pattern.

There is clearly a trade-off between cost and flexibility, since chained binding incurs an incompressible indirection cost. This aspect is discussed in 4.3.

3.5 Known Uses

The `export-bind` pattern has been applied in the following frameworks.

1. Jonathan [4, 10], a generic ORB that consists of a minimal kernel for binding, composition, resource management, and communication protocols. Two personalities have been developed on top of this kernel: CORBA 2.3 and Java RMI.

2. FlexiNet [8], a reflective Java ORB that provides support for mobile object clusters and secure communications. Flexinet makes a direct use of names and bindings as described in this paper, mostly for communications between objects and object clusters. FlexiNet exploits the flexibility provided by the pattern, coupled with reflection, to demonstrate the implementation of a multi-protocol ORB with mobility and security support.

3. THINK [5], a framework for developing customized operating system kernels, directly built on the `export-bind` pattern. All communications between THINK components, whether local (e.g. inside an operating system kernel or between applications in user

space and the kernel - i.e. system calls and IPC) or remote (between different machines), take the form of bindings that can be reified as first-class components. Building an operating system kernel with the THINK framework is an exercise in software component assembly, where the "glue" between components takes the form of bindings with different semantics.

The next section describe the use of the pattern in two of these systems: the Jonathan communication framework (4.1) and the Jonathan Java RMI personality (4.2).

4 Using the Pattern for Communication

4.1 Network Protocols Binding

A *protocol* provides a communication service for exchanging messages through a network. A protocol relies on the services provided by a lower level protocol, down to the hardwired communication interface. This defines a layered organization such as a protocol stack or acyclic graph.

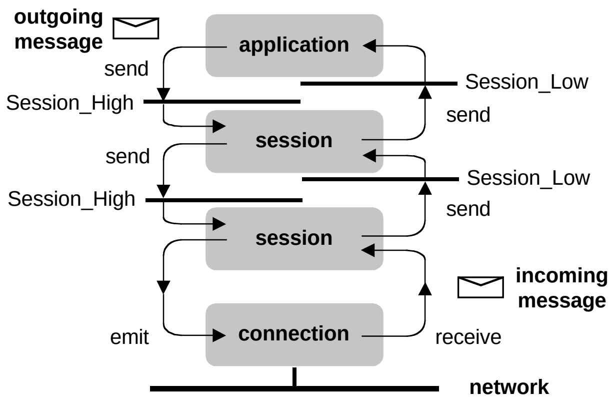

Figure 6: Sending and receiving messages

The model implemented in Jonathan is close to that of the x-kernel [9]. Its main abstraction is a *session*, an object that represents a communication channel and provides two interfaces, `Session_Low` and `Session_High` for incoming and outgoing messages, respectively. A protocol is essentially a session factory. Like protocols, sessions are organized in a hierarchy. The lower level mechanism, called a *connection*, is implemented by sockets in many middleware systems

Incoming messages (when available after a `receive` call on a connection) are passed up to the application by calling the "lower" interfaces of the sessions, in ascending order. Outgoing messages are sent down the protocol stack by calling the "upper" interfaces in descending order, down to an `emit` call on the connection (Figure 6).

A session represents a binding and is named by a session identifier (*session_id*) in the context of a protocol. There are two kinds of *session_id*s, used on servers and clients, respectively, to designate a service-providing interface, of type `Session_Low`. The client *session_id* is initially unbound (i.e. cannot be used for access).

Client to server binding uses the `export-bind` pattern, as described on Figure 7. On the server side, a protocol graph (a `NamingContext`, providing `export`) is first constructed by assembling elementary protocols. `export` takes as parameter an interface `srv_itf`, of type `Session_Low`, which provides the functionality of the server; it returns a server *session_id* (a name for the exported interface), which contains all the information needed to set up a communication with the server (e.g., for TCP/IP, the IP address of the server and a port number). This information may be transmitted over the network and decoded by a client. A call to `export` on the top element of a protocol graph recursively propagates down the graph structure.

In order to use the interface exported by a server, a client must call `bind` on a client *session_id* that designates that interface, passing its application's "lower" interface (`clt_itf`) as a parameter. This identifier may be obtained from the network (e.g. through a name service), or it may be constructed locally using the server address and port number if these are known. `bind` returns an interface `session` of type `Session_High`, which may be used by the client to call the server. Replies from the server are directed to the client application, through the `clt_itf` interface provided as a parameter of the call to `bind`.

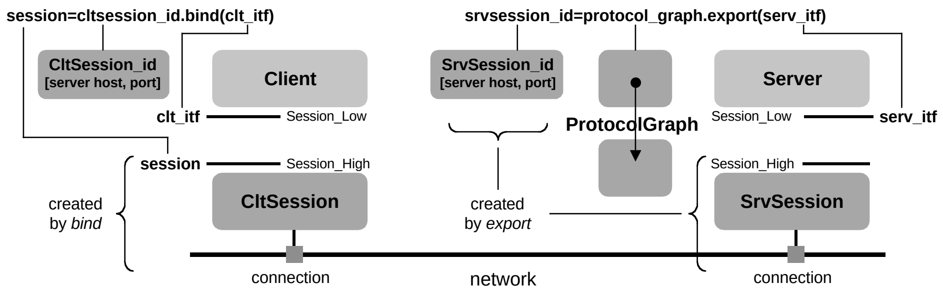

Figure 7: The `export-bind` pattern in communication

This general scheme may be applied in a variety of cases. In Jonathan, it has been used for the following protocols.

TCP-IP. On the client side, the `Session_High` interface of `CltSession` provides the `send` method, which is used to send a message using an appropriate marshaller. On the server side, the `SrvSession` relays an incoming message to the server application by upcalling a `send` method provided by that application as part of its `Session_Low` interface.

IP Multicast. As defined in IETF RFC 1112, "IP multicasting is the transmission of an IP datagram to a "host group", a set of zero or more hosts identified by a single IP destination address". In the IP Multicast protocol, there are no separate client and server roles; therefore there is no need to separate protocol graphs (exporting servers) from session identifiers (used by clients to bind to servers). A single data structure, `MulticastIpSessionIdentifier`, is used for both functions (it implements `SessionIdentifier` and `ProtocolGraph` and thus provides both `export` and `bind`). In the implementation that uses multicast sockets, both methods create a new socket with an associated session. They only differ by the type of the returned value (an identifier for `export`, a session for `bind`); in addition, `export` needs to supply a lower level interface.

Event Channel. An *event channel* is a communication channel on which two types of entities may be connected: *event sources* and *event consumers*. An event produced by a source is delivered to all the consumers connected to the channel. The channel itself is both an event source and consumer: it consumes events from the sources and delivers them to the consumers, using either a "push" or a "pull" mode of communication.

The implementation of the event channel relies on two components that closely interact: the event channel factory, and the event binder. The factory delivers implementations of the `EventChannel` interface, both in the form of actual instances and of stubs. The binder provides an interface (EBinder) allowing both sources and consumers to connect to an event channel. The implementation uses an underlying multicast protocol, typically IP Multicast.

EBinder provides a specific class of names, EIds. An EId designates an event channel built on a particular IP address and port number used by the underlying IP Multicast protocol. EIds are used as follows:

- An event source that needs to connect to an event channel designated by EId channel_id executes channel_id.bind(), which returns a Session_High session on which the source will send events (on its own initiative for push, after an upcall for pull).
- An event consumer that needs to connect to an event channel designated by EId channel_id executes EBinder.bindConsumer(consumer, channel_id), where consumer is the Session_Low interface provided by the consumer to receive events according to the implemented communication pattern (push or pull). bindConsumer is implemented as follows:

```
ProtocolGraph protocol_graph =
     channel_id.getProtocolGraph();
protocol_graph.export(consumer);
```

The export-bind pattern is again used here: bindConsumer() creates a communication session using the protocol graph of the underlying protocol; bind() returns a stub, created by a stub factory using the communication session.

4.2 Client-server Binding

Client-server systems rely on a middleware layer called an Object Request Broker (ORB), which has the following functions: identifying and locating objects; binding client to server objects; performing method calls on objects; managing objects' life cycle (creating, activating, deleting objects).

4.2.1 A Framework for ORBs

An ORB must support a variable number of objects of different types. To separate the management of these objects from that of the client-server interaction, it is usual to distinguish *servants* (object instances that implement specific functions) from the *server*, which supports the servants. The management (i.e. the control of the life cycle) of the servants is delegated to an *adapter*, which is a binder (and therefore a naming context) for the servants that it supports. Several adapters may coexist to implement different servant management policies (e.g. with respect to location, activation, replication, etc.).

The ORB proper is also a binder: an adapter exports servants to the ORB, which in turn allows remote clients to bind to them, using a communication protocol. This architecture is open, i.e. other binders may be inserted as interceptors between the ORB and the adapter to perform additional functions such as security, replication, or QoS management. Likewise, interceptors may be added on the client side. The ORB may itself delegate some of its functions to other binders, e.g. to accommodate different communication protocols.

As a consequence, an abstract view of the architecture of an ORB is a hierarchy (or possibly a graph) of binders between clients and servant objects. The export-bind pattern is used throughout: an invocation of either `export` (on the server side) or `bind` (on the client side) is recursively propagated through the chain of binders. A similar global architecture may be found in FlexiNet [8].

Figure 8 describes a generic implementation of `export` and `bind` for an ORB.

Implementation of `export` (in `Binder`)

```
Name export (Object obj, Object hints){
   create new name id (using binder's name type)
   create New element in context table with (id, obj)
   optionally export obj to other context
   optionally create new session
   return id (or last name created)
   }
```

Implementation of `bind` (in `Name`)

```
Object bind (optional parameters){
   case of
     local object:
        lookup target name in context table;
         if (found)
             {return associated object}
     remote object:
        determine session from target name
              (or create it if needed)
        create stub with this session and parameters
        return stub
   }
```

Figure 8: Implementation of `export` and `bind` for ORB bindings

Recall 3.2.3 that a binder includes a name factory, delegates stub and skeleton creation to a stub factory, and manages the association (context table) between names and objects. Exporting an object causes the binder to create a new name, to associate it with the object, and to prepare part of the future access, by calling `export` on the next binder in the chain. Typically, an adapter calls the ORB, which in turn calls a stub and skeleton factory, and delegates session creation to an underlying protocol, using the techniques described in 4.1. Binding to a name attempts to find a local object with this name; if this fails, a stub and a communication session are created or retrieved, using the structures prepared by `export` and keyed to the name.

4.2.2 A Closer View

In this section, we illustrate the use of `export-bind` by a detailed view of some steps of the client-server binding. The illustration is taken from the Jonathan implementation of Java RMI, but applies to other ORBs as well. A complete description may be found in [10].

Figure 9 describes the creation of a servant object in a "Hello World" application (this is part (a) of Figure 4). The class of the servant, `HelloImpl`, extends the predefined class `UnicastRemoteObject`, the parent class of remotely accessible objects. Starting from that class, the hierarchy of binders is as follows: `MOAContext` and `MinimalAdapter` (a specific adapter and the associated context), `JIOP` (the ORB binder of Java RMI), `IIOPBinder` (a binder associated with the IIOP protocol). The delegation structure shown on the figure makes the framework highly generic: the delegation from `UnicastRemoteObject` to `MOAContext` allows different adapters to be easily plugged in, whereas the delegation from `JIOP` to `IIOPBinder` does the same for protocols.

The net effect of the call (which recursively invokes `export` on the hierarchy of binders) is to return a stub (actually a stub template) for the servant object (this stub will be stored in a registry to be subsequently retrieved), with the side effect of creating both an instance of the servant and the associated skeleton.

Figure 10 describes an instance of binding, specifically binding to a naming registry located on a known host at a known port. This occurs when the server registers the stub constructed in the previous step under a symbolic name.

The same delegation structure applies, i.e. the ORB binder, `JIOP`, delegates to `IIOPBinder`,

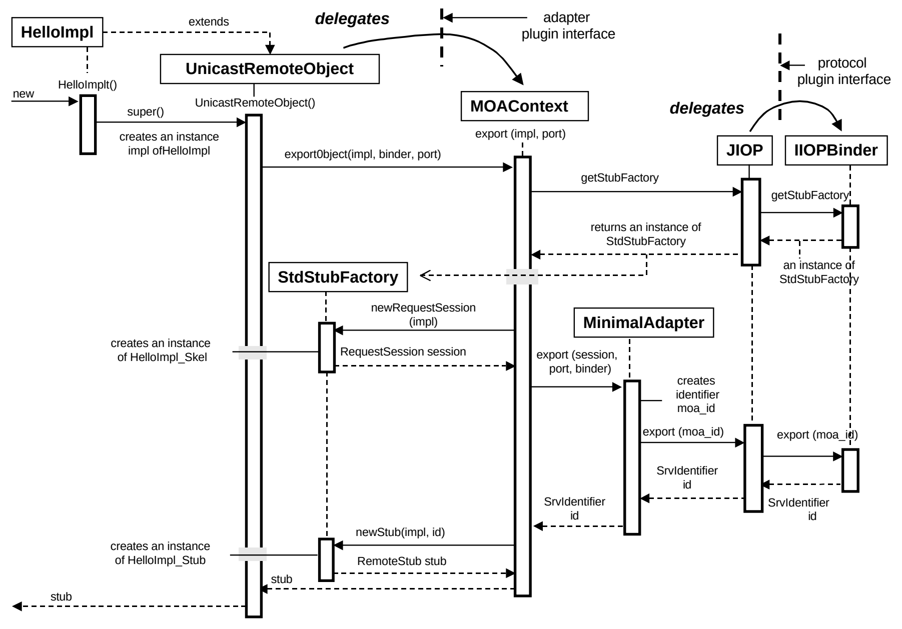

Figure 9: Exporting a remote object

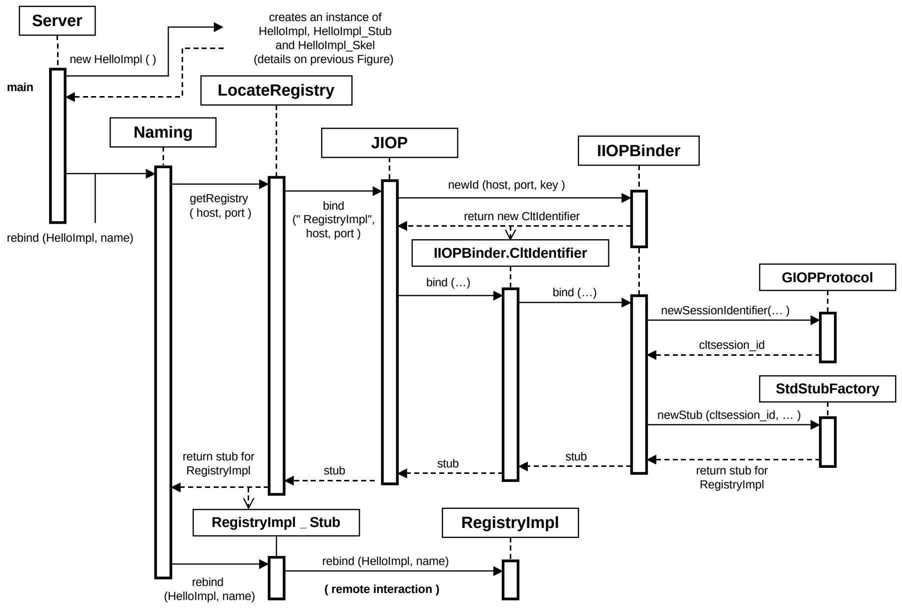

Figure 10: Binding to a remote object

which itself relies on the underlying protocol stack (i.e. GIOP over TCP/IP). Note that binding applies to a name managed by a binder; therefore a distribution-aware name (of class `CltIdentifier`) is created by `IIOPBinder`, using the provided host and port information.

4.3 Experience with Communication and ORBs

We have extensive experience with using the Jonathan framework, including an implementation of the communication layer of JOnAS, an open source EJB platform developed by ObjectWeb and used in several real life applications.

This experience has demonstrated the high degree of extensibility allowed by the open, modular structure that results from the pattern. This allowed us, in particular, to readily perform major changes such as integrating a new Portable Object Adapter (from another open source implementation) in the CORBA personality, providing an implementation of RMI over IIOP, or experimenting with enhancements for mobility.

We also have experience with different points in the performance vs flexibility trade-off spectrum. The implementations of the pattern allow for framework-specific optimizations, of which three examples follow.

- The RMI personality uses a fast path for calls to local objects, which in turn improves the performance of the EJB system built on RMI [3].
- The THINK library for building customized microkernels, not described in this paper for lack of space, uses a lightweight implementation of the pattern. Measurement results [5] show that the kernels built with THINK are competitive with the best existing optimized microkernels.
- In another series of experiments, our group has introduced optimizing techniques [7] that "freeze" a binding once established using code transformation and thus cancel the performance penalty caused by indirection.

5 Conclusion and Perspectives

We have presented the `export-bind` architectural pattern and its use for binding in various frameworks for communication protocols and client-server systems.

To summarize the main contributions of this paper:

1. We have elicited a pattern that provides a unified view and a systematic building method for a general middleware mechanism that takes a wide variety of technical incarnations. We have proved its usability by applying it to frameworks that support real life applications. We believe we have thus contributed to a better understanding of a central aspect of middleware construction.
2. We have shown that the pattern favors separation of concerns by isolating name management from binding, and by isolating the aspects related to physical organization and data representation. This allows for a high degree of flexibility in the implementation of binding.
3. While flexibility generally entails a cost, due to indirection, the openness and modularity of the frameworks derived from the pattern allow for a wide range of trade-offs between flexibility and performance, as demonstrated by the lightweight versions of the binding framework and by post-binding optimizations.

4. The chained structure of the bindings makes the frameworks readily extensible, allowing functions to be easily inserted and replaced. Thus arbitrary types of communication may be supported, not limited to client-server bindings (e.g. event channels, multimedia streams). Bindings can easily be composed and configured.

While different patterns have been proposed for specific distributed systems constructions, none (to the best of our knowledge) exhibits the generality and the wide applicability of `export-bind`.

We envisage two main follow-ups to this work. First, we intend to apply the pattern to more dynamic environments, such as mobile and ad hoc networks (MANETs), by developing frameworks for communication and service provision in such systems. Second, we are investigating the extension of the pattern to cover other functions of distributed systems that are not directly related to communication services, but use communication for their internal operation. These functions include bootstrap and configuration, resource discovery and management, and the monitoring of distributed systems and applications.

Acknowledgments

We acknowledge the contribution of the designers and developers of the systems described in this paper, with a special mention to Bruno Dumant, the main architect of Jonathan.

Code Availability

The code of the Jonathan system, together with the associated documentation and examples, is available in open source (LGPL license) from the ObjectWeb site [11].

References

[1] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Durán-Límon, T. Fitzpatrick, L. Johnston, and R. Moreira. The Design and Implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2(6), September 2001. `computer.org/dsonline`.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* John Wiley and Sons, 1995. 467 pp.

[3] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *Proceedings of OOPSLA'02*, Seattle, WA, USA, November 2002.

[4] B. Dumant, F. Horn, F. Dang Tran, and J.-B. Stefani. Jonathan: an Open Distributed Processing Environment in Java. In R. Davies, K. Raymond, and J. Seitz, editors, *Proceedings of Middleware'98*, pages 175–190, The Lake District, UK, September 1998. Springer-Verlag.

[5] J.-Ph Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A software framework for component-based operating system kernels. In *Proceedings of Usenix Annual Technical Conference*, Monterey (USA), June 10th-15th 2002.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, 1994. 416 pp.

[7] D. Hagimont and N. De Palma. Removing Indirection Objects for Non-functional Properties. In *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2002.

[8] R. Hayton and A. Herbert. FlexiNet: A Flexible Component-oriented Middleware System. In S. Krakowiak and S. Shrivastava, editors, *Advances in Distributed Systems*, pages 497–508. LNCS, vol. 1752, Springer-Verlag, 2000.

[9] N. C. Hutchinson and L. L. Peterson. The x-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[10] S. Krakowiak. The Jonathan tutorial. Jonathan documentation, The ObjectWeb Consortium, 2002. `www.objectweb.org/jonathan/doc/tutorial`.

[11] ObjectWeb. Open Source Middleware. `www.objectweb.org`.

[12] ODP. ITU-T & ISO/IEC, Recom. X.902 & Int. Standard 10746-2. ODP Reference Model: Foundations, January 1995.

[13] ODP. ITU-T & ISO/IEC, Recom. X.903 & Int. Standard 10746-3. ODP Reference Model: Architecture, January 1995.

[14] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects.* John Wiley and Sons, 2000. 666 pp.

[15] M. Shapiro. A Binding Protocol for Distributed Shared Objects. In *Proc. Int. Conf. on Distributed Computing Systems*, Poznan (Poland), June 1994.

[16] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. *Computing Systems*, 9(4):265–290, November/December 1996. USENIX Association.