APM

POSEIDON HOUSE • CASTLE PARK • CAMBRIDGE CB3 0th UNITED KINGDOM

+44 1223 515010 • Fax +44 1223 359779 • Email: apm@ansa.co.uk • URL: http://www.ansa.co.uk

ANSA Phase III

FlexiNet - Binding Framework

Øyvind Hanssen

Abstract

In this paper we report research in progress on a flexible binding architecture. We define the concepts of bindings, activation of bindings and policies for protocol choice, resource management strategices etc. for channels representing binding-activations. The activation of a binding and the choice of policies may also be governed by meta-policies. An engineering model has been sketched and an experimental framework has partly been designed and implemented. It is demonstrated how we can do lazy binding and explicit binding, using the framework. The model and design are centred around the idea of reflective proxy objects and policies represented as binders, activators and meta-objects.

APM.2057.02.00	**Approved** Technical Report	09 October 1997

Distribution:
Supersedes:
Superseded by:

TABLE OF CONTENTS

1 INTRODUCTION

The *FlexiNet Open ORB framework* [Hayton97] may be viewed as an extensible microkernel (see [Hanssen97]). It provides a framework for binding, and naming, plus a stub-generator and the necessary building blocks for creating RPC-channels. The actual binding-mechanisms may be added as extensions. This allows different binding-policies and binding-models to be used with the RPC framework.

The *FlexiNet Open ORB framework* currently supports a simple binding model, closely connected to naming. Names are simply resolved to stubs that have a standard RPC protocol stack activated and attached to them. We are interested in extensions to the framework that support:

- Alternative policies for when bindings are activated (or passivated) in the sense that resources are allocated (protocol-stacks, buffers etc), in particular lazy binding, i.e. when the activation of a binding is deferred until the first attempt is made to invoke operations.
- Explicit binding, i.e. binding is not activated until explicitly requested. Explicit binding also allows bindings to be parametrised, denoting requirements for non-functional properties (QoS) or simply a policy for how the channel should be constructed.

We are interested in support for policy-governed bindings in a wider sense, where policies are extensions that can be dynamically applied to applications. First, we should support different policies for how each binding activation is done. This typically means choice of protocols and resource management to meet certain requirements for non-functional properties of the binding (Quality of Service). Second, we should allow meta-policies that govern the activation and passivation of a binding and the choice of policy for activations.

1 .1 Purpose

The purpose of this paper is to report the progress of investigating how we can design extensions to the *FlexiNet Open ORB framework* that support policy-governed bindings.

1 .2 Report summary

In section 2 we clarify the meaning of the concepts of bindings, binding-activation and policies.

In section 3 we describe the foundations of an engineering model for a framework that supports lazy binding and binding management. First, a simple approach to lazy binding is briefly described, thereafter we introduce a more flexible approach including reflective proxy-objects that represent the bindings.

This engineering model is mapped to the design and implementation of an experimental binding framework for *FlexiNet*. This is mainly based on proxies, binders, activators and environment interfaces. In section 4 we present the essentials of the design, we discuss the most important aspects of using it and extending it with concrete policies.

1 .3 Acknowledgements

The author of this report is seconded to the ANSA Phase III programme by the University of Tromso and is supported by a NATO Science Fellowship through the Norwegian Research Council grant no. 116590/410.

Thanks to Andrew Herbert, Richard Hayton and Mathew Faupel for valuable comments.

2 CONCEPTS

In this section we clarify the meaning of the concepts of bindings, binding-activations and policies.

2 .1 Interface references[1]

Interface references are values which are offered to client programs as unique identifiers for (possibly remote) object-interfaces. They are what client programs use to reference those objects or specific interfaces to them. In object oriented languages like Java or C++, an interface reference is typically an object which has a set of methods that corresponds to the interface (stub-objects) and which encapsulates the names and protocols that are needed to be able to reach the object's implementation.

A proxy is an object representing the remote interface or object. The proxy knows how the interface is reached. The concept of proxy objects is independent of how interface-references are represented in specific language bindings or APIs. If an interface reference is represented as a stub-object, it encapsulate or reference the proxy. Alternatively, the interface reference might be just a pointer to a proxy[2]. This pointer might be used as argument to stub-methods.

2 .2 Binding

Binding is the association of a proxy, representing the (remote object), to the client program. It also involves the activation of a communication path to the remote object. However, the activation may be deferred to later, i.e. either when asked explicitly to (explicit binding), or when the first invocation attempt is made (lazy binding).

---

[1] Here, the term 'interface-reference' refers to what is seen by client-programmers to access object-interfaces (represented as some kind of object or value type). It may also refer to something that is exchangeable between separate components in a distributed environment to uniquely identify object-interfaces. This duality may be confusing.

[2] This is how it is typically done in non-object-oriented languages. See for instance ANSAware's interface references. In client programs they are just pointers to data-structures containing the nececssary information needed to reach the remote interface.

The following conceptual operations should be sufficient to describe what happens between the client application program and the system that engineer the bindings.

- Bind - Defines a binding between a client-program and the object. The Bind operation takes an identification of the object, e.g. a name and returns an interface-reference to the client that may be used to invoke operations on the object.
- Unbind - unbinds an object reference. This is usually done implicitly when the interface-reference is garbage collected.
- Invoke - invoke an operation on the remote object.

Binding is usually implicit, i.e. by invoking operations that returns interfaces to objects, binding will be done implicitly. For each object-interface that is returned from the (remote) invocation, an interface-reference will be returned to the client program.

2 .3 Activation

Activation means that resources are allocated to the object and the communication path between the client and the object, so that invocations may be carried out. This typically involves loading the object (and its class) into memory, activating (or allocating) protocol stacks and other relevant resources. Activation may also mean loading the object or relevant parts of it into the client's address space.

Activation is done according to some policy, which may be very different, due to different ways to implement objects and due to different requirements for non-functional properties (quality of service) for the bindings. When an object is bound to the client, the object will be represented by a proxy-object which know the object's "local" representation, if it's active (typically a name/address if it's remote) or alternatively, how the object should be activated.

2 .4 Policies

A binding will be associated with a policy that tells how the binding is activated to meet certain non-functional requirements. The policy typically captures implementation issues like choice of protocols, resource management strategies etc. Examples of policies include:

- Use of a shared standard RPC-channel.
- Creation of a new protocol stack for each binding, with certain allocations of resources to each channel, to meet real-time requirements.
- Use of authentication of the client, the server or both when activating the binding.

- Use of encryption, using a per-activation session key.
- The logging of each invocation at the binding.
- Adding transactional semantics to the binding, by using the necessary mechanisms and protocols.

2 .4 .1 Meta-Policies

We may also need policies for how activations of bindings are managed and for the choice of and (possibly) dynamic replacement of binding-policies as a response to changing environment, non-functional requirements and usage patterns. We therefore introduce the concept of meta-policies (policies for using policies). Examples of meta-policies include:

- Passivation of the binding after a certain time of inactivity.
- Pre-activation of bindings which are likely to be used in near future.
- Changing policies dynamically to adapt to changing resource availability, usage patterns and Quality of Service from the network. This may also include policies for degradation of the QoS delivered to the application.

Another example of a meta-policy is a permanent-object management policy (see e.g. [Hanssen95]). A persistent object may have temporary identity, i.e. it will typically have a different address for each activation, and it is passivated and reactivated several times during its lifetime, possibly at different locations in a distributed environment. A client may be given an illusion of a permanent object-identity if the changing of addresses are tracked and bindings reactivated as needed. This of course depends on a relocation service or a more permanent way to identify object-state.

2 .4 .2 Policy-trading

Policy-trading is essentially the process of mapping from a (declarative) requirement for the properties of the binding, to an implementation of a policy which is capable of meeting the requirement. A trading service is a repository of software components representing policies and it may provide services to help installing these components where requested.

A policy-trader will typically provide a service which takes the following arguments and returns the software component that match them:

- A requirement for the (non-functional) properties of the binding.
- The type of environment for the policy-implementation to run in. An environment type essentially captures what resources are available. A policy-implementation may require a certain type of environment.

3 TOWARDS AN ENGINEERING MODEL

Here we describe the foundations of an engineering model for a framework that supports lazy binding and binding management. Before we go to that discussion, we need to clarify the concepts of stubs and channels as defined by the ISO Reference Model for Open Distributed Processing [RM-ODP].

3 .1 Stubs and channels

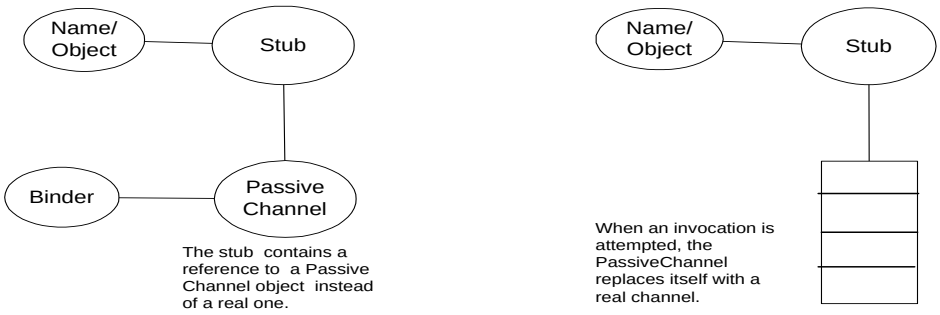
An activation of a binding is represented as a channel. A channel encapsulates all protocol layers and associated resources (e.g. buffers, threads), needed to establish communication between a client and a server-object interface. A channel may be set up differently due to different QoS requirements (different protocols and different resource usage).

A stub will need a reference to a channel to be able to carry out invocations. In the FlexiNet implementation, channels will offer a *CallDown* interface to stubs. This contains the method *callDown* which represent a generic operation invocation. Stubs simply translate typed method calls into those generic calls.

3 .2 A simple approach to lazy binding

The approach of the original FlexiNet framework is that the identifier (name) for the object is resolved to a stub by some binder object implementing a *Resolver*-interface (see [Hayton97], section 2.5). This stub contain references to the identifier plus a channel (a full protocol stack) which the binder allocates and initialises. This is eager binding. Binders represent the policy.

We may achieve lazy binding by initially letting the stub contain a reference to a fake channel instead of a real one. This object (we call it *PassiveChannel*) implements the *CallDown* interface (like protocol stacks do) and when a call from the stub arrives, the *PassiveChannel* will use another binder (it typically know of a default binder) to replace itself with a real channel and forward the call to it. This is illustrated by the figure below.



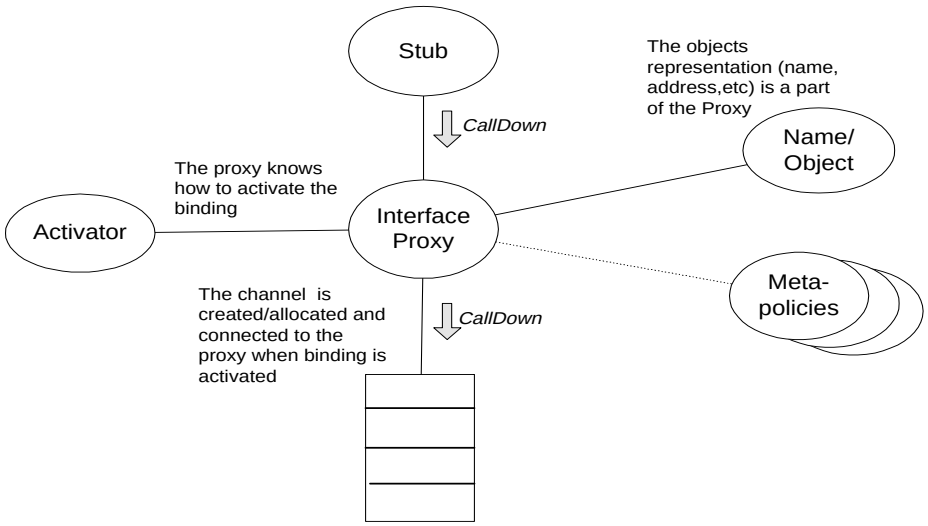
3 .3 A more flexible approach

The figure below illustrates an alternative approach. It captures the distinction between binding and activation of bindings as well as the distinction between stubs and bindings. It offers a flexible framework for adding extensions, representing a wide range of policies and meta-policies.

3 .3 .1 Interface proxies

An interface proxy represents the binding to the (remote) object's interface, regardless of whether this binding is active, and in some cases, regardless of whether the (remote) object itself is active. The proxy encapsulates the identity of the remote interface (it contains the name) and it knows how to activate bindings.

The proxy implements the *CallDown* interface (c.f. section 3.1) and when receiving calls from stubs, it adds the address/name of the object, activates binding if necessary and forwards the call to the channel. From the stub's point of view the proxy act as a transparent channel. The proxy may also have other interfaces to offer policy management, i.e. it is used for control of meta-level decisions that affect non-functional properties of the binding it represent, typically explicit binding and rebinding.



This approach also allows more than one stub to share one proxy. This may be desirable of several reasons. First, different parts of the same process may have a stub each, representing the same interface to the same object. This may be natural if the sharing of data between those parts is limited. It may also be an ad hoc solution to concurrent use of the same interface, because *FlexiNet* stubs are not re-entrant at the moment. The concurrency control between invocations is then the responsibility of the proxy.

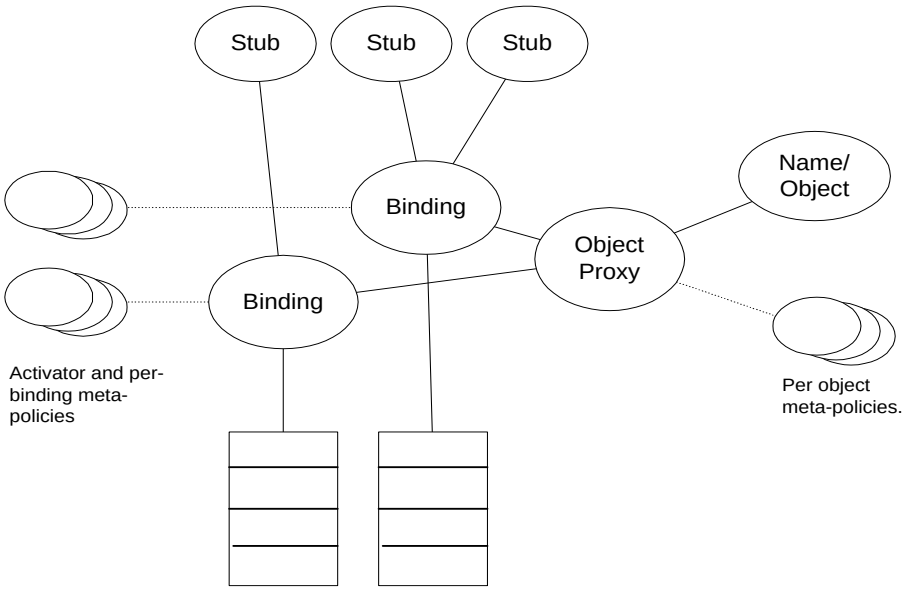
3 .3 .2 Meta-objects

This approach may be regarded as reflective in the sense that most of the semantics of the proxy is defined in replaceable meta-objects which are connected to the proxy. An activator object represents the binding-policy and is capable of constructing channels when needed. It may also have meta-objects representing meta-policies.

3 .3 .3 Object proxies

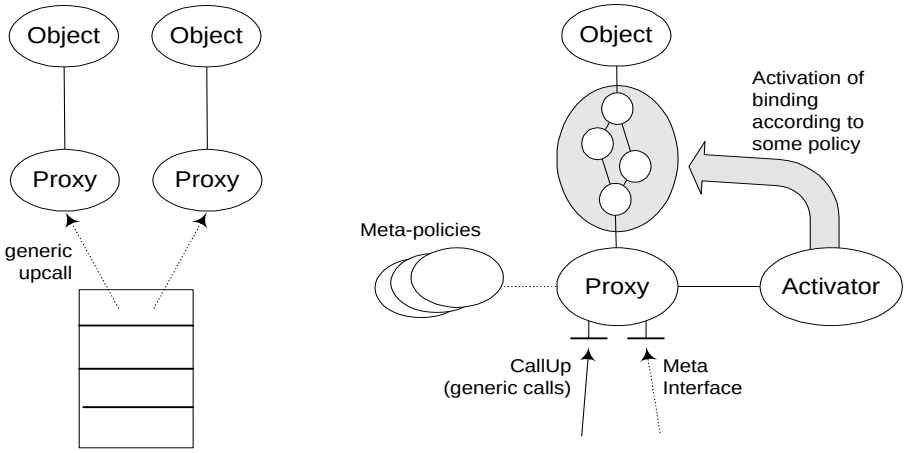
There may be more than one binding using the same proxy. The model illustrated in the figure below reflects the fact that one object may export more than one interface and that there may be more than one binding to one object, each which may be governed by a different set of policies. Each binding will be associated with an activator and (possibly) per-binding meta-policies.

The object-proxy represents what is shared between the different bindings to the same object: The object-identity and per-object meta-policies. A per-object meta-policy may (c.f. section 2.4.1), manage object migration and change of identity.



3 .4 Server interface proxies

The proxy-object idea apply to the server side as well. When an interface is exported, a channel will be used for listening for incoming calls. Instead of registering the object itself in the global namer, one might choose to register a proxy for it. This proxy must implement the *CallUp* interface[3] to receive generic (up)calls from clients. When receiving the first upcall, the proxy (typically by using an activator) activates the binding to the object by inserting per-interface protocol layers between the proxy and the object. It then forwards the call upwards through these layers. This is a kind of lazy binding. If we also need to activate the lower part of the protocol stack lazily and non-shared, a new stack might be created, but this would require part of the interface's address changed. The figure below illustrates the role of server side proxies.



The proxy might also implement a meta object protocol[4] that allow the manipulation of the activations and the policies used. This may be exploited by the client-side proxy to negotiate the choice of policies with the server. This might be done "in-band" when doing simple lazy binding (meta-protocol information is piggybacked on the first invocation). Alternatively we might do it "out-of-band" in form of explicit operation calls.

Therefore, the proxy should implement (typically by using an activator) a meta-object-interface with operations for activation, passivation and policy-management. Activation may also involve activation of the object itself if this is part of the policy. The proxy objects implementation of the *CallUp* interface might be able to recognise calls for meta-operations and direct them to the correct implementations.

[3] Upcalls at the server side corresponds to client side downcalls (see section 3.1).

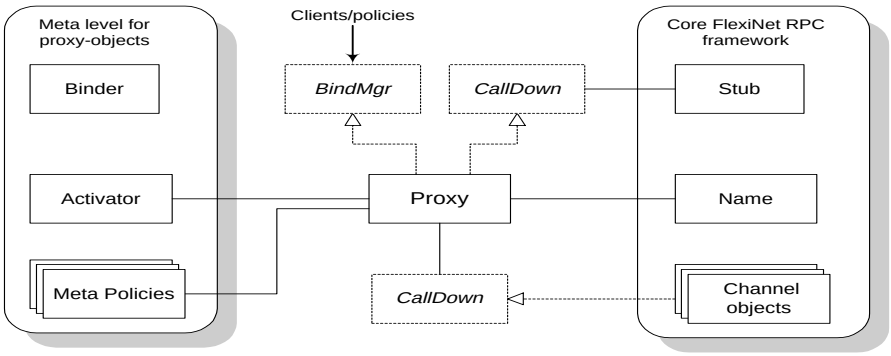
[4] The Open Implementation Approach [Kiczales91, Kiczales96] suggest opening up objects by adding meta-interfaces separate from the functional interfaces, to control non-functional aspects, typically by reflective techniques.

4 AN EXPERIMENTAL FRAMEWORK

Important aspects of the engineering model of section 3 is mapped to the design and implementation of an experimental binding framework, based on *FlexiNet*. It is focused at the client side, but as we saw in section 3.4, many of these ideas, designs and even code, may be reused in a server-side framework. Here, we present the essentials of the design, we discuss the most important aspects of using it and extending it with concrete policies.

4 .1 Design overview

The figure below[5] shows an overview of the core binding framework at the client-side. The design is centred around the *Proxy* class which implements the *CallDown* interface and the *BindMgr* interface. Each stub-object will contain a reference to a proxy through its *CallDown* interface. The *BindMgr* interface may be used by application programs or meta-policies to control the activation of the binding and to do policy management.



4 .1 .1 Binders and activators

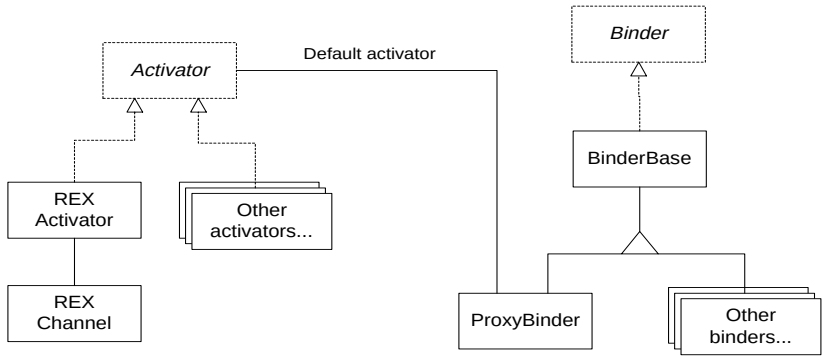
The design includes binder classes which implements the *Binder* interface[6]. *Binder* classes typically subclass the abstract class *BinderBase*. This class

[5] We use the OMT notation where dashed boxes and arrows indicate interfaces and the relationship with classes that implements them.

[6] The *Binder* interface combines the *Resolver* interface which does client-side binding and the *Generator* interface which does server side binding. A single binder-object may do both, but it is also possible to have separate binders for client interfaces and server interfaces (c.f. [Hayton97] section 2.5).

implements a generic *resolve* method which creates a stub given a name. This uses a special *resolve* method implemented by subclasses, which create and return a proxy representing the binding. In this sense, binders can be regarded as factories for proxies. Different subclasses may be written to create differently configured proxies. We provide a simple *ProxyBinder* class which in fact support a wide range of policies. Here, the default activator objects to be used, is referenced by each instance of *ProxyBinder*. However, the activator referenced by each proxy may be replaced later by using the *BindMgr* interface.

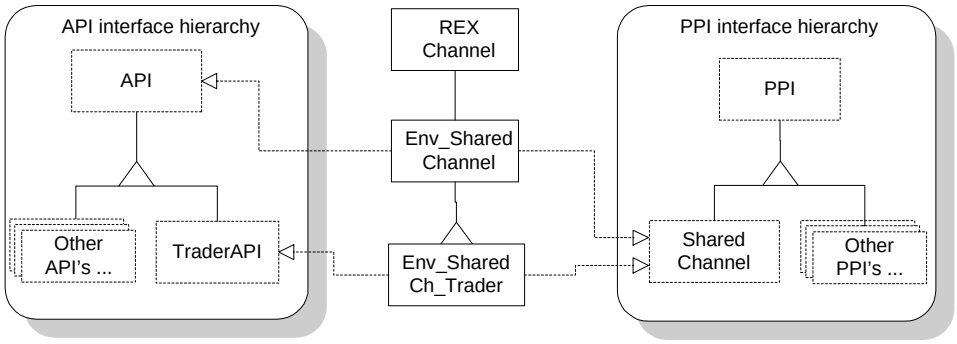
Similarly, we have activator classes, each which implements the *Activator* interface. Different activator classes may be written, each which represent a different policy for how channels are set up. The meta-level design with the *Binder* and *Activator* frameworks is shown in the figure below



4 .1 .2 Environments

Different environments will provide slightly different API's and different sets of services/resources for use by activators (and possibly binders). To help managing different environments, we could encapsulate them into environment objects, each which implements two interfaces: (1) The API for use by application programmers and (2) the Policy Programmer Interface (PPI) for use by policy programmers. Clearly, every activator implementation will depend on a PPI interface compatible with a particular type.

Standardising on a set of API and PPI interfaces forming two separate type-hierarchies, would then support the writing of portable components, both applications and policy-components, that could be deployed in many different environments conforming to the interfaces the components require. The figure below illustrates this idea. The roots of the interface hierarchies (*API* and *PPI*) provides a minimal functionality which could be extended by subtyping. Two example environments are shown, one which provides a shared channel for RPC interaction (*REXChannel*) and one which adds the use of a trading service (c.f. [Hayton97] section 3.3) to the *API*.



4 .2 Usage and extension

Here, we illustrate how lazy binding and explicit binding can be realised using this framework. The framework is meant be extended with concrete policies and meta-policies, so we illustrate how these may be represented (as binders, activators and meta-objects).

4 .2 .1 Lazy binding

The current *Proxy* class supports lazy binding. The *callDown* method simply checks if the binding is active and if not, it uses the activator to activate it. The code fragment below shows the essential workings of the *callDown* method (concurrency control code removed for clarity):

```
class Proxy implements CallDown, BindMgr
{
   ...
   void callDown(CallData data)
   {
      data.address = _name;                              // Add name to call-frame
      if (_channel == null)                              // If binding is not active (null channel)
          _channel = _activator.getChannel(_name);       // ... activate it, by using activator
      _channel.callDown(data);                           // forward the call to the channel
   }
   ...
}
```

4 .2 .2 Explicit binding management

Explicit binding and binding management is supported through the *BindMgr* interface, which offers the following methods:

- passivate - passivates the binding by deallocating the channel.
- activate - if the binding is passive, use the activator to set up a channel.
- rebind - replace the activator. If the binding is active, passivate it, replace activator and reactivate.

The code fragment below shows an example of how a binding may be activated explicitly using an alternative activator. A tricky part of this is to

get hold of the *BindMgr* interface. This is done by using the *getBody* method provided by stubs (through the generic *FlexiStub* interface) to get the proxy. The *getBody* method returns a *CallDown* interface, so we try to cast it to a *BindMgr* interface type. We have no guarantee that the returned object also implements the *BindMgr*, so there is always a chance of a *ClassCast Exception* being thrown, if not. The next step is to find an activator that support our requirements for non-functional properties and which can make use of our current environment. We might create a trading service for looking up such policy objects. In this example we use a policy-trader to look up an activator that provides secure bindings, using the PPI of the current environment.

```
{
   ...

   Account a = bank.access(accno, pin);
   try {
      BindMgr   mgr = (BindMgr) ((FlexiStub) a).getBody()
      Activator act = policyTrader.import("Secure", env.getPPI() );

      mgr.rebind(act);
      mgr.activate();

      a.withdraw(amount);
      ...
   }
   catch (ClassCastException e) {
      System.out.println("Sorry, BindMgr not supported");
   }
}
```

4 .2 .3 Writing binders and activators

New binder-classes may be created by subclassing the *BinderBase* class like shown in the example below. Essentially, we need to write a method *resolveName* which creates and returns a proxy. The proxy needs to be given an activator when created. We may use a default activator or a more dynamic way of finding a suitable activator.

```
class MyBinder extends BinderBase
{
    ...

    public CallDown resolveName (Name name)
    {
        Proxy p = new Proxy(name, SELECT-DEFAULT-ACTIVATOR);
        return p;
    }
    ...
}
```

New activator-classes essentially have to implement the method *getChannel*, which should return the interface to a channel. The channel may be set up differently by different activator classes to reflect different policies for allocating resources, for the use of protocols (typically multiple layers), or for sharing of channels or sub-layers.

A simple activator, *REXActivator* is provided as an example. This simply makes use of a standard RPC channel, which is shared between all proxies using this activator. The shared channel is implemented and installed by the *REXChannel* class. This may also be used by other activators which may add

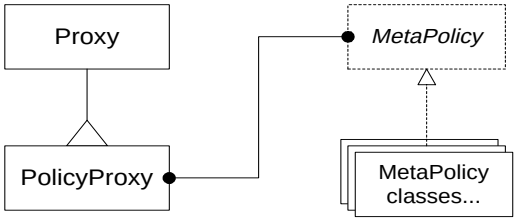
extra layers on top of the shared REX channel. An example of this is the *LogActivator*. It adds a logger object as a layer between the proxy and the shared channel. The logger prints information on each invocation to the console and forwards it to the layer below. The essentials of this class is shown below. It depends on the SharedChannel PPI interface. A PPI is argument to the constructor method which checks the actual type of it and throws an exception if it is incompatible. Arguments to the instantiation of the logger object are a text to be inserted in each printout and a *CallDown* interface, representing the layer below. The method *getClientCall*, returns the interface to the channel's topmost layer.

```
class LogActivator implements Activator
{
    private SharedChannelPPI _ppi;

    public LogActivator(PPI ppi) throws IncompatiblePPI
    {
       if (ppi oftype SharedChannelPPI)
          _ppi = (SharedChannelPPI) ppi;
       else
          throw IncompatiblePPI(ppi);
    }

    public CallDown resolveName (Name name)
    {
        Logger l = new Logger("["+name+"]", _ppi.getChannel());
        return l;
    }

    ...
}
```

4 .2 .4 Meta-policies

Meta-policies are typically represented by binder classes. If for instance the meta-policy is to do eager binding, this is reflected in the binder class, which then calls the *activate* method of the proxy as soon as it is created. More advanced meta-policies may need to be reflected on the behaviour of the proxy itself. A simple way to do this is to subclass the proxy and redefine or add behaviour.

A more flexible way to do this is to separate meta-policy behaviour into one or more meta-objects. We may use reflection techniques like those investigated by the Reflective Java project [ReflJava]. The framework provide a subclass of *Proxy* which redefine the *activate* and *passivate* methods such that they before and after activation/passivation notify the meta object(s). Every meta-object should then implement the *pre_activate*, *post_activate*, *pre_passivate* and *post_passivate* methods. Some policies may use one meta-object per proxy and others may share meta-objects between proxies, so there is a many-to-many relationship between proxies and meta-objects. The figure below illustrates this idea:



An example of a meta-policy is one that simply logs the events to the console. This meta-object-class is then very simple. Each method just prints an identification of the remote interface and an indication of what is going on.

```
public class MetaLogger implements MetaPolicy
{
    private String _name;

    public MetaLogger(String n) {
          _name = n;
    }

    public void pre_Activate()  {
          System.out.println("["+_name+"] -- Pre Activate");
    }

     public void post_Activate() {
          System.out.println("["+_name+"] -- Post Activate");
     }

     public void pre_Passivate() {
          System.out.println("["+_name+"] -- Pre Passivate");
     }

     public void post_Passivate() {
          System.out.println("["+_name+"] -- Post Passivate");
     }
}
```

We also need to create a special binder class that uses the correct proxy class and installs the meta-objects in each proxy.

```
class LogBinder extends BinderBase
{
    ...

     public CallDown resolveName (Name name)
    {
          PolicyProxy p = new PolicyProxy(name, _activator);
          p.setMeta(new MetaLogger(name.toString()));
          return p;
     }

      ...
}
```

4 .3 Summary

The core of this design is the proxy-object which represent the binding to the remote interface. It is a transparency layer in the sense that it through the *CallDown* interface hides the underlying activation management of the binding. It also exports the *BindMgr* interface which allows explicit activation, passivation and policy management.

What we have designed is a framework which is meant to be extended by concrete policies and (possibly) meta-policies. Policies are represented by activators which sets up and configures channels in different ways. A specific channel configuration is a working manifestation of a policy. Meta-policies are mainly represented in different binders. In addition, many meta-policies may need specialisations of the proxy. One may create a reflective proxy-class where meta-objects connected to proxies implement meta-policy behaviour. The table below summarises how policies are represented.

	Instantiation	Run-time
Policy	Activator	Channel-config
Meta-Policy	Binder	Meta-object

To support the portability of components representing policies to different environments and to support the choice of the right policy for the right environment, we separate the environment from its application programmer interfaces and policy programmer interface. Application- or policy-components can then require compatibility with certain interfaces.

5 CONCLUDING REMARKS

In this paper we report research in progress on a flexible binding architecture. A binding is an association between the client program and the remote interface, represented by some proxy-object. A binding needs to be activated before it can carry out invocations. Activation involves allocating necessary resources to the binding and (possibly) the object itself. Activation is done according to some policy which denotes protocol choice, resource management strategies etc. The activations of a binding and the choice of policies may also be governed by a meta-policy.

A engineering model has been sketched and an experimental framework has partly been designed and implemented. It is demonstrated how we can do lazy binding and explicit binding, using it. We introduce the proxy as a transparency object, representing the bindings. At the client side the proxy knows how to reach the remote interface and activate the binding. Policies are represented as activators which can set up channels and allocate the necessary resources. Meta-policies are reflected on binders that are responsible for instantiating and configuring proxies. Proxies are reflective in the sense that its behaviour representing policies and meta-policies are separated out as replaceable metaobjects: (1) The activator defining the policy behaviour and (2) zero or more metaobjects defining meta-policy behaviour.

The framework described here is still under development. What needs to be done further includes the realisation of a server side framework, performance evaluation, optimisations and testing of different environment configurations and policies to evaluate the flexibility. The framework is believed to be an useful testbed for further binding related research. Issues for further research and research in progress include:

- Policy selection based on declarative requirement-statements and descriptions of the environment. This could be simple selection from a set of pre-implemented policies (policy-trading) or automatic construction of policy implementations, possibly by using building blocks from a pre-implemented set.
- Languages for declarative specifications of policies and automatic generation/construction of implementation from these, possibly in run-time.
- Protocols and techniques for end-to-end policy-binding. Negotiation between client and server.

REFERENCES

[Hanssen95]
Ø. Hanssen, F. Eliassen, “On the design of a generic object adaptor”, proc. 8th ERCIM Database Research Group Workshop on Database Issues and Infrastructure in Cooperative Information Systems, Trondheim, August 1995.

[Hanssen97]
Ø. Hanssen, “FlexiNet - Extensible Kernel Investigation”, ANSA Phase III report APM.2002.01.00, July 1997.

[Hayton97]
R. Hayton “FlexiNet RPC framework”, ANSA Phase III draft report APM.2047.00.01, August 1997.

[Kiczales91]
G. Kiczales, J.D.Riveres, D.G.Bobrov, “The Art of the Metaobject Protocol”, MIT-Press, 1991

[Kiczales96]
G. Kiczales, “Beyond the Black Box: Open Implementation”, IEEE Software, 1996, 13(1), p. 8-11. (se also the Open Implementation Home Page Xerox Palo Alto Research Center, http://www.parc.xerox.com/oi).

[ReflJava]
Reflective Java, meta-level programming from APM: http://www.ansa.co.uk/Research/ReflectiveJava.htm

[RM-ODP]
ISO/IEC JTC1/SC21/WG7, “Basic reference model of Open Distributed Processing – Part 1: Overview”, ISO/IEC DIS 10746-1.