A Versatile Kernel for Distributed AOP

Éric Tanter Rodolfo Toledo

University of Chile, Computer Science Dept.
Avenida Blanco Encalada 2120, Santiago, Chile
{etanter,rtoledo}@dcc.uchile.cl

Abstract. Aspect-Oriented Programming (AOP) promotes better separation of concerns in software systems by introducing aspects for the modular implementation of crosscutting concerns. As a result, modularity and adaptability of software systems are greatly enhanced. To date, very few AOP proposals take distribution into account. This paper considers the explicit introduction of distribution in AOP, by proposing support for distributed aspects: all dimensions of aspects are studied in the light of distribution. The result of this work is a versatile kernel for distributed AOP in Java: a flexible infrastructure that allows aspects to be defined and applied in a distributed manner, on top of which various distributed aspect languages and frameworks can be defined.

1 Introduction

Aspect-Oriented Programming (AOP) provides means for proper modularization of crosscutting concerns [8], *i.e.* concerns that cannot be cleanly modularized using traditional programming paradigms. Typical examples of such concerns are *non-functional* concerns such as monitoring, security, concurrency, etc., but also *functional* concerns such as observation relationships and, in general, coordination between different modules. Without AOP, the implementation of such concerns is scattered across several modules. The importance of AOP for practical software engineering is reflected in the growing interest manifested by industrial actors, in particular in application servers [10]. AOP also helps *adaptation* of software systems: for a given concern to be adaptable, it first has to be modularized.

The relation between AOP and distributed computing is interesting. Even though AOP is used in application servers, aspects are defined and applied locally to enhance the implementation of the application server; most AOP proposals to date do not support the remote definition and/or application of aspects. In other words, *AOP in distributed systems is NOT distributed AOP*. To our knowledge, only JAC [16], DJcutter [15], and AWED [4] address distributed AOP as such, by enhancing the language constructs of AOP to cover distribution. However, each proposal has its set of limitations, as will be discussed later. Among motivating examples of distributed AOP are distributed unit testing [15], sophisticated distributed cache policies and checking of architectural constraints in distributed systems [3].

In this paper, we adopt a general approach to distributed AOP, by extending our previous work on *versatile kernels for AOP*: expressive and flexible infrastructures for AOP on top of which different AOP languages and frameworks can be developed [23, 24]. Our methodology consists in revising all the concepts of our AOP kernel for Java, Reflex [1], in the light of distribution. The result is a versatile AOP kernel for distributed AOP in Java, named ReflexD, which can be used to define and apply aspects in a distributed manner.

In Section 2, we discuss the notion of distributed AOP, analyzing the different elements of AOP and what it means to consider them in the light of distribution. Section 3 briefly introduces the notion of AOP kernels in general, since we follow this line of work here. Section 4 exposes the different elements of ReflexD, our versatile kernel for distributed AOP in Java. In Section 5 we explain how a distributed notion of control flow can be built with ReflexD, and apply it in Section 6. Section 8 discusses related work and Section 9 concludes.

2 Distributed AOP

We now briefly discuss the main elements of an aspect in AOP, in order to later analyze what *distributed AOP* means.

2.1 Elements of AOP

The anatomy of an aspect can be roughly described as follows:

- the **cut** of an aspect describes the execution points of a program to which the aspect applies, *e.g.* calls to state-changing methods on shape objects;
- the **action** of an aspect describes the effect of the aspect at its cut, *e.g.* tracing the underlying calls, or requesting a lock before proceeding;
- the **binding** between a cut and an action specifies issues such as when the action is executed (before, after or around the intercepted execution point), the context information to be exposed to the action, etc.

An aspect language typically extends a traditional programming language with language constructs for the above elements. For instance, the most-used Java AOP extension to date is AspectJ [11], which extends Java with constructs to define aspects, with pointcuts (the cut) and advices (the action). In AspectJ, the binding between a cut and an action is split between both: it is not a separate entity. Below is a simple tracing aspect in AspectJ:

```
aspect Trace {
  pointcut fooCalls(Object x): call(* A.foo(..)) && this(x);
  before(Object x): fooCalls(x) { // log call made by x }
}
```

A pointcut `fooCalls` is defined, matching all calls to method `foo` on objects of type `A`, and exposing a single parameter `x`, bound to the instance performing the call (using `this(x)`). Then an advice is associated to the pointcut: when the pointcut matches, the body of the advice is executed before the original call is performed. Variable `x` is available in the scope of the advice body.

[1] http://reflex.dcc.uchile.cl

2.2 Distributed AOP

Aspect-oriented programming enhances software modularity and adaptability by promoting better modularization of otherwise crosscutting concerns. The entailed benefits of using AOP are of interest for any kind of complex software systems, and in particular, distributed systems: for instance to express aspects covering crosscutting interactions between remote entities. However, as argued in [15], simply combining an existing AOP language such as AspectJ with an existing framework for distributed systems like Java RMI (Remote Method Invocation) [19] is not a solution.

As a matter of fact, a framework like RMI extends OOP to the world of distributed programming, but does not help for AOP. The fact that the very concepts of aspect languages are not extended to distribution forces programmers to define a distributed aspect as a collection of distributed entities realizing the whole aspect based on remote calls. It is not possible to define a distributed aspect as a simple, non-distributed entity [15]. Hence developing distributed aspects is made much more complex than it should be, and deployment issues are exacerbated. Leveraging AOP to distribution requires the very concepts of AOP to be revisited in the light of distribution:

- **distributed cut**: describing execution points of interest must possibly discriminate among execution *hosts* (*a.k.a. remote pointcuts* [15, 16]); a method call may be of interest only if called in a particular host.
- **distributed action**: the effect of an aspect should possibly be executed on a *remote* host, not necessarily where the cut is realized; *e.g.* the activity of a process in a production machine monitored on a separate machine.
- **distributed binding**: the specification of the binding between the cut and the action of an aspect may be done in any host, which may not be the host where the cut is realized or the action is executed.

This defines our approach to distributed AOP. Given the variability in each of the above elements, we target a flexible architecture covering these notions, focusing on the core semantics first; syntax is not considered in this work.

3 The Kernel Approach to AOP

This section briefly introduces the necessary background concepts on AOP kernels and our Java implementation, Reflex. More elements on Reflex will be introduced as necessary in the course of the paper.

3.1 Versatile Kernels for AOP

In previous work [23, 24], we have motivated the interest of being able to define and use different aspect languages, including domain-specific ones, to modularize the different concerns of a software system. We have proposed the architecture of a *versatile kernel* for multi-language AOP, and our current Java implementation, Reflex. An AOP kernel supports the core semantics of various AO languages

through proper structural and behavioral models. Designers of aspect languages can experiment comfortably with an AOP kernel as a back-end, as it provides a higher abstraction level than low-level transformation toolkits. Furthermore, a crucial role of an AOP kernel is that of a *mediator* between different coexisting AO approaches: *detecting* interactions between aspects, possibly written in different languages, and providing expressive means for their *resolution* [21].

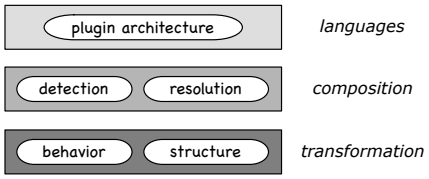

Fig. 1. Architecture of a versatile kernel for multi-language AOP.

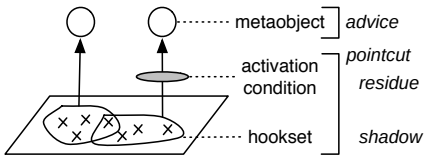

Fig. 2. The link model and correspondence to AOP concepts.

The architecture of an AOP kernel (Fig. 1) consists of: a *transformation* layer for basic weaving, supporting both structural and behavioral modifications of programs; a *composition* layer, for detection and resolution of aspect interactions; a *language* layer, for modular definition of aspect languages (as plugins).

3.2 Reflex in a Nutshell

Reflex is a portable library for structural and behavioral reflection in Java, operating as a `java.lang.instrument` agent on bytecode. This paper only deals with behavioral facilities, which follow the model of partial behavioral reflection of [25]: explicit *links* binding a set of program points (a *hookset*) to a *metaobject*. A hookset is defined as a condition over reifications of program elements: an `RPool` object gives access to `RClass` objects, which in turn give access to their members as `RMember` objects (either `RField`, `RMethod`, or `RConstructor`), which in turn give access to their bodies as `RExpr` objects (with a specific type for each kind of expression). These objects are causally-connected representations of code, offering a source-level abstraction over bytecode.

A link is characterized by a number of attributes, among which the *control* at which metaobjects act (before, after, around), their *scope* (per object, class, or global), and a dynamically-evaluated *activation condition*. Fig. 2 depicts two links, one of which is not subject to activation, along with the correspondence to the AOP concepts of the pointcut/advice model of AspectJ. In Reflex one can specify, on a *per-link* basis, the exact communication protocol (which method to call with which arguments) with the metaobject implementing the aspect action.

Links are a mid-level abstraction, in between high-level aspects and low-level code transformation. How aspect languages are defined and implemented over the kernel is out of the scope of this paper (see [24]); aspect composition in Reflex

is treated in [21]; a detailed case study of supporting the dynamic crosscutting of AspectJ in Reflex can be found in [17]. A simple AspectJ aspect, comprising of a single advice associated to a simple pointcut, is straightforwardly implemented in Reflex with a link (as in Fig. 2). Below is the implementation of the link equivalent to the `Trace` AspectJ aspect shown in the previous section[2]:

```
Hookset fooCalls = new Hookset(MsgSend.class, new NameCS("A"),
                               new NameOS("foo"));
Link trace = Links.get(fooCalls, new Tracer());
trace.setControl(Control.BEFORE);
trace.setCall("Tracer", "log", Parameter.THIS);
```

We first create a hookset selecting occurrences of the message sending operation, with a name-based class selector matching class `A` and a name-based operation selector matching occurrences of `foo` messages. Then a link is created, binding this cut to the action defined in a `Tracer` metaobject. The control of the link is set to before, and we specify that the `log` method of the tracer must be called, with the predefined parameter corresponding to the current instance (`THIS`).

Nevertheless, most practical AOP languages, like AspectJ, make it possible to define aspects as modular units comprising *more than one* cut-action pair. In Reflex this corresponds to different links, with one action bound to each cut. Furthermore, AspectJ supports higher-order pointcut designators, like `cflow`. In Reflex, the implementation of such an aspect requires an extra link to expose the control flow information. This is further discussed in Section 5.

4 A Kernel for Distributed AOP

We now go through the different features of our versatile kernel for distributed AOP in Java, ReflexD. ReflexD is an extension of Reflex, currently implemented using Reflex itself (for transparently handling remote communication and consistency between objects) and RMI as a base for remote invocation.

4.1 Distributed Cut

Reflective model extended. Cut definition in Reflex is based on a reflective model representing code as Java objects (`RClass`, `RMethod`, `RExpr`, etc.). To take distribution into account, the model is extended with the reification of a host:

```
public interface RHost {
  public String getName();
  public String getAddress();
  public Properties getProperties();
}
```

A `RHost` object reifies a running Reflex-enabled VM, identified by its name given at launch time; a `RHost` object can be obtained with `RHosts.get(address, name)` where `address` is the physical address (*server*`:`*port*) of the Reflex host named `name`. Apart from the name and address, the system properties of a host are also exposed. All other entities of the reflective model are augmented with the information of the host in which they are defined.

[2] Concrete syntax for Reflex is under development [22], but we do not use it here.

Hookset extended. An aspect cut is expressed with a hookset, *i.e.* a condition over program elements from the reflective model. In addition to class and operation selectors, *host selectors* are used to express conditions over hosts:

```
public interface HostSelector {
  public boolean accept(RHost aHost);
}
```

The host selector discriminates the hosts of interest. A simple `NameHS` can do name-based selection, while more advanced selectors can use the host system properties. For instance, the following selector matches the group of *development hosts*, *i.e.* hosts that have a custom property `"type"` with value `"devel"`:

```
public class DevelopmentHosts implements HostSelector {
  public boolean accept(RHost aHost){
    return "devel".equals(aHost.getProperties().get("type"));
} }
```

It is therefore possible to define a link whose cut matches events in different hosts, providing the necessary support to handle distributed crosscutting.

Activation extended. Dynamic activation of links in Reflex is done via either restrictions [17] or activation conditions (the main difference between both being the time at which they are bound, either weaving or runtime). These conditions, evaluated on the host where operations occur, can now take the current host into account (obtained with `Reflex.getThisHost()`) in order to condition links to *dynamic properties* of the hosts in which their cut is realized. Dynamic activation of links in Reflex has been used to provide *context-aware aspects* [21], which could also be of interest in the context of distributed AOP.

4.2 Distributed Action

Parameterization extended. Passing parameters to metaobjects (*e.g.* `THIS` as in Sect. 3.2) makes it possible to define parameterized actions. A number of predefined parameters are provided beyond the `THIS`: method name, arguments, etc. Considering distribution, we add two predefined parameters: `HOST` and `HOSTNAME` to refer to the host (resp. its name) in which the cut is realized.

```
public class Tracer {
  public void log(Object aThis, RHost aHost){
    if("devel".equals(aHost.getProperties().get("type")))
      // do verbose logging
    else // do light logging
} }
```

Above is a tracer metaobject that accepts the current host as extra parameter, and performs verbose logging for development hosts, light logging otherwise.

Since a metaobject can execute remotely, the programmer needs control over *how* parameters are passed from the host where the operation occurrence is intercepted to the metaobject. The default Java RMI semantics is used (passing all objects by copy except remote ones), but in addition, Reflex makes it possible to explicitly state that a parameter must be passed by reference (a small Reflex library handles transparent remote invocations on any object). Below, we specify that the `THIS` must be passed *by reference* to the tracer:

```
Link trace = /* as before */;
trace.setCall("Tracer", "log", new ByRef(Parameter.THIS), Parameter.HOST);
```

Scope extended. The scope attribute of a link specifies the association scheme of metaobjects *w.r.t.* base entities involved in the cut. If it is *per object*, then each object involved in the cut has its own metaobject reference (which may point to the same metaobject), while if it is *per class*, each class has one reference to it, and if it is global, the link itself holds the reference shared among all objects and classes involved. Considering distribution, the `Scope.GLOBAL` attribute is renamed `Scope.HOST` in order to make clear that there is one global instance *per host*. In order to obtain a globally-unique metaobject, one simply needs to use explicit (remote) creation of the metaobject, as discussed below.

Instantiation extended. While the scope of a link determines the metaobject referencing scheme, instantiation addresses the bootstrapping of the metaobject reference. Reflex provides two alternatives for instantiation:

- explicit instantiation: the metaobject is manually instantiated before defining the link; the instance is shared among all entities involved in the cut.
- implicit instantiation: at link definition, the class of the metaobject is specified[3] so that, when first needed, a new metaobject instance is created and bound; subsequent invocations are performed on that metaobject instance.

For explicit instantiation, ReflexD provides a *remote object creation* service to create any object on any host, which returns a type-compatible reference to the remote object[4]. Below we remotely create a tracer on the `monitor` host, then interact with it (*e.g.* to configure it), before using it in the link definition:

```
RHost host = RHosts.get("178.1.2.3:4567", "monitor");
Tracer t = (Tracer) host.create("Tracer");
// ...interact with t...
Link trace = Links.get(fooCalls, t);
// ...configuration continued...
```

For implicit instantiation, the link definition must specify, in addition to the metaobject class to instantiate, the host on which the metaobject instance will reside. It can be either (Fig. 3):

- `ExecHost.THIS_HOST`: the current host, *i.e.* where the link is being defined;
- `ExecHost.APP_HOST`: the current application host, *i.e.* where the interception of operation occurrences occur;
- Any arbitrary host (with `new ExecHost(addr,name)`/`(aRHost)`).

For instance, using the following definition, if the link scope is *per object*, then any object involved in the cut of the link will have a dedicated tracer instance automatically created on the `monitor` host:

[3] using an `MODefinition.Class` object. Reflex also supports metaobject factories to bootstrap metaobject references [25], but we do not discuss them in this paper.

[4] Further remote interaction with the object via RMI is handled transparently.

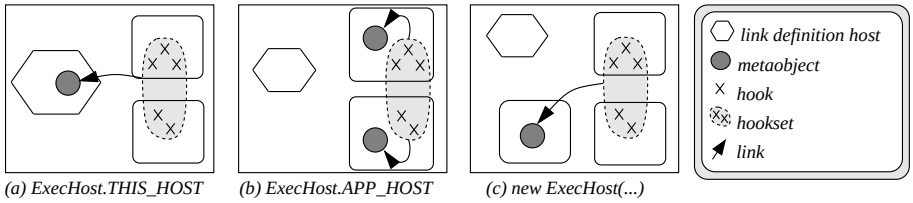

Fig. 3. Execution hosts.

```
RHost host = /* as above */;
Link trace = Links.get(fooCalls,
             new MODefinition.Class("Tracer", new ExecHost(host));
```

Conversely, using `ExecHost.APP_HOST` implies that each tracer instance will reside on the same host than the object in which the cut is realized.

4.3 Distributed Binding

The binding between a cut and an action in Reflex is an explicit entity: a link. So far, we have not explained how links are stored and applied.

Link definition, storage and application. Link definition can be done at runtime, or prior to executing the main program, by specifying *link providers* to invoke on startup. Link providers can either be plugins of aspect languages, or plain Java classes (*a.k.a. configuration classes*) defining one or more links; a (set of) configuration class(es) can be given on the command line as arguments to the Java agent of Reflex:

```
% java "--javaagent:reflex.jar= -lp class:Config1,Config2" Main
```

The above launches the `Main` program using Reflex, first performing the configuration in classes `Config1` and `Config2`. Links are stored in a local *link repository*. Then, upon class loading, the Reflex agent queries the link repository to determine whether any link applies to the class being loaded. If it does, then code transformation is performed before the class is finally loaded in the VM.

Definition, storage and application extended. In order to support a flexible distributed aspect infrastructure, ReflexD provides a *complete decoupling* of link definition, link storage, and link application. Hence the distributed architecture of ReflexD involves three types of hosts: *(1) Reflex hosts*, in which Reflex runs a program (possibly subject to links); *(2) aspect hosts* in which one or more link repositories are exposed, and to which Reflex hosts can connect; *(3)* any Java program running in any host can remotely populate link repositories. Such a decoupling is convenient to group links that can apply to a program according to some criteria, thereby raising the abstraction level of aspect configuration.

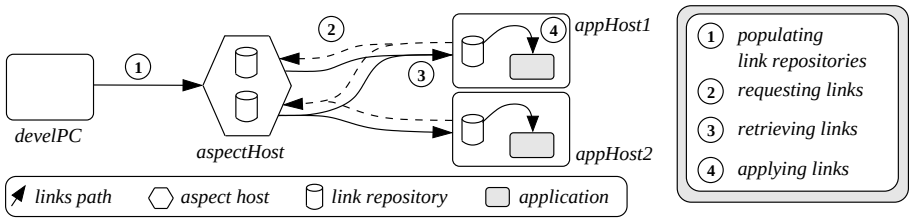

Fig. 4. Link repositories.

Illustration. Consider four machines (Fig. 4): an aspect host machine `aspectHost`, on which two link repositories are started, namely `debugLinks` and `prodLinks`, to hold debug links (*e.g.* logging) and production links (*e.g.* business observation relations), respectively; the developer's machine `develPC`, on which link definition is executed, populating both link repositories; and two Reflex hosts, `appHost1` and `appHost2` running the application.

First, on the `aspectHost` machine, the two repositories are started:

```
% java reflex.StartLinkRep debugLinks
% java reflex.StartLinkRep prodLinks
```

Then, on the `develPC` machine, two configuration classes `ConfDebug` and `ConfProd` are defined, and used to populate the corresponding repositories:

```
% java reflex.ExportToRep reflex://aspectHost/debugLinks ConfDebug
% java reflex.ExportToRep reflex://aspectHost/prodLinks ConfProd
```

Note that it is also possible to access a link repository programmatically, *e.g.*:

```
LinkRepository rep = LinkRepository.get("reflex://aspectHost/debugLinks");
rep.addLink(/* a link */);
rep.removeLink(/* a link */);
```

Finally, supposing the application in `appHost1` is deployed in a development environment, it is configured to use the links defined in both repositories:

```
% java "--javaagent:reflex.jar= -lp reflex://aspectHost/debugLinks,
                                   reflex://aspectHost/prodLinks" Main
```

If `appHost2` is deployed in a production environment, it is enough to remove the reference to the `debugLinks` repository in the command line above. No other modification is needed, and only production links will apply.

Runtime link manipulation. A feature of Reflex that we have not mentioned until now is the possibility to manipulate links at runtime [25]: *e.g.* changing the metaobject associated to a base entity for a given link, or changing the activation condition of a link. Note that the latter makes it possible to dynamically deploy/undeploy aspects. Maintaining consistency between changes made

to links in different hosts is done with a remote consistency framework developed with Reflex, which ReflexD makes great use of[5].

5 Distributed Control Flow

Control flow in aspect-oriented languages, as exemplified by AspectJ's `cflow` pointcut designator, is a very valuable feature that makes it possible to pick out execution points of interest provided they are in the control flow of others.

Control Flow. In Reflex, if a link depends on a control flow condition (*e.g.* log only top-level position changes on shape objects), it is subject to an activation condition, which checks the associated control flow condition. The control flow information has to explicitly exposed, by a dedicated link.

```
1 Hookset shapeMove = /* shape position changes */;
2 Link moveCflow = CFlow.get(shapeMove);
3 Link trace = Links.get(shapeMove, new Tracer());
4 trace.addActivation(new CFlow.IsNotBelow(moveCflow)); // dependency
```

The hookset corresponding to shape position changes is defined (1). This hookset is used in the two link definitions that follow. First, it is used to obtain a link that exposes control flow information (2). `CFlow.get` is a convenience method that returns a link matching the given hookset, to which a before-after meta-object called a `CFlowExposer` is bound. Such an exposer maintains a thread-local counter (or stack if context information must be kept) that keeps track of control flow. Overloaded versions of `CFlow.get` make it possible to explicitly pass the exposer instance to use, and to specify context information that must be collected by the exposer if any. The `trace` link also relies on the `shapeMove` hookset (3), and its activation depends on the control flow exposed by `moveCflow` (4): only calls that are *not below* that control flow will match (*i.e.* only top-level calls). Class `CFlow` offers other predefined activation conditions like `IsIn`, `IsOut`, and `IsBelow`.

Distributed Control Flow. Extending control flow to distributed systems is highly interesting, as it makes it possible to capture particular patterns of inter-host communications; *e.g.* trace all calls on a machine that are performed in the control flow of a call originating from another machine. There is an implementation challenge associated to distributed control flow: control flow information is intrinsically bound to a given thread, and thread identity is not preserved in a typical remote method invocation middleware like RMI. An alternative is to make use of a distributed call stack [1], however this raises other issues of efficiency. We rather adopt the same approach than in [15, 3]: custom socket

[5] Due to space limitations, we do not discuss these issues in detail, nor do we present the remote consistency framework and other elements of the implementation. More information on the Reflex website (http://reflex.dcc.uchile.cl).

implementations for RMI [20], which manage the propagation of thread-local information from one host to another in order to simulate the unicity of the caller thread. This solution works, but it is dependent on the RMI implementation.

This being said, a distributed control flow library for Reflex is provided, illustrated in the next section. The underlying details are transparently handled by Reflex. Finally, note that control flow as discussed here is only a particular case of what event collectors can expose: it is possible to provide event collectors for matching event sequences for *stateful aspects* [6], or to support more advanced control flow properties as in [7].

6 Application: An Adaptive Image Server

We now consider an image server: an `ImgServer` is an RMI object that delivers images stored in a storage area. Clients can (in parallel) request images by calling `getImg(name)`; the `ImgServer` object translates the image name to a path, and requests an `ImgFinder` to retrieve the actual bytes of the image.

We consider an *image quality adaptation* aspect, which, based on the available bandwidth of *each client*, returns a possibly lower quality image. The design of this aspect relies on distributed control flow with context exposure: when `ImgFinder.findImg()` is called *in the control flow* of a client call to `ImgServer.getImg()`, the *actual* bandwidth value at the client site is used to determine the quality of the returned image. Link definition code is as follows:

```
1  RHost server = /* retrieve server host object */;
2  CFlowExposer exposer =
3          (CFlowExposer) server.create("CFlowExposer");
4  Hookset clientCalls = /* call to ImgServer.getImg() in any client */;
5  Link callCflow = DCFlow.get(clientCalls, exposer, new BWParam());
6
7  Hookset findCalls = /* calls to ImgFinder.findImg() in ImgServer */;
8  Link adapt =
9      Links.get(findCalls, new MODefinition.Class("QualityAdapter", exposer);
10 adapt.setControl(Control.AROUND);
11 adapt.setCall("QualityAdapter", "getImg", new Parameter.Arg(0));
12 adapt.addActivation(new DCFlow.IsInside(callCflow)); //dependency
```

First, a link to expose control flow information from client calls is defined (1–5). We explicitly create an exposer metaobject on the server host (1–3), which will store the bandwidth value for a client in a thread-local when `ImgServer.getImg()` is called (4). The corresponding link is obtained by passing both the hookset and the exposer to `DCFlow.get`, as well as a custom parameter object `BWParam` that encapsulates the know-how for extracting bandwidth value (5). Then the link matching calls to `ImgFinder.findImg()` in the server is defined (7–12). A `QualityAdapter` object will be created (on the server), passing it the exposer as constructor parameter (9). The link is set to act *around* such calls (10), by invoking the `getImg` method of `QualityAdapter` with the first argument as parameter (*i.e.* the path of the image to find) (11). Finally, the control flow dependency is set (12): the adaptation link only applies if `findImg` is called in the control flow of a client call.

Class `QualityAdapter` is straightforward:

```
class QualityAdapter {
  QualityAdapter(CFlowExposer exp){ this.exp = exp; }
  byte[] getImg(String path){
    int bw = exp.getValue(0);
    if(bw < threshold){
      // check existence of low-quality img, generate it otherwise
      // proceed with modified path
    } else // proceed as normal
} }
```

This example demonstrates how one can concisely define a distributed aspect in ReflexD. The example uses distributed hooksets (`clientCalls` matches on any client host), remote actions (the event collector operates on the server host), and distributed control flow. Without distributed AOP, coding such an aspect requires to manually handle the distributed nature of the aspect.

7 Discussion

Distribution is an inherently large and complex topic. Although it seems that distributed AOP can help in tackling some of the challenges faced in distributed computing, it would be simplistic to claim that distributed aspects can turn the development of distributed programs into an easy go. A number of challenging issues for distributed AOP need to be further explored.

Scalability. Our experiments with ReflexD are, as of now, pretty small. The exercise of Section 6 is a valid proof of concept, showing the interest of distributed AOP versus a manually-distributed implementation. Still, distributed AOP can only get to be a convincing approach for distributed programming if its scalability to larger and far more complex scenarios can be shown. As a first step in this direction, Benavides *et al.* report on a successful larger scale study with replicated caches [4]. Finally, it has to be expected that larger experiments will be developed if distributed AOP attracts attention from the distributed computing community, thereby helping in shaping the future of distributed AOP.

Failures. A distinctive characteristic of distributed programming is partial failures of the system. Introducing an infrastructure for distributed aspects therefore adds a new dimension of possible partial failures: for instance, in the communication between the cut of an aspect and its associated action (back and forth, once for the call, once for the return), or in the communication with link repositories. There are several approaches to this issue. At the very least, it should be possible to guarantee that the behavior of the original application is preserved when communication with an aspect action fails. We are currently exploring this solution and possible variants. It is important to note that this concern is different from that of *handling* partial failures in a given application using aspects.

Performance. The use of distributed aspects ought also to be evaluated in the light of performance. As of today, we have not performed significant bench-

marks of ReflexD. Reflex as such is among the most efficient portable AOP implementations in Java [9]. Still, the ReflexD infrastructure introduces a number of possible overheads. A major source of potential overhead actually lies in the use of advanced control flow features in aspects: distributed control flow as presented previously, and most importantly aspects that rely on distributed event sequences [4], pose a challenge to efficient implementations. However, the recent achievements in optimizing (local) trace matching for aspects brings optimism in this regard [2].

8 Related Work

The issue of crosscutting concerns and code tangling related to distribution was first addressed in the literature by Lopes [14]: it is shown that dedicated aspect languages for handling concurrency and remote parameter passing strategies greatly improve understandability and maintainability of code. However no distributed aspects are considered. More recently, Soares *et al.* have reported on the use of AspectJ to encapsulate RMI code in aspects, showing that current AOP technologies (that do not support distributed AOP as such) require in-depth knowledge of the middleware (RMI) [18].

In the area of distributed AOP, three proposals relate to ours: JAC [16], DJcutter [15], and AWED [4][6] (previously known as Dhamaca [3]). DJcutter and JAC both introduced remote pointcuts, making it possible to specify on which hosts join points should be detected. Although JAC allows distributed aspect deployment to various containers with a consistency protocol between hosts, DJcutter adopts a centralized architecture with an aspect host where all aspects reside and advices are executed. This is in contrast with AWED and ReflexD, which make it possible to execute advices in (several) arbitrary host(s): multiple parallel advice execution in specific hosts is possible, and programmers can control where aspects are deployed. In this regard, ReflexD goes a step further than AWED by providing greater flexibility in the localization of advices (meta-objects), and by allowing to customize the remote parameter passing strategy for each parameter passed to a remote advice. Furthermore, compared to the centralized architecture of DJcutter, both AWED and ReflexD adopt a decentralized architecture. AWED only supports two deployment modes: local to the aspect definition host, or global to all hosts. Conversely, ReflexD is more flexible by supporting stand-alone link repositories to which a Reflex host can connect. JAC, AWED and ReflexD support dynamic deploy/undeploy of aspects with distributed effect.

Both DJcutter and AWED represent hosts as plain strings, whereas in ReflexD they are reified as `RHost` objects giving access to the system properties of the hosts. So groups of hosts, as provided in AWED, can be intensionally and dynamically defined in ReflexD. Since the fact that a host belongs to a group is

[6] The implementation of the AWED language is called DJasCo. In the following we simply refer to both the language and the DJasCo implementation as AWED.

just one kind of metadata that can be associated to it, the explicit representation of hosts as objects in ReflexD is more general and expressive.

An interesting feature of AWED is the possibility to control whether advice execution is done synchronously or asynchronously. This is something we have not considered yet, but which is clearly possible to achieve. As of now, advice execution is synchronous in ReflexD.

With respect to control flow, DJcutter, AWED and ReflexD adopt the same implementation strategy. However in ReflexD the use of custom sockets is completely hidden from the programmer. Finally, AWED supports distributed sequences of events for stateful aspects [6]. However, the AWED implementation does not handle the challenging issue of distributed time, so inconsistencies can occur when matching event sequences. At present stateful aspects have not been implemented in Reflex, but they can be supported via event collectors. Their correct semantics in a distributed setting remains a challenge for future research.

Finally, work on distributed AOP can be useful for a new generation of reflective middleware [12] based on AOP. ReflexD can be seen as an open middleware for distributed AOP, which in turn can be used in the implementation of adaptable middleware.

9 Conclusion

We have presented the extension of our work on versatile kernels for AOP to distributed systems, yielding a flexible and expressive infrastructure for distributed aspect-oriented languages and frameworks in Java. All dimensions of aspects have been revisited in the light of distribution, including distributed cut based on an extended reflective model, distributed action with fine-grained customizable parameter passing and flexible instantiation, and complete decoupling of definition, storage and application of aspects. We have illustrated the expressiveness of ReflexD with the provision of abstractions for distributed control flow and their application in an adaptive image server. Compared to other distributed AOP proposals, ReflexD provides more flexibility. Furthermore, although this paper does not focus on this issue, the fact that ReflexD is based on our work on AOP kernels implies that it is able to automatically detect interactions between (distributed) aspects, and provide expressive means for their resolution.

As regards future work, apart from extending the concrete syntax developed for Reflex to ReflexD, we plan to study the support for stateful aspects, and experiment with different aspect languages useful in a distributed setting, both general purpose (such as AWED) and domain-specific (such as SOM [5] for scheduling of concurrent requests).

Acknowledgments. We thank Guillaume Pothier and Ángel Núñez for their work on ReflexD, and Leonardo Rodríguez for his comments on a draft of this paper, and the anonymous DAIS reviewers for their useful remarks.
É. Tanter is financed by the Milenium Nucleous Center for Web Research, Grant

P01-029-F, Mideplan, Chile. Work partially-funded by the EU Network of Excellence CoreGRID and ITCC Chile-Korea.

References

1. Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *International Conference on Parallel Processing*, pages 4–11, 1999.
2. P. Avgustinov, J. Tibble, E. Bodden, O. Lhoták, L. Hendren, O. de Moor, N. Ongkingco, and G. Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, abc Group, Mar. 2006.
3. L. D. Benavides Navarro. Dhamaca – an aspect-oriented language for explicit distributed programming. Master's thesis, Vrije Universiteit Brussel, Belgium, 2005.
4. L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 51–62, Bonn, Germany, Mar. 2006. ACM Press.
5. D. Caromel, L. Mateu, and É. Tanter. Sequential object monitors. In M. Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, number 3086 in Lecture Notes in Computer Science, pages 316–340, Oslo, Norway, June 2004. Springer-Verlag.
6. R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr [13], pages 141–150.
7. R. Douence and L. Teboul. A pointcut language for control-flow. In G. Karsai and E. Visser, editors, *Proceedings of the 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *Lecture Notes in Computer Science*, pages 95–114, Vancouver, Canada, Oct. 2004. Springer-Verlag.
8. T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), Oct. 2001.
9. M. Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Technischen Universität Darmstadt, Germany, Dec. 2005.
10. JBoss AOP website, 2004. http://www.jboss.org/developers/projects/jboss/aop.
11. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.
12. F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for distributed middleware. *Communications of the ACM*, 45(6):33–38, 2002.
13. K. Lieberherr, editor. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, Mar. 2004. ACM Press.
14. C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.
15. M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut – a language construct for distributed AOP. In Lieberherr [13], pages 7–15.
16. R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC: an aspect-oriented distributed dynamic framework. *Software Practice and Experience*, 34(12):1119–1148, 2004.

17. L. Rodríguez, É. Tanter, and J. Noyé. Supporting dynamic crosscutting with partial behavioral reflection: a case study. In *Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, Arica, Chile, Nov. 2004. IEEE Computer Society Press.
18. S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2002)*, pages 174–190, Seattle, Washington, USA, Nov. 2002. ACM Press. ACM SIGPLAN Notices, 37(11).
19. SUN Microsystems. *Remote Method Invocation*, 1998.
20. SUN Microsystems. *Using custom socket factories with Java RMI*, 2005.
21. É. Tanter. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, Lecture Notes in Computer Science, Vienna, Austria, Mar. 2006. Springer-Verlag. To appear.
22. É. Tanter. An extensible kernel language for AOP. In *Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages*, Bonn, Germany, 2006.
23. É. Tanter and J. Noyé. Motivation and requirements for a versatile AOP kernel. In *1st European Interactive Workshop on Aspects in Software (EIWAS 2004)*, Berlin, Germany, Sept. 2004.
24. É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In R. Glück and M. Lowry, editors, *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188, Tallinn, Estonia, Sept./Oct. 2005. Springer-Verlag.
25. É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In R. Crocker and G. L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, CA, USA, Oct. 2003. ACM Press. ACM SIGPLAN Notices, 38(11).