# Distributed Separation of Concerns with *Aspect Components*

Renaud Pawlak, Laurence Duchien, Gerard Florin, Laurent Martelli[1], Lionel Seinturier[2]
*Laboratoire CEDRIC - 292, rue St Martin, Fr 75141 PARIS Cedex 03*
[1]*AOPSYS - 5, rue Brown Séquard, Fr 75015 PARIS*
[2]*Laboratoire LIP6 - 4, place Jussieu, Fr 75252 PARIS, Cedex 05*
*{pawlak,duchien,florin}@cnam.fr, Laurent.Martelli@aopsys.com, Lionel.Seinturier@lip6.fr*

## Abstract

*This paper presents A-TOS, an aspect-oriented reflective middleware for distributed programming. It provides a very special kind of entities called* aspect components *that implement global and transversal properties of (distributed) applications like* security, fault tolerance, transactions, *and so on. Since the application code does not directly refer to the* aspect components, *A-TOS achieves clean and powerful* separation of concerns *based on a wrapping composition model. Its adaptability and aspect distribution capabilities make it well suited to aspect-oriented programming in a distributed environment.*

## 1. Introduction

During the last decade, needs for constructing applications, and especially distributed ones, in a consistent, secure, controllable, and reusable way have consequently increased. Object and component-based solutions like CORBA or DCOM provide frameworks that allow the interoperability of a set of distributed objects or components. However, because those programming solutions and related methods focus on components assemble, they lack a global vision of the final behavior and properties of so-constructed applications. As a consequence, changing a global property — like an ordered calling policy — often means an entire re-thinking of the application.

Recent trends in computer science research are tightly coupled with this problem and show that one of the most crucial issue in (distributed) software development consists in finding an accurate and consistent means to achieve *separation of concerns* within application programs. This separation of concerns, praised by the Aspect Oriented Programming approach (AOP) [11] should allow the programmer to tackle more complex problems by providing clear separation between the component (or base) program that provides the basic functional behavior, and some clearly identified aspects of the program that affect the base program performance or semantics in a systematic way [11].

In distributed applications, separation of concerns needs are coupled with some global or transversal properties that spread over the application entities, and consequently over the machine boundaries. However, with existing generic AOP languages, programming distributed aspects can be quite tricky because aspects languages generally relate to local base program entities (e.g. classes).

To solve this issue, we present A-TOS (stands for Aspect-TOS) [18], a general purpose aspect-oriented reflective middleware. A-TOS defines new kind of software entities for AOP called ***aspect components.*** Those special components can be seen as an extension of regular components but with a global semantics. Indeed, their attributes and methods specify some transversal properties on bunches of objects that are not necessarily localized on the same host.

Section 2 discusses available techniques and choices for *separation of concerns* when programming distributed applications. Section 3 presents the A-TOS solution and explains the advantage compared to other *separation of concerns* means. Section 4 presents a real-life distributed application example using *aspect components.* Last section compares A-TOS to some related approaches.

## 2. Providing distributed separation of concerns

### 2.1. AOP issues and definitions

AD-TOS intends to provide aspect-oriented features for distributed programs. It means that it must achieve *separation of concerns* within a distributed environment for distributed applications.

According to AOP guidelines [1], *separation of concerns* implies that the functional program should not directly access well-identified aspect-related primitives, like multi-threading, broadcasting, or synchronization libraries. Instead of these direct accesses, the programmer should be able to reverse the utilization dependence, so that the aspects have a sufficient knowledge of the base application to be able to put the aspects semantics into the application (in the AOP terminology, the aspect is said to be *woven* with the application). Thus, since the functional code is independent from the aspect transversal semantics, it is much more reusable and adaptable. In AOP, a piece of base program knowledge is called a *join point* and can be a function, an object, a class name, or any information about the base program.

However, several techniques to achieve *separation of concerns* compete. Subsequently, we present some existing solutions and briefly evaluate them regarding our constraints.

### 2.2. Basic constraints

When achieving aspect-oriented distributed programming, three main properties of the aspects are especially needed:

- **Adaptability**: because of the natural environment variations of a distributed program, an aspect that provides a global policy has to be easily removed and replaced by another, and this, without stopping a system that is most of the time shared by a great number of users.
- **Distribution abilities**: since they have to be applied on distributed applications, aspects must also be able to be distributed. Thus, an object-like structuring of the aspect code is desirable so that the aspect can be achieved by a set of interoperating distributed objects.
- **Easy composition**: the aspect system must be able to easily handle the fact that many different aspects cohabit within the same system.

Notice that *adaptability* and *easy composition* properties are also desirable for non-distributed systems. However, there are much more crucial (in fact they are mandatory) in distributed systems (as the Internet) where environment is prone to change.

## 2.3. Three existing solutions for separation of concerns

**2.3.1. Inheritance-based solutions:** The first one is based on object-oriented frameworks using *inheritance* and *polymorphism* (see, for instance, the *visitor pattern* [6]). This technique provides *ad hoc* solutions and inherently presents some limitations due to the well-known problem of *inheritance anomaly* [14] (because of this problem, an aspect-related policy such as synchronization cannot be reused as is when subclassing). Moreover, complex inheritance schemes are needed if one wants the system to be flexible at run time, implying hardly understandable programs and bad distribution capabilities (since a lot of classes would need to be distributed).

**2.3.2. Transformative solutions:** A second kind of solution is known as *transformative* solutions (based on program transformation or interpretation). This solution consists in implementing an aspect as a meta-program (or an interpreter) that is applied to the base program to produce a new program with new semantics as an output. Transformative solutions can be sub-classed in three variants.

- *Integrated transformative solutions* that add into the language some specific keywords or instructions to parameterize the language interpreter regarding an aspect (e.g. Fortran HPF annotations, the *synchronized* keyword in Java, pragmas in C/C++).
- *Composition-based transformative* solutions where code transformations are especially specified as existing code composition statements (Subject-Oriented Programming [9], the AspectJ language [2]).
- *Reflective*[1] *transformative solutions* where the structure and/or syntax of the language is reified within a meta-model, most of the time at the compile time. Examples of this technique are OpenC++v2 [5], OpenJava [17] (for a structural approach), and Iguana [8] (for a more syntactical approach).

Despite those solutions are really efficient, it is quite tricky to compose different aspects (because they can transform the base program in such a way that another aspect will not be able to apply consistently to the output program). Moreover, distribution and adaptation capabilities are very weak since it is difficult in practice to apply the transformative algorithm to a running distributed program.

**2.3.3. Event-based solutions:** Last kind of solutions are *event-based* solutions. Those kind of systems link the functional code to the aspect code by binding some aspect entry points to well-defined base-program events like *method invocation*, *attribute accessing*, or *object migration*. Event-based solutions for separation of concerns are by essence reflective since the events give to the receiver some indications about what is happening in the system so that the receiver can "reason" about it, consistently to a reflective system definition. As a consequence, those solutions are widely used in metaobject-based reflective

[1] A reflective system is a system that is able to reason about itself by accessing (introspection) and modifying (intercession) chosen parts of its internal structures.

languages (in [4] events are sent to the metaobject on *message arrival*, *object creation*, and *attribute accessing*), metaclass-based reflective languages (in OpenC++v2, events are sent to the metaclasses when the parser recognizes special statements like attributes or methods declarations) and reflective operating systems.

Metaobject-based solutions are well fitted to the constraints enumerated in 2.2, and to distributed separation of concerns. However, most of them provide very poor introspection an intercession features, implying limited and somewhat insufficient reflective means.

Metaclass-based solutions provide a much more powerful meta-programming framework that allows deep structural changes of the base program. However, because those changes are centralized in metaclasses (this means that the base-program scope control is the class and not the object), they are not suitable for a distributed environment (where you need a control per-object).

## 3. A-TOS solution: the aspect components

Subsequently, we present our solution for separation of concerns and show that it meets the requirements defined in section 2. This solution introduces *aspect components* that are used to implement aspects within (distributed) object-oriented applications.

### 3.1. Aspect component classes

In the same way regular objects are described in classes, *aspect components* are described in *aspect component classes* which is a specialization of a regular class but with a global semantics (see figure 1). Thus, an *aspect component class* (*ac-class* for short) can define slots and functions that describe global properties or behaviors that can be pushed into a given application. Slots values should also contain the *join points* (see section 2.1) descriptions so that the aspect component will be able to weave with the base program.

One important point of *ac-classes* (see figure 1) is that they can contain other classes or objects (it is also a container). Thus, global classes and wrapper classes (see the next sections for details) used to define an aspect semantics are defined within the *ac-class*. For instance, if we need to add an aspect that implements persistence, we define a *Storage* class that is part of the *Storage* ac-class.

*Aspect component classes* should at least define two functions:

1. ***init(join_points)***: this function initializes the aspect component when it is created (it is automatically called when a new ac-instance is created). In general, *init* sets the join points values so that they are not hard-coded within the aspect component code and apply to several base programs.
2. ***weave(target_module)***: this function is used to apply the aspect semantics to a base program (that is usually contained in the target module). The weaving process mostly consists in wrapping well-chosen objects of the base program with some wrappers defined in the aspect component.

### 3.2. Programming ac-classes: wrappers and structural reflection

A-TOS combines metaobject and metaclass solutions in order to keep the adaptability and distribution abilities provided by the former and the powerful reflective features provided

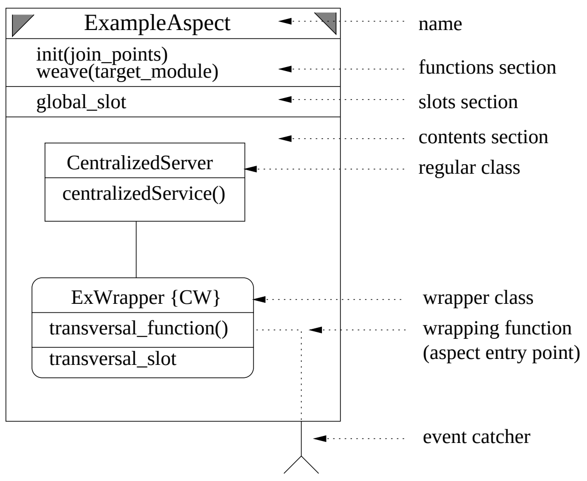

**Figure 1. An aspect component class.**

by the latter.

Thus, A-TOS first key feature when programming are wrappers that are based on runtime message arrival reification (to be related to metaobject based solutions). Wrappers are used at runtime to change the base program semantics by adding some code before and after the original code. Since it is an event-based wrapping technique, wrappers can be easily added and removed at runtime and thus provide adaptability and flexibility. When wrapping the base objects with wrappers, any wrapping function is said to be an *entry point* and is automatically associated with an aspect component *event catcher* (see the *transversal_function* in figure 1).

A-TOS second key feature is metaclass-based structural reflection (i.e. class definitions are read/write available in metaclasses [7][10]) that allows aspect components to easily access and modify the base program entities definitions. This feature is mostly used during the weaving/unweaving process to get some precise information on the base program and to transform and/or wrap the base objects.

## 3.3. Composition

Many wrappers (from several aspects) can wrap a base object. Thus, when a function of this base object is called, all the wrapping functions are called before it and can perform some treatments like tracing, authentication, and so on. Everything happens as if the message goes through a set of functions:
*message* → *authentication* → *synchronization* → *persistence* → *base function*

One of the interesting point in A-TOS, is that it provides some support for wrappers composition when you use several wrappers to wrap the same base function. Indeed, since the wrappers are sequentially called, the composition problematic is tightly linked to the calling order of the wrappers. For example, a wrapper that is used for authentication must always be called before a wrapper that implements persistence. If not, the base object changes might be saved on the disk even if the authentication fails.

To help solving this issue, A-TOS provides a framework that automatically orders the wrapper regarding a wrapper classification (wrappers can be mandatory and called first, conditional and called second, or exclusive and called last) and some calling priorities (when the order is not automatically decidable). For further insights on the solution, we encourage the reader to take a look at [15] where this feature is discussed.

## 3.4. Example

Let's take the example of a *Stack* class defined as follows[2] (see the A-TOS tutorial at [18] for a complete example):

```
Program new BaseProgram {
  Class new Stack {
    slot list elts {}
    func void push {elt} { ... }
    func elt pop {} { ... }
} }
% Stack new s
% s push 1
% s pop ==> 1
```

Suppose now that you need to limit the number of elements in the stack in some particular situations. However, for reusing and simplicity sake, you would like to keep the *Stack* code clean from any change. You also do not want to multiply the stack subclasses for such a kind of problem that does not reflect a lot of interesting functional points.

Thus, a simple and clean way to add a length control feature is to program an aspect component that is used to check the *'elts'* length.

```
Aspectcomponent new LengthControl {
  slot Object object_to_control
  slot Func func_to_control
  slot Slot slot_to_control
  # initializes the join points...
  func void init {o f s} { # sets the slots values... }
  # called by the ac-user weaving is needed...
  func void weave {baseprog} {
    # creates a new wrapper...
    LengthControlWrapper new lcw
    # the new wrapper wraps the function to control...
    lcw wrap $object_to_control {{controlLength $func_to_control OW 20}}
  }
  # defines the wrapper class...
  WrapperClass new LengthControlWrapper {
    # the default maximum length is 2...
    slot integer max_length 1
    # 'reflect' calls the wrapper object ($component)
    func scalar controlLength {} {
      if {[llength [$component slotValue $slot_to_control]] < $max_length } {
        this reflect $component $caller_component $func_name $func_args
      } else { error "Stack is full!" }
} } }
```

[2]Here is a brief TOS syntax overview. *'X* ***new*** *x [<definition>]'* creates a new object *'x'* that is an instance of *'X'*. *'**slot** <type> <name> <default_value>'* and *'**func** <type> <name> <b dy>'* declare new slots and functions in a class, wrapper class, or ac-class definition. *'<object> <func_name> [<args>]'* calls a function on an object. *'**$**<slot_name>'* gets a slot value. *'**[**<expr>**]**'* replaces the expression by its evaluation within the current context. *'**;**'* is the command separator.

You can now instantiate it and weave it into the base program. Notice that you give the join points to the aspect component so that it can weave properly (on an object called *'s'* and control the *'elts'* slot when *'push'* is called).

```
% LengthControl new lc {s push elts}
% lc weave BaseProgram
```

Finally, the *'lc'* aspect component changes the *'s'* stack behavior as follows:

```
% s push 1
% s push 1 ==> error: Stack is full!
```

## 4. Aspect components application: a concrete example

### 4.1. The agenda at a glance

Let us imagine an agenda application. This application program can be composed of three classes: a *User* class, that defines the users of the agenda objects, an *Agenda* class that implements all the functions to make some appointments with other users, and an *Appointment* class that defines the appointment times and members. Although this application semantics seem clear at first sight, a deeper analysis shows that some aspects can be quite complex and prone to change:

- **Security**. Shall we implement a Capability-Based system (e.g., Amoeba [16])? A Kerberos-like authentication algorithm [12]?
- **Distribution**. Is it a local application? Do we need some client-server? Do we need some replication algorithm for fault-tolerance or performance matter?
- **Appointment agreement**. How do users agree when making an appointment? Do they send an email to notify each other? What happens when a user makes an appointment with a user that already has an agreement with another user at the same time?

A major advantage brought by the AOP is to let the programmer think about the base problem in a totally independent way from the aspects ones. Thus, we can easily implement a first prototype of the application without even addressing the issues mentioned above. In a second time, the different aspect components (each one defines a global/distributed solution for one of the three issues) can be put into (*woven with*) the base program without changing a line of code.

### 4.2. Security and appointment aspect components

Figure 2 shows an implementation of an agenda application with two simple aspects (described in aspect component classes): a Kerberos-based authentication aspect and an appointment agreement policy. We can see that the authentication algorithm can be described in a totally independent way from the base program. The *KeyServer* class (that is a sole instance global class) knows global information: the private key (or password) of all the clients (identified by login) and of all the applications (identified by name) — to one private key corresponds one application or one client. The authentication algorithm can be found in [12].

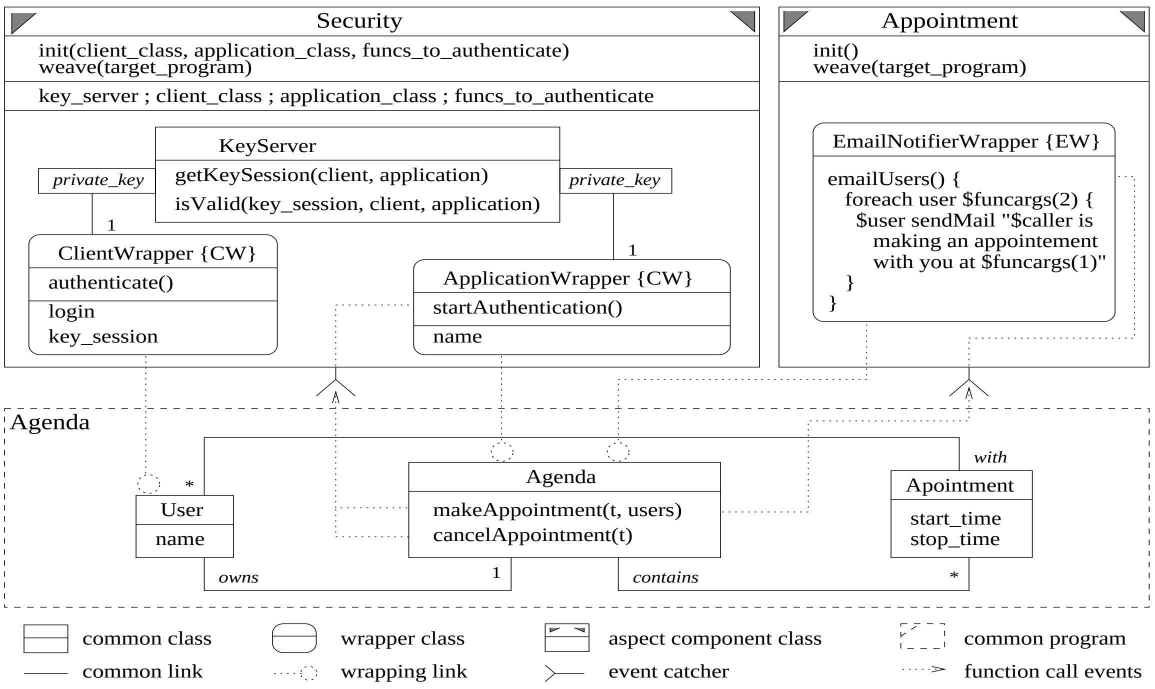

**Figure 2. The agenda example, and its security and appointment aspects.**

To weave this algorithm with the *Agenda* program, we simply need to wrap all the *User* instances with *ClientWrapper* instances and all the *Agenda* instance with *ApplicationWrapper* instances. In A-TOS, this weaving policy is programmed in the *weave* function:

```
SecurityAspect::weave {target_program} {
  # for each object in the program we weave...
  foreach curobj [$target_module objects] {
    switch [$curobj class] {
      # if this object is of a client type (defined by a join point in init)...
      $client_class {
        # then wrap it with the appropriate wrapper...
        [new ClientWrapper {}] wrap $curobj {}
      }
      # if this object is of the application type...
      $application_class {
        # wrap the functions to authenticate...
        [new ApplicationWrapper {}] wrap $curobj
          {{startAuthentication $funcs_to_authenticate OW 0}}
} } } }
```

By choice, the appointment agreement *aspect component class* is even more simple. In fact, we only decide to email all the participants of a newly created appointment. As one can expect, the weaving policy of the aspect component consists in wrapping the *Agenda* instances with a (unique) *EmailNotifierWrapper* instance.

Finally, the following program runs a new agenda application and modify its semantics, first by adding authentication features and second by adding an email notification when a new appointment is created:

```
Agenda run [Module new agenda_program]
```

```
SecurityAspect weave agenda_program
AppointmentAspect weave agenda_program
```

### 4.3. Adding distribution with D-TOS

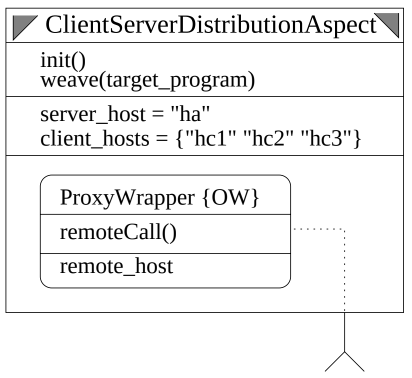

**Figure 3. A client/server distribution aspect component class.**

T ofulfill the previous example and present all the A-TOS capabilities w eneed to add a *Distribution asp ectcomponent* and w eav eit with the *Agenda* program. The D-TOS extension pro videssome useful functionalities lik e object migration/copying services and some common wrappers that are v ery useful to seamless distributed programming. Thus, an aspect component for distribution is very easily programmable as long as the distribution scheme is simple (of course, *sep ar ation of concerns* does not reduce the inherent complexity of an aspect — it reduces the aspect program in tegration complexity).

F or example, we can choose a simple client/server distributed architecture as follows:

- the agenda is centralized on a unique host *ha*,
- clien ts (users) are distributed on hosts *hc1*, *hc2*, and *hc3*.

For this v ery simple scheme, w e only need a *proxy wrapper* (proxies are a v ery powerful model for distribution) that can make the *A genda* class remotely used from the clients hosts. The *Pr oxyWrapper* is one of the useful wrapper furnished by D-TOS (see the D-TOS tutorial at [18]). Thus, a *DistributionAspect* component would contain *ProxyWrapper* and define the *init/weave* functions (see figure 3). We shortly define the *we ave* function (for readability reasons we hard-code *Agenda* and *User* join points).

```
DistributionAspect:: we ave{target_module} {
  if {[Dtos localHost] == $server_host} {
    # we create a new proxy wraper
    set a_proxy [ProxyWrapper new {} {set remote_host=$serve r_ host} ]
    # we wrap the unique instance of the Agenda class with the wrapper
    $a_proxy wrap [Agenda instances] {{remoteCall * OW 0}}
    # we wrap the User instances (for simplicity, we suppose there
    # is 1 user on each site, called aUser[1,2,3])
    set i 1
    while {$i <= 3} {
      set a_proxy [ProxyWrapper new {} {set remote_host=hc$i}]
      $a_proxy wrap aUser$i {{remoteCall * OW 50}}
```

```
        incr i
      }
      # we copy the program on each client host
      Dtos copy $target_module hc1 hc2 hc3
    } else {
      # copy all on 'ha' and do the same
  }  }
```

Figure 4 shows the running application resulting from this simple aspect component (wov en with *"ClientServerDistributionAspect weave agenda_program"*) (for simplicity sake, we hav e omitted the security aspect — despite there would be no problem composing it). It is quite impressive because the *copy* function duplicates all the needed objects and modules that are the instances of the classes described in figure 2. On the clien tssides, an ycall to the agenda is made remote b y the wrapper, and similarly, on the server side, any call made to the users is made remote and directed to the proper client host.

In figure 4, we can see what happens when *aUser1*, makes an appointment with *aUser3*.

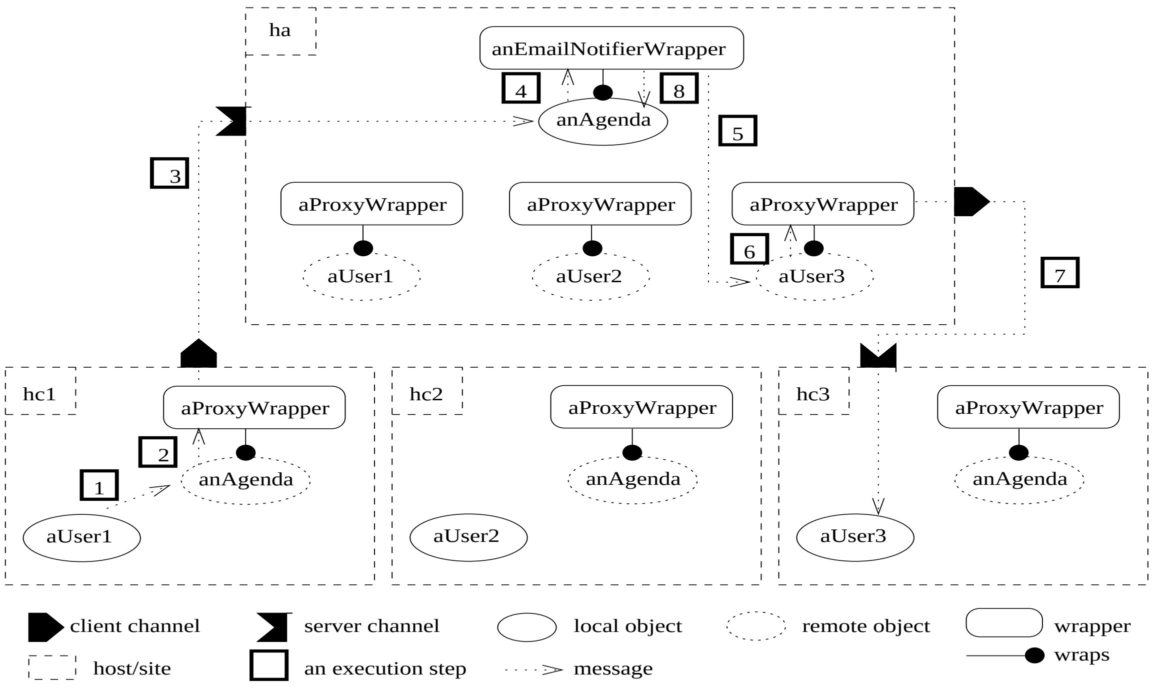

**Figure 4. The agenda application objects and wrappers after applying (weaving) the client/server distribution aspect component.**

1. *aUser1* sends *<makeAppointment '10H00' 'aUser3' to anAgenda>*
2. *anAgenda* is wrapped by *aProxyWrapper* ⇒ the message arrival is catched
3. *aProxyWrapper* sends the message do the *A genda*instance located on *ha* (through the TOS Communication Protocol built on TCP soc kets— provides client/server communication channels)
4. *anEmailNotifierWrapper* catc hes the message
5. It sends a notification message to *aUser3*

6. *aProxyWrapper* catc hes the message
7. It sends it to *aUser3* located on *hc3*
8. *anEmailNotifierWrapper* actually makes the appointment

## 5. Related work

Metaobject-based approaches can be closely related to the A-TOS approach. Indeed, A-TOS wrappers can be seen as metaobjects as defined in [4]. However, A-TOS framework is only based on message reification and allows the user to define its own reification interface (contrary to metaobjects). This makes the A-TOS wrappers muc h more easy to specify and understand.

A-TOS wrappers can also be compared to the *Composition Filters* (CF) approach [3] since an A-TOS *message-event-based wr apper* can be seen as a *{message filter + internal object}* couple. By separating *inc oming/outgoing message filters* and the objects where they dispatch the messages, CF can be seen as a more general and flexible approach, but, since wrappers can also easily be used as filters, w eclaim that the tw oapproaches are fundamentally equivalent.

Another v ery close approach is the w orkon aspectual components [13]. They pro videa larger-than-class aspect structure than can be seen as aspect components. However, since they do not use an even t-based framework, they are less flexible than our approach.

Generally speaking, A-TOS real addins to those approaches are an easy class-level specification possibility (see figure 2) and the *asp ectcomponents* described in section 4.

## Conclusion and future work

In this paper, w epresent A-TOS, a reflective framework for aspect-oriented distributed programming based on *aspect components*. *Aspect components* are some special kind of objects that can modify the base program objects semantics b y making them wrapped b y *wrappers*. They represent some global and transversal properties of the base program that can be centralized (mostly in regular objects) or distributed (mostly in wrappers) depending on the *asp ectcomponent* specification (i.e. its *aspect component class* definition).

Therefore, w e believe that the *Aspect Component* approach proposed b y A-TOS can be easily coupled or in tegrated in class-level specification formalisms lik e UML to allow distributed (and also non-distributed) application designers to actually easily handle *separation of concerns* and thus greatly enhance industrial software quality. Since they can be added and remov ed at runtime, aspect-components-based systems, languages and methodologies, w ouldallo w complex and adaptable distributed softw aredevelopment with better understandable/maintenable code and shorter refinement cycles.

A-TOS is a prototype of such a system and is av ailable at [18]. It implements in a naive manner most of the features described in this article. It can be seen as a first step to an *aspect component* based tool suite.

## References

[1] AOP. Aspect Oriented Programming home page.
`http://www.parc.xerox.com/csl/projects/aop/`.

[2] AspectJ. AspectJ home page.
`http://aspectj.org/`.

[3] CF. TRESE project homepage.
`http://wwwtrese.cs.utwente.nl/`.

[4] S. Chiba. Open C++ release 1.2 programmer's guide. Technical Report 9303, Department of Information Science, University of Tokyo, 1993.
`ftp://ftp.is.s.u-tokyo.ac.jp/pub/techreports/TR93-03-letter.ps.Z`.

[5] S. Chiba. A metaobject protocol for C++. In *Proceedings of OOPSLA'95*, volume 30 of *SIGPLAN Notices*, pages 285–299. ACM Press, October 1995.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley, 1989.

[8] B. Gowing and V. Cahill. Meta-Object Protocols for C++: The Iguana Approach. In *Proceedings of Reflection'96*, 1996.

[9] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure object). In *Proceedings of OOPSLA'93*. ACM Press, 1993.

[10] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the ECOOP'97*, 1997.

[12] J. Kohl and C. Neuman. The kerberos network authentification service (v5). *IETF Network Working Group, Request for Comments 1510*, September 1993.

[13] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. Technical report, Northeastern University's College of Computer Science, 1999.

[14] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[15] R. Pawlak, L. Duchien, and G. Florin. An automatic aspect weaver with a reflective programming language. In *Reflection'99*, July 1999.

[16] A.S. Tanenbaum, S. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th European Conference on Distr. Comp. Sys.*, 1986.

[17] M. Tatsubori and S. Chiba. *OpenJava 1.0 API and Specification*. Programming Language Lab., University of Tsukuba, 1998.
`http://www.softlab.is.tsukuba.ac.jp/~mich/openjava`.

[18] TOS. The TOS project main page.
`http://cedric.cnam.fr/personne/pawlak/tos.html`.