Towards a robust model for distributed aspects

Ismael Mejia, Mario Südholt

Towards a robust model for distributed aspects

Ismael Mejía, Mario Südholt
ASCOLA group; EMN-INRIA, LINA
Dépt. Informatique. École des Mines de Nantes
4 rue Alfred Kastler, 44307 NANTES Cedex 3, France
{ismael.mejia, mario.sudholt}@mines-nantes.fr

ABSTRACT

In this paper, we present some of the problems we have found in distributed aspect models and introduce a set of criteria that we consider necessary for a robust distributed aspect system. We outline of a first version of model based on aspects and actors capable of meeting these criteria.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks, Patterns*

General Terms

Design, Languages

Keywords

Actor model, Aspect oriented programming, Distributed aspects, Distributed systems, Invasive software composition

1. INTRODUCTION

Novel approaches to compose and create applications have often been developed in response to the increasing complexity of software systems. These approaches typically rely on new ways to abstract the structure of the programs with components or services which are managed by specialized software called middleware *e.g.,* JavaEE Application servers, or the OSGi service platform.

Application developers rely on these abstractions and their corresponding tools because they help to build software applications in a more efficient way as they can concentrate on specific application needs without spending time on the details of other common requirements which are supplied by their composition model and its middleware implementation. For example, if a programmer follows the contract of the JavaEE specification when he implements a module (EJB) the application server guarantees some properties like transactionality or security. Middleware also offers tools to monitor and evolve a system during his lifetime *e.g.,* tracing, profiling, etc.

Aspect Oriented Programing (AOP) is a complementary composition methodology introduced by Kiczales in [8] as a partial solution to the problem of correct modularization of application concerns that crosscut many of the decomposed parts of a system. The best studied model of AOP is AspectJ [9] which is based in two mechanisms: (1) a rich declarative language which can identify the particular scattered/tangled points that are present in the application and (2) the ability to alter the current application semantics with a method-like construct called advice. AOP has been embraced immediately by system software developers as many tasks of middleware, specifically application servers fit exactly with an aspect-based adaptation model. Common scenarios where this applies are the validation of security credentials, tracing, and dependency injection.

Nowadays, most of the traditional isolated computing environments have evolved into distributed and mobile contexts. Indeed, distributed computing is now the main form of computing due to the widespread availability of the Internet and the introduction of wireless network support in mobile phones and other appliances. Distributed Aspects have been developed to augment the expressivity of the traditional aspect model with explicit distribution constructs which are able to detect join-points and execute advice on different machines in order to support the particular needs of distributed systems (*e.g.,* coordination, concurrency, etc).

Today many middleware platforms include constructs based on AOP, such as Spring AOP, or the EJB interceptors in the case of JavaEE application servers. Nevertheless, there is not any existing middleware platform which includes distributed AOP support, even if many middleware platforms support explicit distribution requirements via message brokers, or other mechanisms. We argue that this is in part because current models for distributed aspects have been focused on the language expressivity, but not much in the the particular characteristics of the dynamic, heterogenous and asymmetric environments commonly present in distributed systems.

In this paper we study in detail some of the problems we have found through our experimentation with different distributed aspect approaches. We present and motivate why the actors models [7],[1] seem to be a useful complement to distributed aspects. We also introduce our first approach for a design based in actors. We expect that this integration paves a way for the construction of a robust and resilient distributed aspect model that can be integrated into existing

MISS '11 March 21-25, 2011, Pernambuco, Brazil

infrastructure software, and software development tools.

This paper is structured as follows. In Sec. 2 we introduce the concepts of distributed aspects, its motivation and some of the problems of adding distribution to an aspect model. In Sec. 3 we present a set of desirable characteristics for a reliable distributed aspect model, as well as our design to accomplish those characteristics inspired by the actor model. Finally in Sec. 4 we present the conclusions and some ideas for future work.

2. DISTRIBUTED ASPECTS: MODELS AND PROBLEMS

The evolution of distributed systems, and in particular the evolution of legacy systems is a difficult task. In this context it is common that some application concerns become distributed and hard to manage, *e.g.,* the authentication and access rights of a user between multiple parts of a system, or the tracing of the different operations and systems involved in an electronic business transaction. Some of these tasks are commonly treated via complex refactorings of the existing applications, or via time demanding standarization processes to achieve a certain level of homogeneity between applications. Distributed aspects offer a rich and more expressive model to treat such evolution problems and enable the modularization and isolation of concerns in a nice way.

The relation between AOP and distribution was first studied as part of the work of Soares et al. [14]. They presented a use case of the use of aspects to modularize the distribution concern of an application. This interesting use of AOP is not what we refer to as distributed aspects, but it paved a way to the analysis of the relation of aspects and distribution. Distributed aspects are an approach to have a more expressive and sophisticated aspect model to deal with the issues of the introduction of communications in software evolution.

The first model that introduced specific constructs for distributed AOP was proposed by Pawlak et al [13]. They introduced a group of re-usable aspect components that defined distribution properties deployed in remote instances, and included broadcasting between remote objects. Nishizawa et al. [12], introduced the concept of remote point-cuts to identify join points on the execution of remote hosts, but used a centralized advice server. Benavides et al. [10] proposed a more expressive language and model called AWED that supported not only remote pointcuts, but remote advices and asynchronous communications. Another interesting model presented by Tanter et al. [15] takes a different approach: aspects travel with the requests between different hosts and alter their semantics as they go by.

In the presence of distribution, aspects can be seen as an event-driven model: join-points are implicit events matched through a point-cut language, and advice reacts to those events via event handlers. However, a big difference between distributed event systems and distributed aspects is their goal. Event-driven systems model the flow and reaction of programs via events and handlers. Aspects care about the set of points in the system where particular events happen in order to inject new functionalities via meta-programming techniques which alter the semantics of the existing program.

A model of distributed aspects share the advantages but unfortunately the problems of both worlds. First, aspects, and in general invasive composition approaches have been criticized for being too powerful and unrestricted which can lead to errors when the semantics of the original programs is modified in unexpected ways. Second, distribution adds a new set of issues, *e.g.,* what to do in the case of unavailable hosts, or in the case of topology changes. We present now a summary of some of the most important problems that we have found based in our work on the AWED model and invasive composition of distributed patterns::

1. Events (join-points in AO-speak) cannot be delivered to all the interested hosts (because of availability). This can lead to problems if those events should modify the behavior of other systems.
2. The correct version of the aspects should be available in order to react to the received events and broadcast event information.
3. Unrestricted access from external systems, aspects are powerful mechanisms, so it is normal that we want to have ways to control or restrict who and when can modify a system, e.g. A message from an unauthorized external entity should not be able to modify the core functionalities of a system.
4. Reactions (advices in AO-speak) cannot be processed on the designated hosts, due to (1) host unavailability or (2) different software versions, or programming errors which make those non-applicable or simply wrong.

3. TOWARDS AN ACTOR-BASED MODEL FOR DISTRIBUTED ASPECTS

Most of the existing research on distributed aspects focuses on the power of expression of the abstractions used to deal with the modularization of complex interactions between components and aspects in distributed settings. Nevertheless, some technical and practical consequences of the introduction of distributed aspects have not been profoundly studied. Below we present a set of characteristics that we consider important in order to achieve a robust model but that have been addressed only rudimentarily in existing systems:

- *Scoping*: A system with many participants and interactions needs to be controlled to be managed. One approach to achieve this is to support a model of domains for interactions between groups of participants, akin to ideas like ambients in mobile programming [5].
- *Controlled Invasiveness*: The level of invasiveness of aspects should be controllable because in many cases application programmers don't want the semantics of the core part of their programs to be modified arbitrarily and prefer to use explicit hooks to create aspect-aware components (similar to the formal model of open modules [2]. Controlled invasion is particularly important in the presence of stringent security requirements, which are crucial for many application domains, such as large-scale service compositions.
- *Mobility*: Aspects should be able to move and to be applied between different parts of the system in order to improve not only its expressive power, but also its scalability. This is essential given the dynamicity of distributed systems.

- *Fault-Tolerance*: Communication delays and failures are the norm, not the exception, in many distributed systems and should be considered as an essential part in the design of a pragmatic model.

Our thesis is that the actor model provides a certain number of concepts and mechanisms that can be integrated advantageously with distributed aspects in order to make them more robust. In the following, we therefore briefly review the characteristics of the actor model and then introduce those of our actor-aspect model.

3.1 A brief introduction to actor models

The actor model has been introduced by Hewitt et al. [7] as a model for concurrent computing. Actors are independent entities that encapsulate a state and a thread of control. They communicate with one another via asynchronous message passing, There are three laws that govern how an actor responds to a message: (1) they can operate at a local level, altering its current state and behavior, (2) they can send messages to other actors asynchronously, and (3) they can create other actors.

Many approaches have explored the capabilities of the actor model and refined its properties in the presence of particular conditions (e.g. migration of actors, changes of topology, guaranteed delivery of messages, reordering of messages, among others. [3, 11, 6]).

We consider that the properties of the actor model are appropriate to fix some of the problems of the existing distributed aspect models *e.g., asynchronous* communication enables the decoupling of communication actions that is a big very useful at the level of distributed aspects. Another important property is the *absence of shared state* between actors, that provides a strict model to control the access to the actor state. This is essential to control the interaction between aspects and to a clean definition of aspect instantiation models. *Locality* is another useful property given that actors can only send messages to actors they know, this creates a concept of group which can be used to limit the scope of aspect application, as well as to define rules for the activation or application of collaborative aspects. *Migration* of actors is one of the most important properties in order to support heterogeneous and dynamic systems where actors can be moved and applied in different points. Finally *concurrency* is a natural characteristic of the actor model which can be exploited to reduce the overhead of the presence of aspects.

The actor model has some weak points that affect our design *e.g.,* there are no guarantees on the order of the received messages. Another issue is that the control flow of an actor-based application often becomes difficult to follow. Finally, another issue arises in systems who need synchronous semantics as the actor model does note support it natively.

To the best of our knowledge, no existing aspect-oriented approach combines an expressive actor-like model with invasive access that supports simple and sophisticated compositions. Furthermore, we consider the presence of failure as an essential feature of our design. This is an important idea that has been elaborated only very rudimentarily in other distributed aspect models, a notable exception being work on aspect deployment [16].

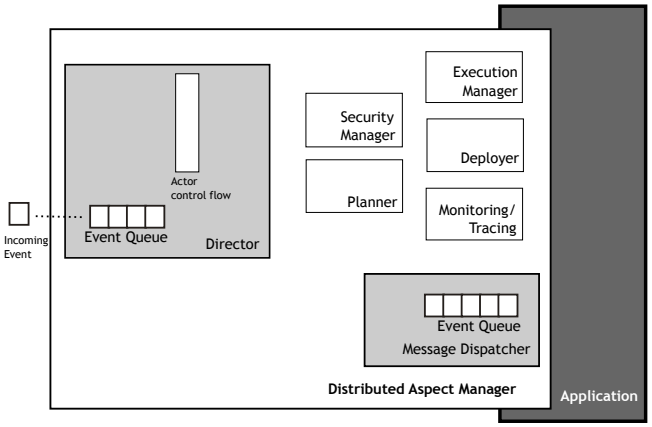

Figure 1: Architecture of our actor-based framework for distributed aspects

3.2 Characteristics of a language and execution model for actor-based aspects

Using the common aspect language and execution dimensions introduced in [4], we present some of the desired characteristics of our model derived from our design goals, and the selection of an actor-based approach.

The joinpoint model should support both static and dynamic joinpoints. Static joinpoints identify execution events, such as concrete method/function executions. Dynamic joinpoints evaluate runtime conditions which in the case of distributed aspects address not only state-related conditions, but also conditions involving locations. The pointcut language should allow the selection of events but also sequences of related events in order to respond to complex conditions involving multiple hosts. Such event sequences permit to catch situations such as a user authenticating in a given location and trying to execute a service in a different location.

Executing aspects can be mapped one-to-one to actors. In this case, we can exploit the actors' properties and need not synchronize the access to the shared mutable state: aspect instances are thus ready for concurrency, and finer instantiation models can be supported *e.g.,* per-object, per-request and in particular per-sequence because sequences can be naturally encoded as state machines inside each actor.

Aspect composition is facilitated given the decoupled communication characteristics of actors: in most of the cases it would only be necessary to create or forward messages between actors. Deployment strategies should support dynamic weaving and take advantage of the mobility properties of the actor model.

3.3 Towards an actor-based architecture for distributed aspects

We designed an extensible architecture for aspects and actors via an aspect manager that should be installed in every host, ideally but not necessarily as part of a middleware platform, and that is going to manage execution concerns, such as communication, security, scoping, monitoring, etc.

The model of the figure 1 presents two main communication points: A component that receives incoming messages, *e.g.,* information about events (joinpoints), new aspects to deploy, or internal communication messages, *e.g.,* about changes in the topologies. The main role of this com-

ponent (called director in the figure) is a broker that forwards messages to the corresponding parts of the system. Another important communication component is the message dispatcher. It decouples the sending of messages between nodes, to guarantee their delivery in the case of host unavailability.

The rest of the components in the figure can be classified in two groups: a group in charge of security and monitoring, and a group in charge of deploying and management of the life-cycle of aspects, including weaving and activation.

The first group is in charge of the validation of the permissions and the scope of the demanded actions *e.g.,* messages from unrelated hosts or actions that should be ignored, or those managing which hosts have the rights to deploy aspects. It also monitors the system for anomalies and sends data to the second group. The second group controls the changes in the topology and the versions of the aspects, this implies the planning of the best strategies to be applied — which may include optimization ideas like batching of sequences of messages and alternative processing strategies — the deployment and activation of aspects in the system.

Our model observes a number of interesting correctness properties, in particular, because it supports guaranteed delivery of events in the case of unavailable hosts or communication errors. It therefore addresses the issues 1 and 4 (delivery problems) presented in section 2. A control of versions is given by a combination of our security actor, and the implicit isolation properties of the actor model: our models thus covers the issues 2 and 3. The desirable technical characteristics of the model that we described previously in this section are also addressed in our model in the following way: fault tolerance thanks to the asynchronous nature of actors and its interaction properties. Manageability is addressed by means of the director and the planner components, controlled invasiveness is achieved thanks to the locality properties of the model, the deployer component, and the security manager. Finally, mobility is addressed by the actor model in a native way, but is enforced also by the security component.

4. CONCLUSION AND FUTURE WORK

In this paper we have detailed some problems of current distributed aspect models in the presence of common requirements of distributed systems like fault-tolerance and security. We introduced the actor model and we explained why an actor-based design is appropriate to build a more robust infrastructure for distributed aspects. We also introduced a small set of desired properties and a suitable architecture for an actor-based distributed aspect language and corresponding execution environment.

Many areas still remain to be studied in our design: several features of actors have to be explored in further detail, *e.g.,* how to encode synchronous behavior, how to deal with intra-actor dependencies. We plan to address these issues using a more expressive integrative model and develop a first implementation of our model.

5. REFERENCES

[1] G. Agha. Actors: a model of concurrent computation in distributed systems. *AITR-844*, 1985.

[2] J. Aldrich. Open modules: Modular reasoning about advice. *ECOOP 2005-Object-Oriented Programming*, pages 144–168, 2005.

[3] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. Concurrent programming in Erlang. 1993.

[4] J. Brichau and M. Haupt. Survey of aspect-oriented languages and execution models. *European Network of Excellence in AOSD*, 2005.

[5] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures*, pages 253–292. Springer, 1998.

[6] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. *ECOOP 2006–Object-Oriented Programming*, pages 230–254, 2006.

[7] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[8] G. Kiczales et al. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *11th Europeen Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, pages 327–353, 2001.

[10] B. Luis et al. Explicitly distributed aop using awed. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, Mar. 2006. ACM Press.

[11] M. Miller and J. Shapiro. *Robust composition: Towards a unified approach to access control and concurrency control*. Citeseer, 2006.

[12] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut: a language construct for distributed AOP. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 7–15. ACM New York, NY, USA, 2004.

[13] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier. Jac: A flexible solution for aspect-oriented programming in java. *MetaLevel Architectures and Separation of Crosscutting Concerns*, pages 1–24, 2001.

[14] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 174–190. ACM, 2002.

[15] É. Tanter, J. Fabry, R. Douence, J. Noyé, and M. Südholt. Expressive scoping of distributed aspects. In *AOSD2009*, March 2009.

[16] E. Truyen and W. Joosen. Run-time and atomic weaving of distributed aspects. *Transactions on Aspect-Oriented Software Development II*, pages 147–181, 2006.